

现代操作系统 (Modern Operating System)

福州大学 数计学院 江兰帆



第2章 进程与线程

2.1 进程

2.2 线程

2.3 进程间通信

2.4 经典的IPC问题

2.5 调度

2.6 有关进程和线程的研究

2.7 小结

2.1 进程



2.1 进 程

2.1.1 进程模型

2.1.2 进程的创建

2.1.3 进程的终止

2.1.4 进程的层次结构

2.1.5 进程的状态

2.1.6 进程的实现

引言

- 程序：
 - 指令或语句序列，体现了某种算法，所有程序是顺序的
- 顺序环境：
 - 在计算机系统中只有一个程序在运行，这个程序独占系统中所有资源，其执行不受外界影响。

程序顺序执行的特征

- 顺序性：处理机的操作严格按照程序所规定的顺序执行，只有当上一个操作完成后，下一操作才能开始执行。
- 封闭性：程序运行时独占全机资源，资源的状态（除初始状态外），只有本程序才能改变。
- 可再现性：只要程序执行时的初始条件和执行环境相同，一定会得到相同的结果。

程序的并发执行

引入并发的目的：

引入并发是为了提高资源利用率，从而提高系统效率。

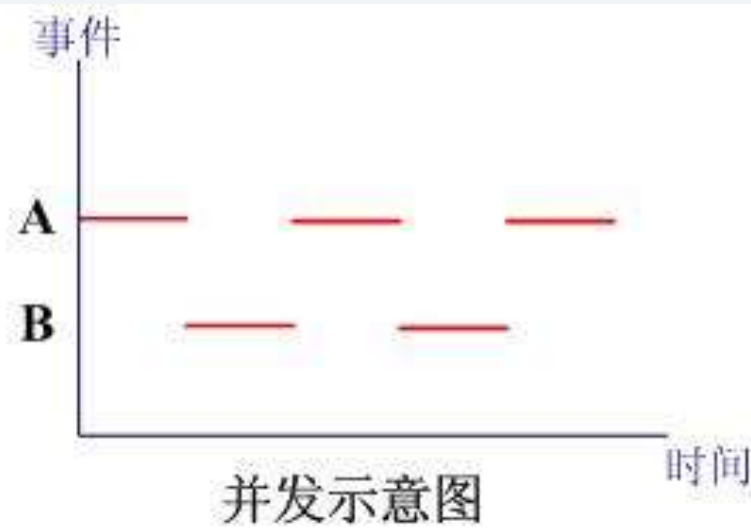
并发与并行概念的区别：

concurrency, parallel

并行与并发的概念差别

- 并行 (Parallel)
 - 同一时刻，两个事物均处于活动状态
 - 示例：CPU中的超标量设计
- 并发 (Concurrency)
 - 宏观上存在并行特征，微观上存在顺序性：即同一时刻，只有一个事物处于活动状态
 - 示例：分时操作系统中多个程序的同时运行

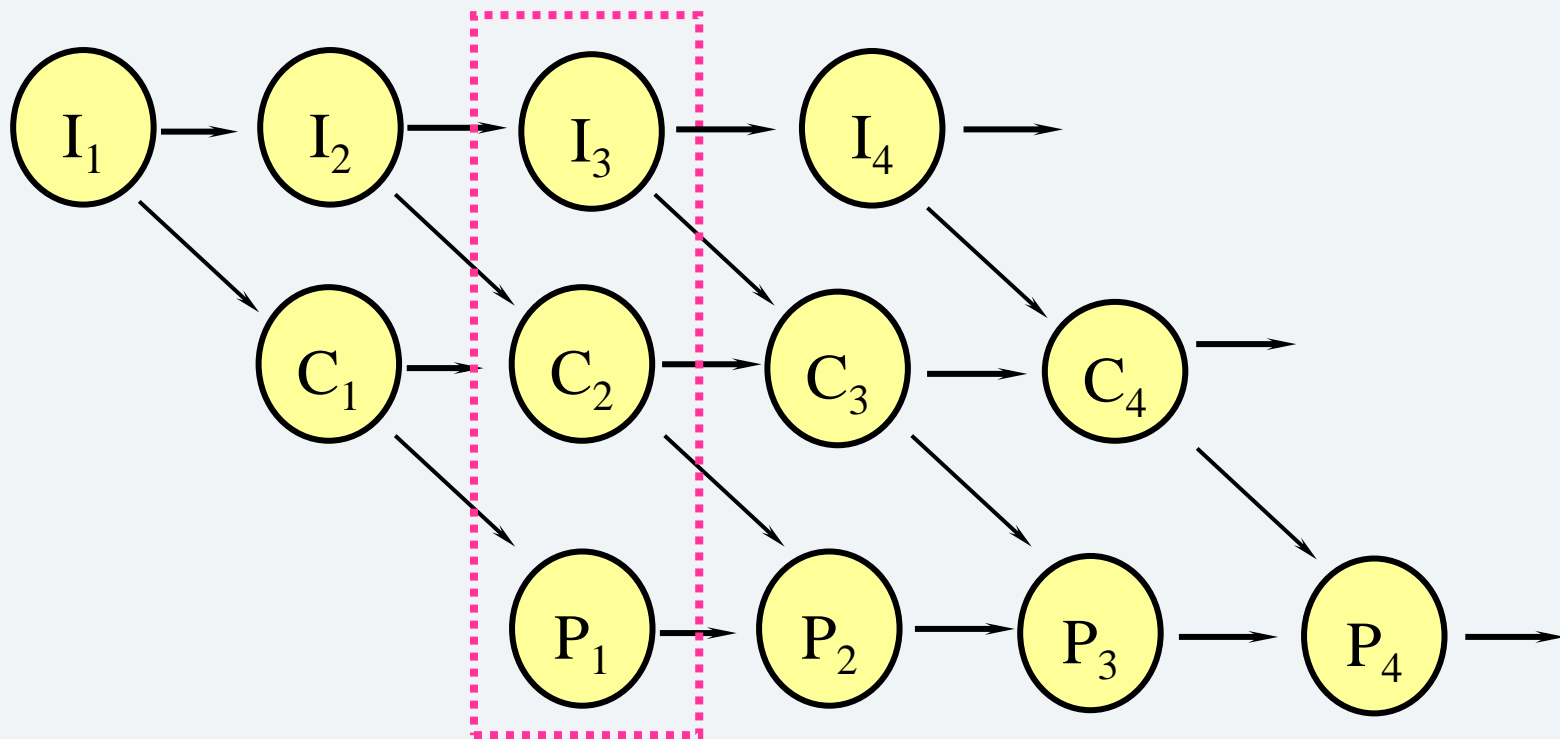
并行与并发的概念差别



程序的并发执行

- 所谓程序的并发执行是指：若干个程序同时在系统中执行，这些程序的执行在时间上是重叠的，一个程序的执行尚未结束，另一个程序的执行已经开始。

程序的并发执行



例：程序A、B，共享变量N。代码如下：

程序A

BEGIN

REPEAT

...

...

N: =N+1

...

...

UNTIL FALSE

END

程序B

BEGIN

REPEAT

...

...

N:=0

PRINT(N)

...

UNTIL FALSE

END

程序的并发执行

两个程序以不同速度运行，可能出现三种情况：

- $N := N + 1$ 在 $N := 0$ 和 $\text{Print}(N)$ 之前
——输出的N值为0
- $N := N + 1$ 在 $N := 0$ 和 $\text{Print}(N)$ 之后
——输出的N值为0
- $N := N + 1$ 在 $N := 0$ 和 $\text{Print}(N)$ 之间
——输出的N值为1

程序并发执行的特征

- 间断性：由于资源共享和相互合作，并发执行的程序间形成了相互制约关系，导致程序的运行过程出现“执行—暂停—执行”的现象。
- 失去封闭性：程序在并发执行时，是多个程序共享系统中的资源，因此这些资源的状态将由多个程序来改变。
- 不可再现性：由失去封闭性导致。同样的初始条件，一个程序的多次重复执行，可得到不同的结果。

2.1.1 进 程 模 型

- 为什么引入进程？
 - 为了使程序能正确地并发执行，且对并发执行的程序加以描述和控制。

进程的概念

- 定义：进程(process)是程序在一个数据集上运行的过程，它是系统进行资源分配和调度的一个独立单位。
- 即：一个进程就是一个正在执行的程序，包括程序计数器、寄存器和变量的当前值。

2.1.1 进程模型

进程的核心思想：

- 进程是某种类型的一个活动，它有程序、输入、输出和状态。
- 在分时操作系统中，单个CPU被若干进程共享，它使用某种调度算法决定何时停止一个进程的运行，转而为其他进程提供服务。

进程同程序的比较

- 进程是动态的，程序是静态的：进程是程序的一次执行过程；程序是有序代码的集合。
- 进程是暂时的，程序是永久的：进程是一个状态变化的过程，是有一定生命期的；而程序可以作为一种软件资料长久保存。

进程同程序的比较

- 进程与程序的组成不同：进程是由程序和数据、进程控制块三部分组成的。
- 进程与程序的对应关系：一个程序可对应多个进程，反之亦然。进程具有创建其它进程的功能，而程序不能形成新的程序。

2.1.2 进程的状态

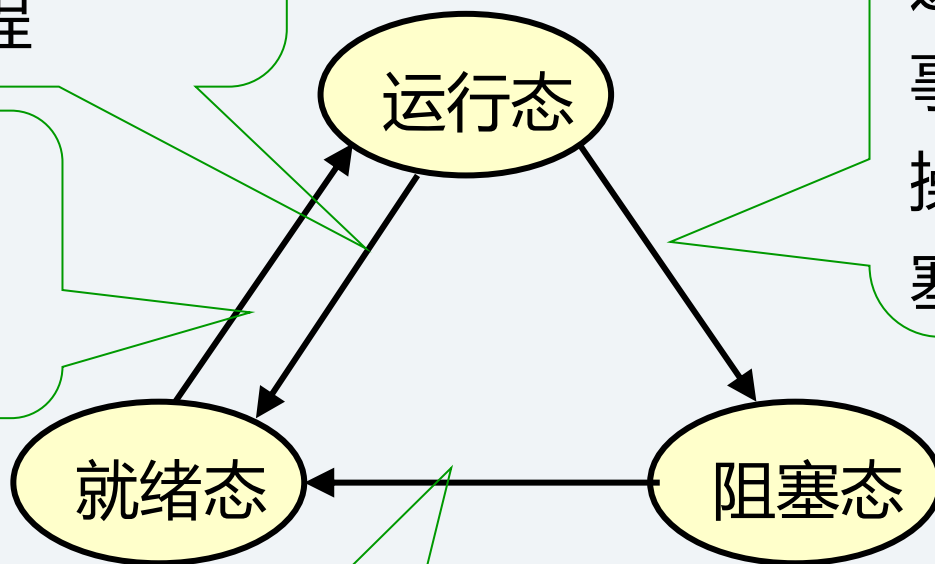
进程的三种基本状态：

- 就绪状态：进程已获得除 CPU 以外的所有必要资源，只要得到 CPU，便可立即执行。
- 执行状态：进程已得到 CPU，其程序正在CPU 上执行。
- 阻塞状态：正在执行的进程因某种事件(如I/O 请求)的发生而暂时无法继续执行，只有等相应事件完成后，才能去竞争 CPU。

进程状态转换图

时间片已用完或
调度程序选择另
一个进程

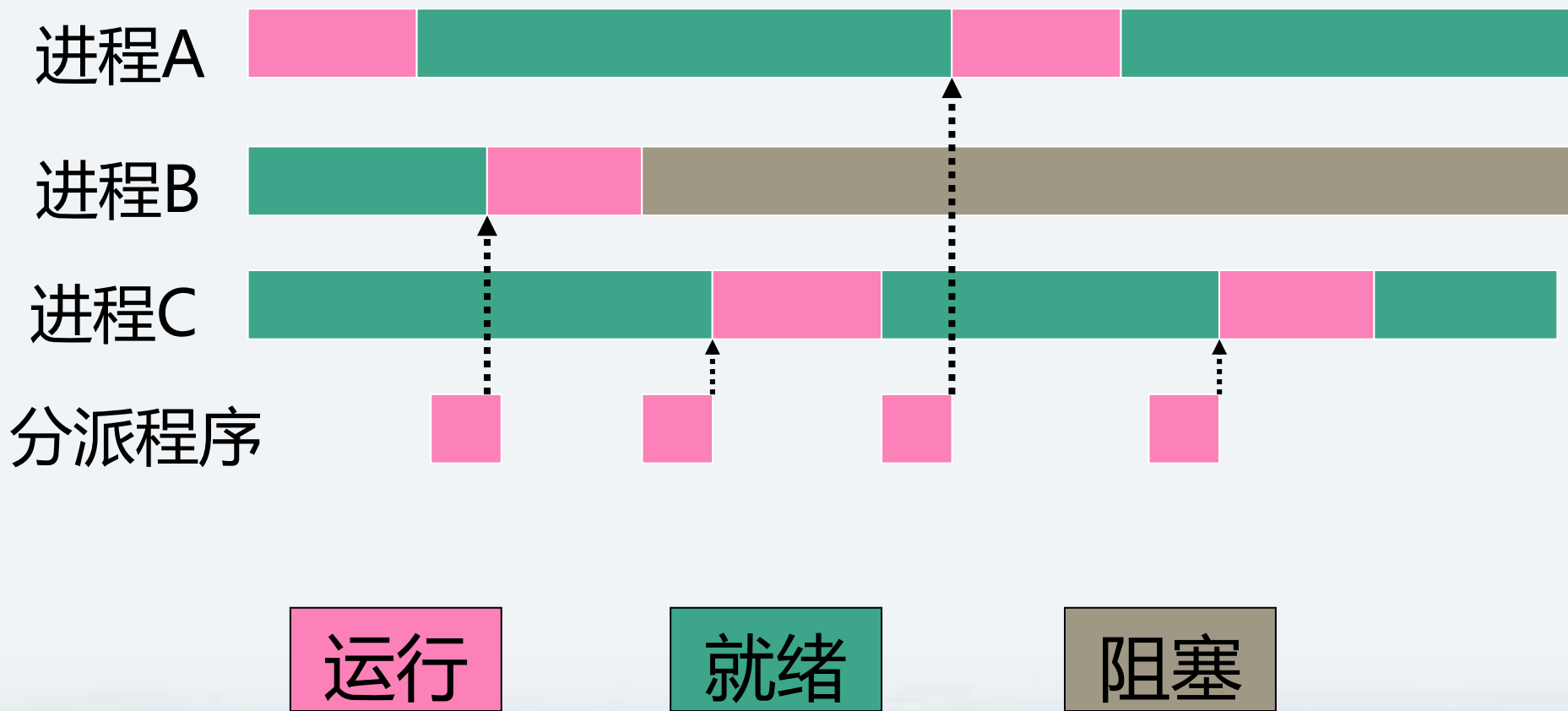
进程调度：决
定把CPU分配
给哪个进程



进程因等待某
事件（如I/O
操作）变成阻
塞状态

某事件被解除
(如I/O完成)

进程的状态转换



【 思考题 】

单处理机系统中如果有N个进程，则：

- 运行的进程最多几个，最少几个？

----- 1, 0

- 就绪进程最多几个，最少几个？

----- N-1, 0

- 等待进程最多几个，最少几个？

----- N, 0

【 思考题 】

有没有这样的状态转换，为什么？

- 阻塞→运行
- 就绪→阻塞

2.1.3 进程的实现

概念的引入

- 进程上下文(context): 上下文是指进程运行的环境。例如, 针对x86的CPU, 任务上下文可包括程序计数器、堆栈指针、通用寄存器的内容。

概念的引入

- 上下文切换（Context Switching）：多任务系统中，上下文切换是指CPU的控制权由运行进程转移到另外一个就绪进程时所发生的事件，当前运行进程转为就绪（或者阻塞）状态，另一个被选定的就绪进程成为当前运行进程。上下文切换包括保存当前进程的运行环境，恢复将要运行进程的运行环境。

2.1.3 进程的实现

1. 进程控制块的概念

- 进程控制块 (Process Control Block, PCB) 是系统为了管理进程设置的一个专门的数据结构，它记录了操作系统所需的用于描述进程的当前情况以及控制进程运行的全部信息。
- 进程与PCB是一一对应的

2.1.3 进程的实现

- 进程控制块（PCB）的作用是使一个在多道程序环境下不能运行的程序（含数据），成为一个能独立运行的基本单位，一个能与其它进程并发执行的进程。

进程的组成

- Process Control Block：进程存在的唯一标志。
- 程序：描述了进程要完成的功能，是进程执行时不可修改的部分。
- 数据：进程执行时用到的数据（用户输入的数据、常量、静态变量）。
- 工作区：参数传递、系统调用时使用的动态区域（堆栈区）。



2.1.3 进程的实现

2. 进程控制块中的信息

- 进程标识符：用于唯一地标识系统中的每个进程。
 - 内部标识符：数字，通常是进程序号，方便系统使用。
 - 外部标识符：字母、数字。由创建者提供，方便用户使用。

进程控制块中的信息

- 处理机状态：主要由处理机中各种寄存器的内容组成，用于 CPU 切换时保存现场和恢复现场。
 - 通用寄存器
 - 指令计数器
 - 程序状态字
 - 用户栈指针

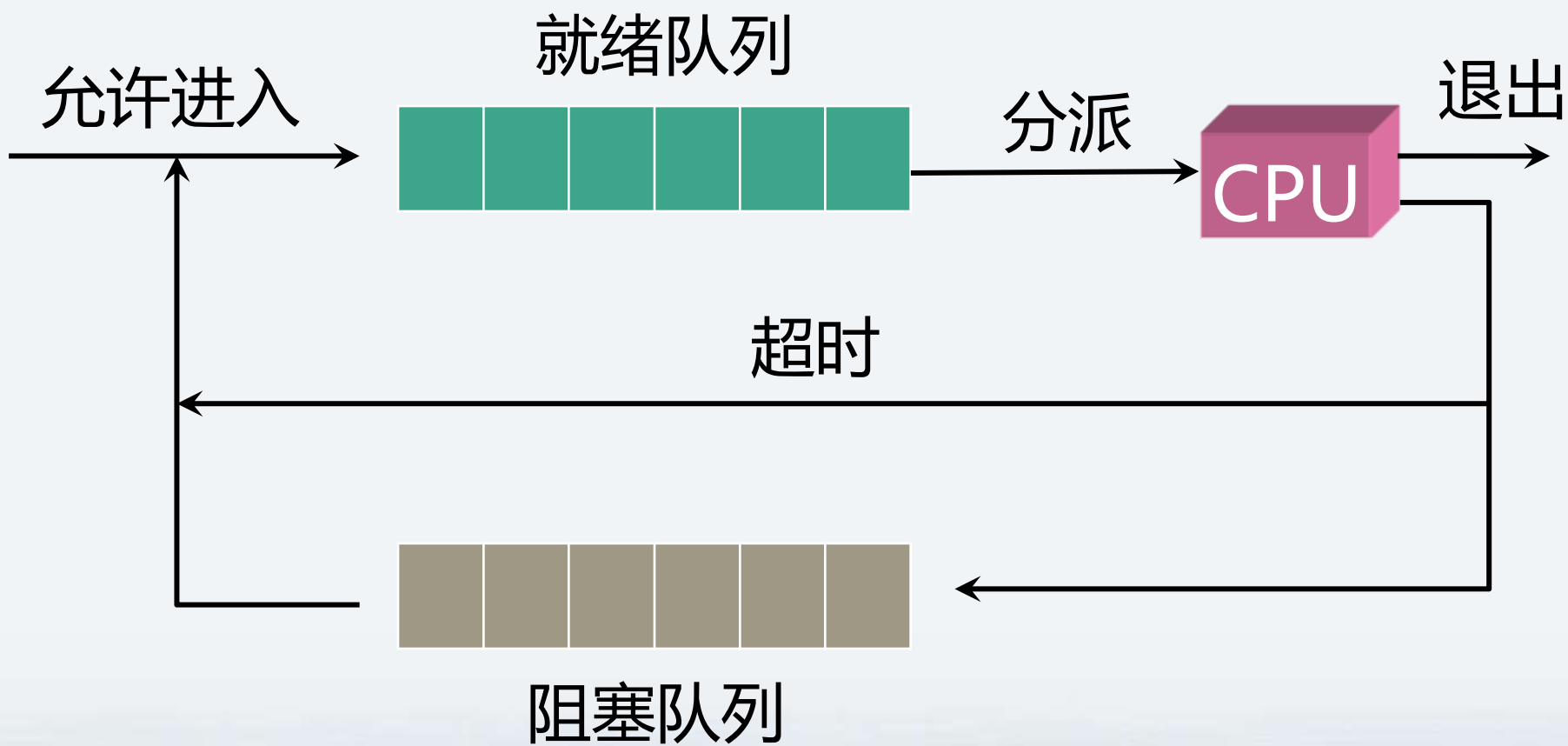
进程控制块中的信息

- 进程调度信息：用于进程调度和进程对换的相关信息
 - 进程状态
 - 进程优先级
 - 进程等待和使用 CPU 的时间总和
 - 阻塞原因

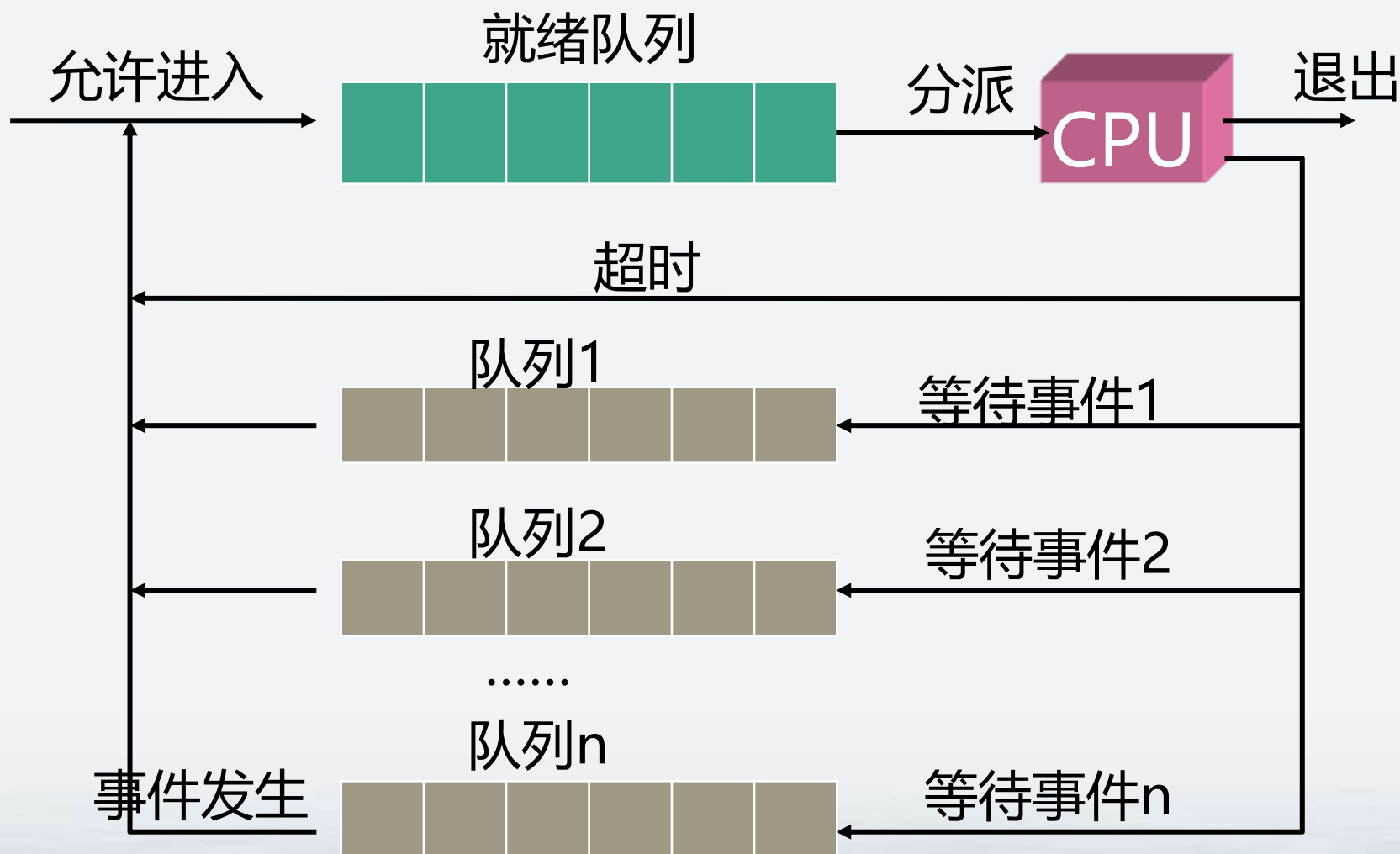
进程控制块中的信息

- 进程控制信息：
 - 程序和数据的地址
 - 进程同步和通信信息
 - 资源清单
 - 进程队列指针

单一阻塞队列



多阻塞队列



2.1.3 进程的实现

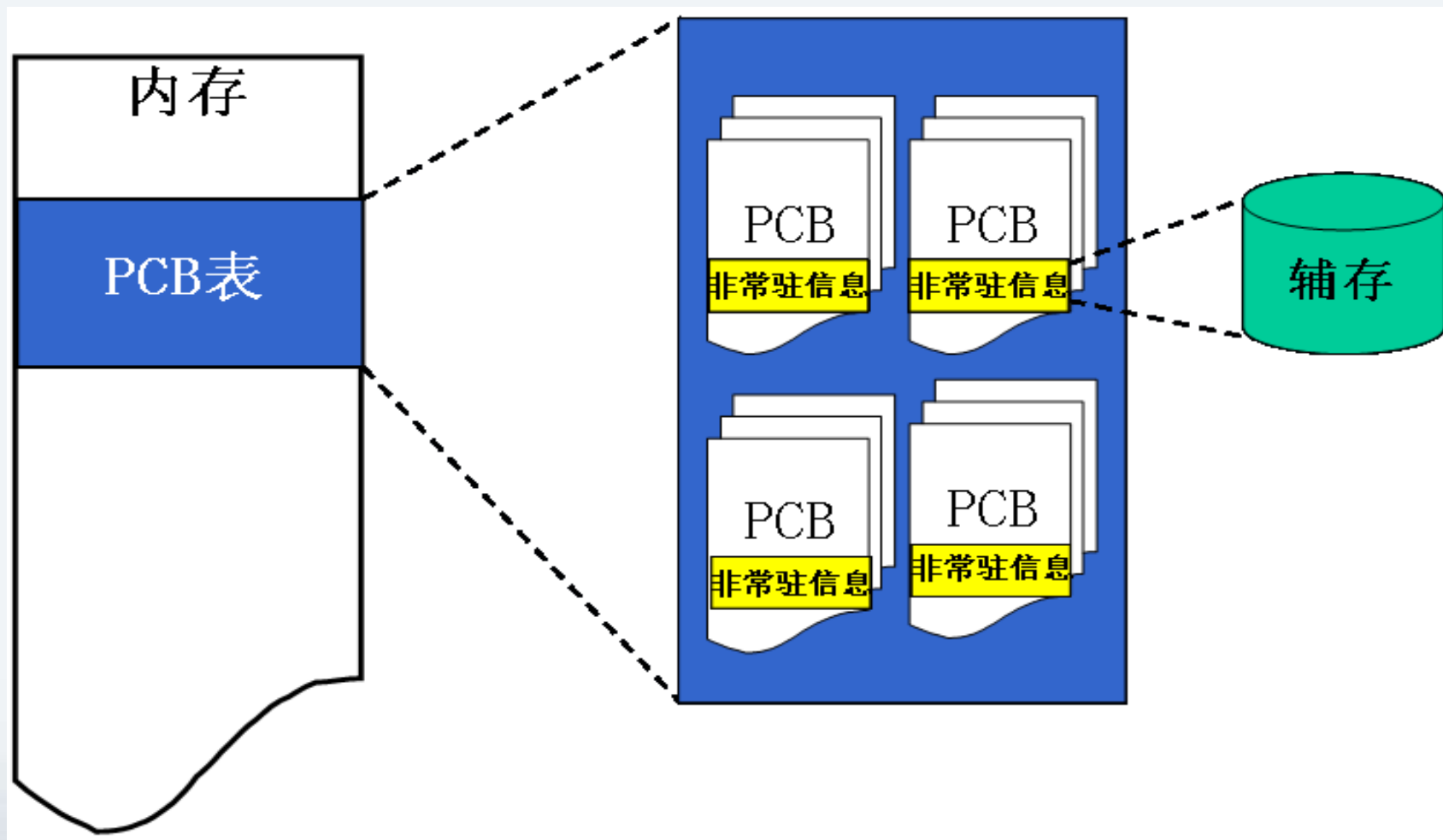
3. 进程控制块的组织方式

- 在一个系统中通常有许多的 PCB，为了有效地进行管理，系统必须用适当的方式将 PCB 组织起来，常用的方式有链接方式和索引方式。

进程控制块的组织方式

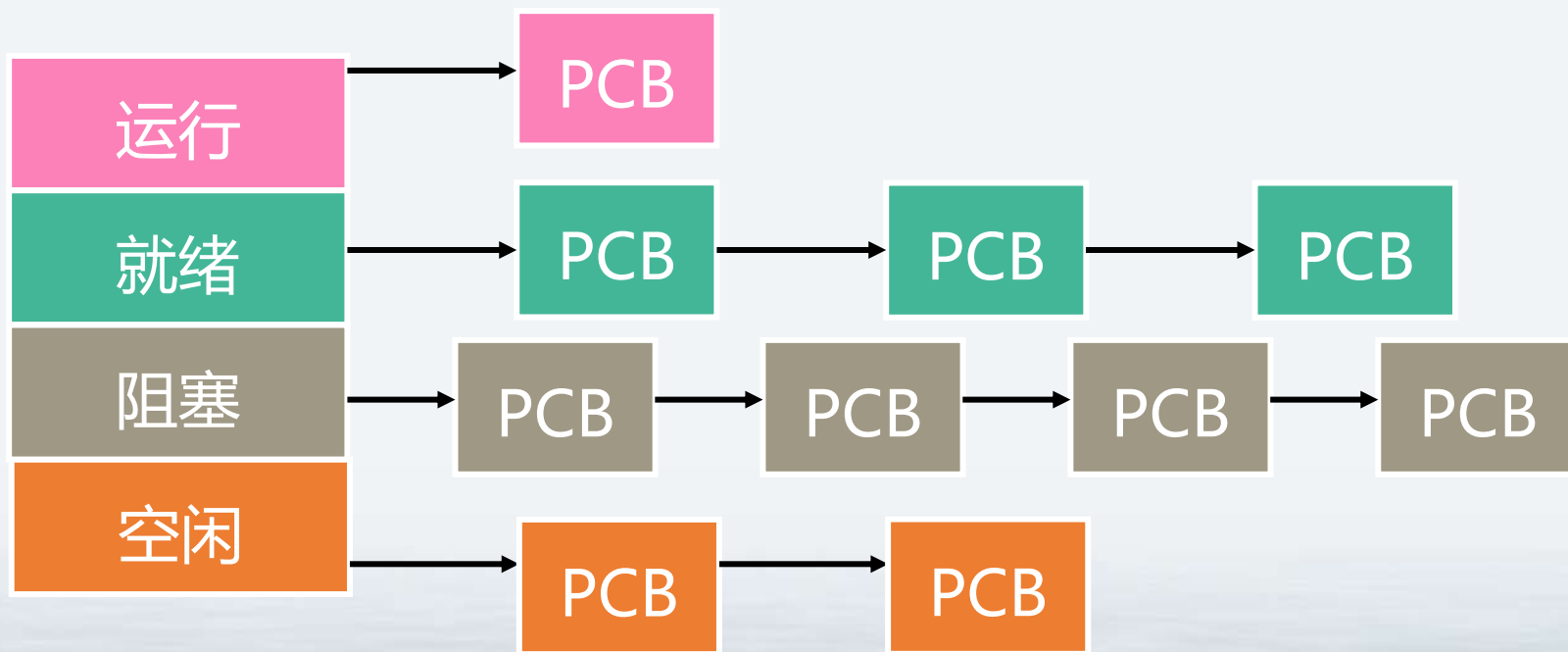
- PCB表：
 - 系统把所有PCB组织在一起，并把它们放在内存的固定区域，就构成了PCB表
- PCB表的大小决定了系统中最多可同时存在的进程个数，称为系统的并发度。

操作系统对PCB的管理：集中统一



进程控制块的组织方式

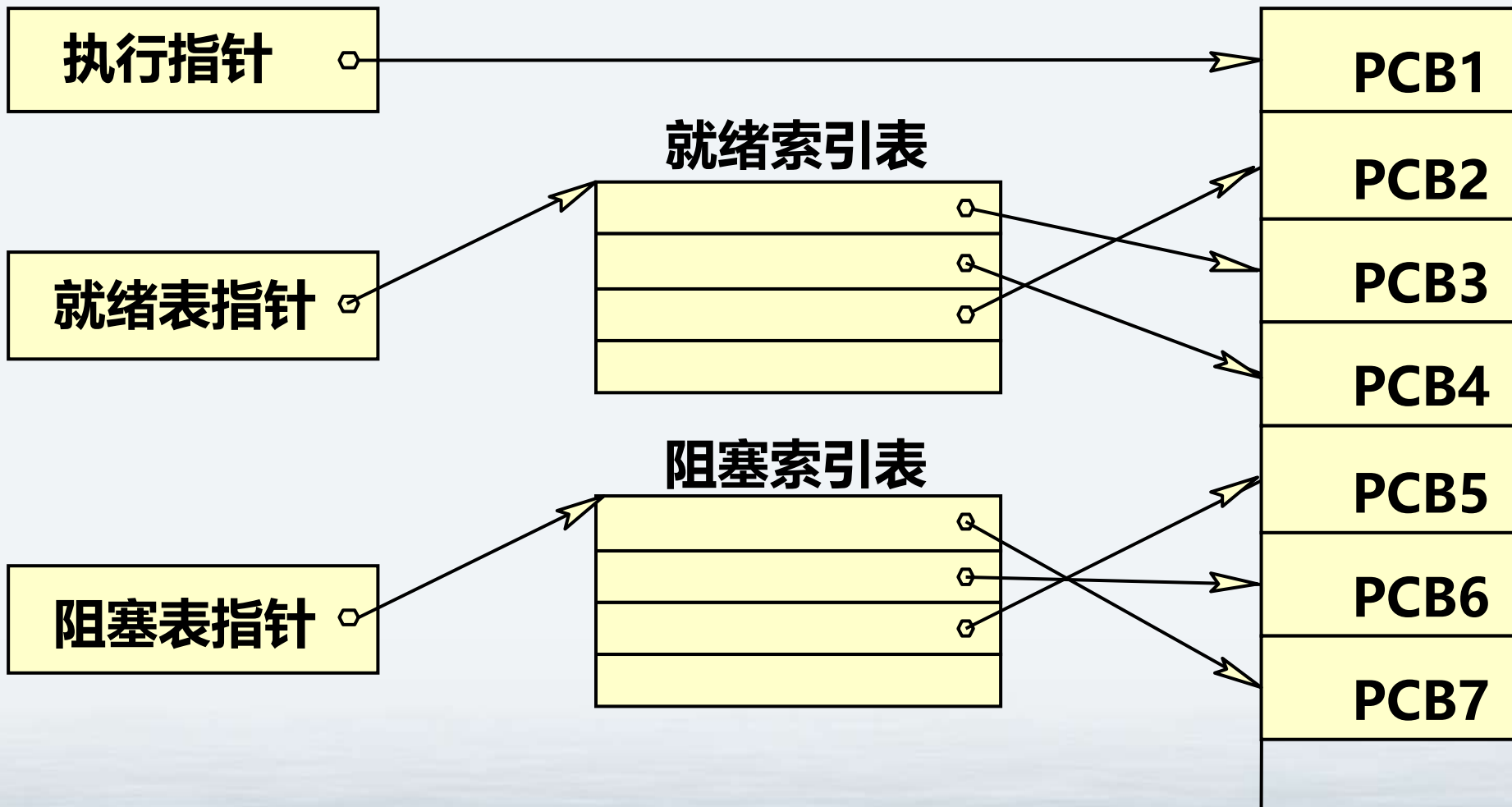
- 链接方式：相同状态的进程PCB组成一个链表，不同状态对应多个不同的链表。



进程控制块的组织方式

- 索引方式：对具有相同状态的进程，分别设置各自的PCB索引表，表明PCB在PCB表中的地址。

索引方式



2.1.4 进程的创建

引起创建进程的原因：

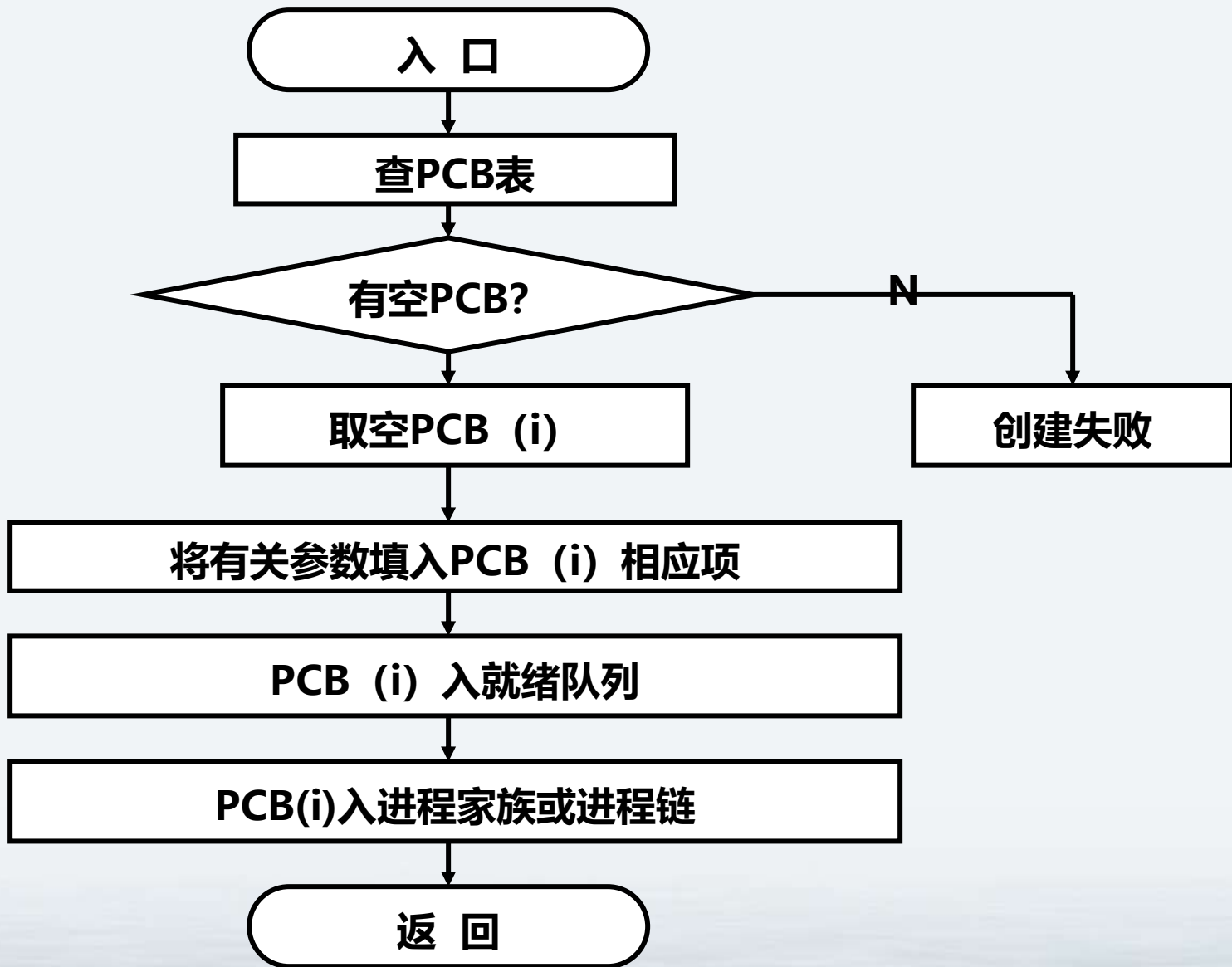
- 系统初始化；
- 当前运行的进程执行一个创建进程的系统调用；
- 用户请求创建一个新进程；
- 一个批处理作业的初始化。

2.1.4 进程的创建

- 创建新进程通过进程创建原语来完成，其主要任务是创建进程控制块 PCB。

- ① 申请空白PCB。
- ② 为新进程分配资源。
- ③ 初始化进程控制块。
- ④ 将新进程插入就绪队列。

进程的创建



2.1.5 进程的终止

- 当某进程完成任务正常结束时，或在运行过程中遇到某些异常情况，或外界干预需要结束时，应予以终止（撤消）。
- 通过进程终止原语来终止进程。
- 终止进程的实质是收回 PCB。

2.1.5 进程的终止

正常结束

- 进程运行完毕退出

异常结束

- 无内存
- 越界
- 保护错误
- 算术错误
- I/O失败
- 无效指令
- 特权指令
- 数据误用

外界干预

- 操作员或操作系统干预
- 父进程请求
- 父进程终止

2.1.5 进程的终止

- 进程的终止过程:

- ① 查找对应的PCB
- ② 终止该进程及子孙进程
- ③ 释放资源
- ④ 回收PCB

Linux进程控制块PCB

- Linux在内存空间中开辟了一个专门的区域存放所有进程的PCB----task_struct结构。
- task_struct结构是一个复杂的结构，占一千多字节，其各个成员用来准确描述进程在各方面的信息。主要有以下几个部分 (include/linux/sched.h文件)：

task_struct

1. 进程的状态和标志

volatile long state

//进程的状态: -1 unrunnable, 0 runnable.....

unsigned long flags

//进程的状态标志

task_struct

2. 进程的标识

pid_t pid //进程标识号

pid_t tgid //线程组的领头线程的标识号

unsigned short uid, gid //用户标识号, 组标识号

unsigned short euid, egid //用户有效标识号, 组有效标识号

unsigned short suid, sgid //用户备份标识号, 组备份标识号

unsigned short fsuid, fsgid //用户文件标识号, 组文件标识号

task_struct

- 3. 进程的族亲关系

struct task_struct *p_opptr //指向祖先进程PCB的指针

struct task_struct *p_pptr //指向父进程PCB的指针

struct task_struct *p_cptr //指向子进程PCB的指针

struct task_struct *p_ysptr //指向弟进程PCB的指针

struct task_struct *p_osptr //指向兄进程PCB的指针

task_struct

4. 进程间的链接信息

struct task_struct *next_task //指向下一个PCB的指针

struct task_struct *prev_task //指向上一个PCB的指针

struct task_struct *next_run

//指向可运行队列的下一个PCB的指针

struct task_struct *prev_run

//指向可运行队列的上一个PCB的指针

task_struct

5. 进程的调度信息

- long counter //时间片计数器
- long nice //进程优先级
- unsigned long rt_priority //实时进程的优先级
- unsigned long policy //进程调度策略

task_struct

6. 进程的时间信息

- long start_time //进程创建的时间
- long utime //进程在用户态下耗费的时间
- long stime //进程在核心态下耗费的时间
- long cutime //所有子进程在用户态下耗费的时间
- long cstime //所有子进程在核心态下耗费的时间
- unsigned long timeout //进程申请延时

task_struct

7. 进程的虚存信息

struct mm_struct *mm //进程的虚存信息

struct desc_struct *ldt //进程的局部描述符表指针

unsigned long saved_kernel_stack

//核心态下堆栈的指针

unsigned long kernel_stack_page

//核心态下堆栈的页表指针

task_struct

8. 进程的文件信息

```
struct fs_struct *fs
```

//进程的可执行映象所在的文件系统

```
struct files_struct *files
```

//进程打开的文件

task_struct

9. 与进程间通信有关的信息

unsigned long signal //进程接收到的信号

unsigned long blocked //阻塞信号的掩码

struct signal_struct* sig //信号处理函数表的指针

int exit_signal //进程终止的信号

struct sem_undo *semundo //进程要释放的信号量

struct sem_queue *semsleeping

//与信号量操作相关的等待队列

task_struct

10. 其它信息

- int errno //系统调用的出错代码
- long debugreg[8] //进程的8个调试寄存器
- char comm[16] //进程接收到的信号

进程列表
struct list_head tasks

任务ID

```
pid_t pid;  
pid_t tgid;  
struct task_struct *group_leader;
```

亲缘关系

```
struct task_struct __rcu *real_parent;  
struct task_struct __rcu *parent;  
struct list_head children;  
struct list_head sibling;
```

任务状态

```
volatile long state;  
int exit_state;  
unsigned int flags;
```

权限

```
const struct cred __rcu *real_cred;  
const struct cred __rcu *cred;
```

运行统计

```
u64 utime;  
u64 stime;  
unsigned long nvcsw;  
unsigned long nivcsw;  
u64 start_time;  
real_start_time;
```

调度相关

```
int on_rq;  
int prio;  
int static_prio;  
int normal_prio;  
const struct sched_class *sched_class;  
struct sched_entity se;  
unsigned int policy;
```

信号处理

```
struct signal_struct *signal;  
struct sighand_struct *sighand;  
struct sigpending pending;
```

内存管理

```
struct mm_struct *mm;  
struct mm_struct *active_mm;
```

文件与文件系统

```
struct fs_struct *fs;  
struct files_struct *files;
```

内核栈

```
struct thread_info thread_info;  
void *stack;
```

2.2 线程



2.2 线程

2.2.1 线程模型

2.2.2 线程的使用

2.2.3 在用户空间中实现线程

2.2.4 在内核中实现线程

2.2.5 混合实现

2.2.1 线程模型

进程的两个基本属性：

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

以上两个属性构成进程并发执行的基础

2.2.1 线程模型

为了完成程序的并发，系统必须完成的操作：

- 创建进程：必须为之分配其所必需的、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结构。
- 撤消进程：必须先对进程所拥有的资源进行回收操作，然后再回收PCB结构。
- 进程切换：要保留当前进程的CPU环境和设置新选中进程的CPU环境。

为什么要提出线程的概念？

以进程为单位的并发程序设计效率不高：

- 进程时空开销大
 - 频繁调度耗费大量CPU时间
 - 空间占用大，内存资源
- 进程通信代价高
- 进程间并发粒度大

2.2.1 线程模型

设想：

- 是否可以把进程的管理和执行相分离？
- 允许一个进程中包含多个可并发执行的控制流，这些控制流切换时不必通过进程调度，通信时可以直接借助于共享内存区？

2.2.1 线程模型

- 将原来进程的两个属性分开处理!!

2.2.1 线程模型

线程的引入:

- 引入进程的目的是为了为了使多个程序并发执行，以改善资源利用率、提高系统吞吐量。
- 引入线程则是为了减少程序并发执行时的所付出的时空开销。

线程概念

- 线程是进程内一个相对独立的、可调度的执行单元，线程自己基本上不拥有资源，只拥有一点在运行时必不可少的资源（如程序计数器，一组寄存器和栈），但它可以与同属一个进程的其他线程共享进程拥有的全部资源。

2.2.1 线程模型

在一个进程中
所有线程
共享的内容

每个线程自
己的内容

Per process items

- Address space
- Global variables
- Open files
- Child processes
- Pending alarms
- Signals and signal handlers
- Accounting information

Per thread items

- Program counter
- Registers
- Stack
- State

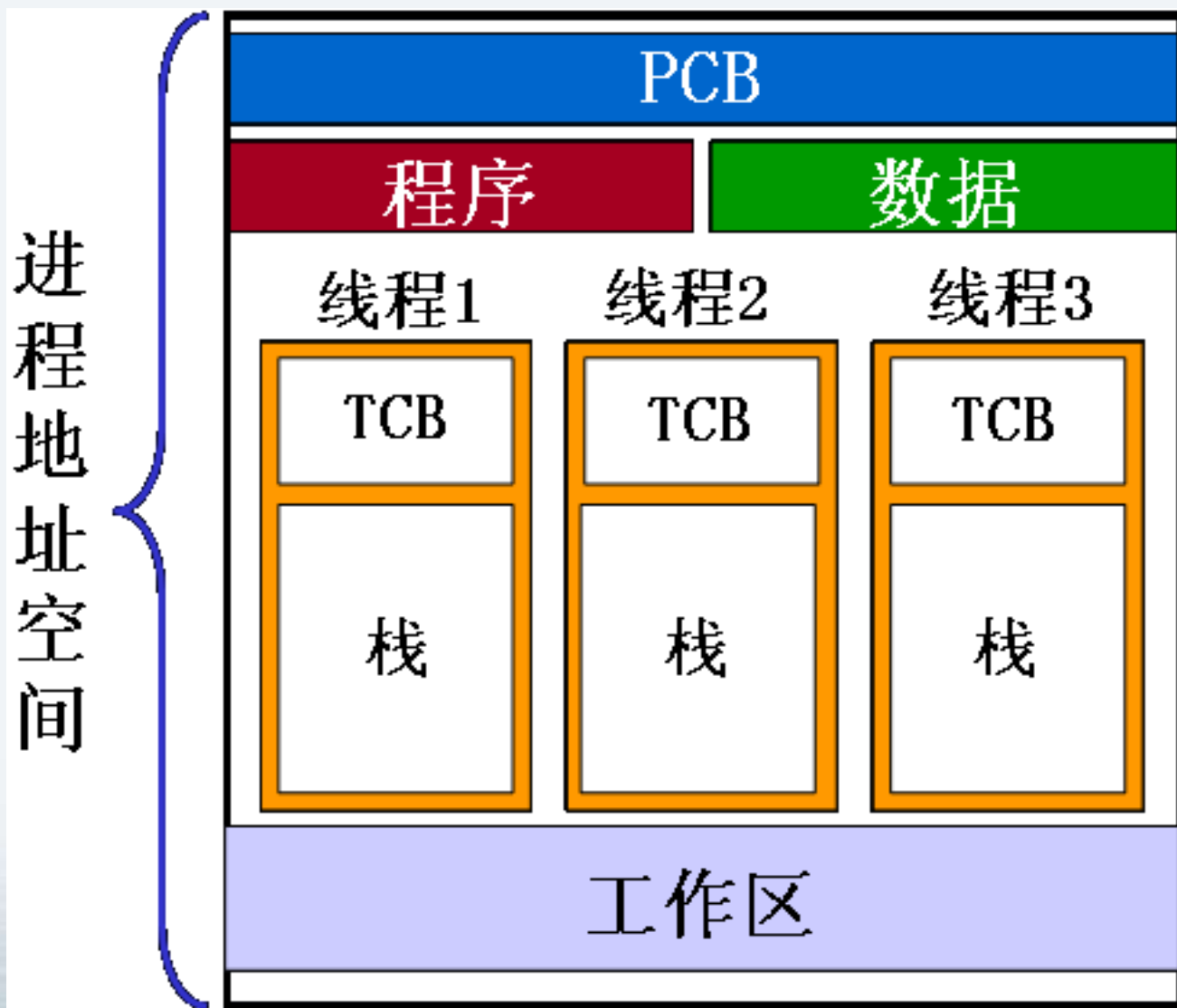
2.2.1 线程模型

- 例如，如果一个线程打开了一个文件，该文件对该进程中的其他线程都可见，这些线程可以对该文件进行读写。由于资源管理的单位是进程而非线程，所以这种情形是合理的。如果每个线程有其自己的地址空间、打开文件、即将发生的报警等，那么它们就应该是不同的进程了。线程概念试图实现的是，共享一组资源的多个线程的执行能力，以便这些线程可以为完成某一任务而共同工作。

线程和进程的关系

- ① 线程是进程的一个组成部分，线程由进程创建，因此一个进程中至少存在一个线程，线程还可以创建其它线程。
- ② 进程依然是资源分配和保护的基本单位，线程只能在进程的地址空间活动，线程只能使用其所在进程的资源。

线程的结构



每个线程都有其自己的堆栈!!

- 线程的堆栈有一帧，供各个被调用但是还没有从中返回的过程使用。在该帧中存放了相应过程的局部变量以及过程调用完成之后使用的返回地址。例如，如果过程X调用过程Y，而Y又调用Z，那么当Z执行时，供X、Y和Z使用的帧会全部存在堆栈中。通常每个线程会调用不同的过程，从而有一个各自不同的执行历史。这就是为什么每个线程需要有自己的堆栈的原因。

线程的属性

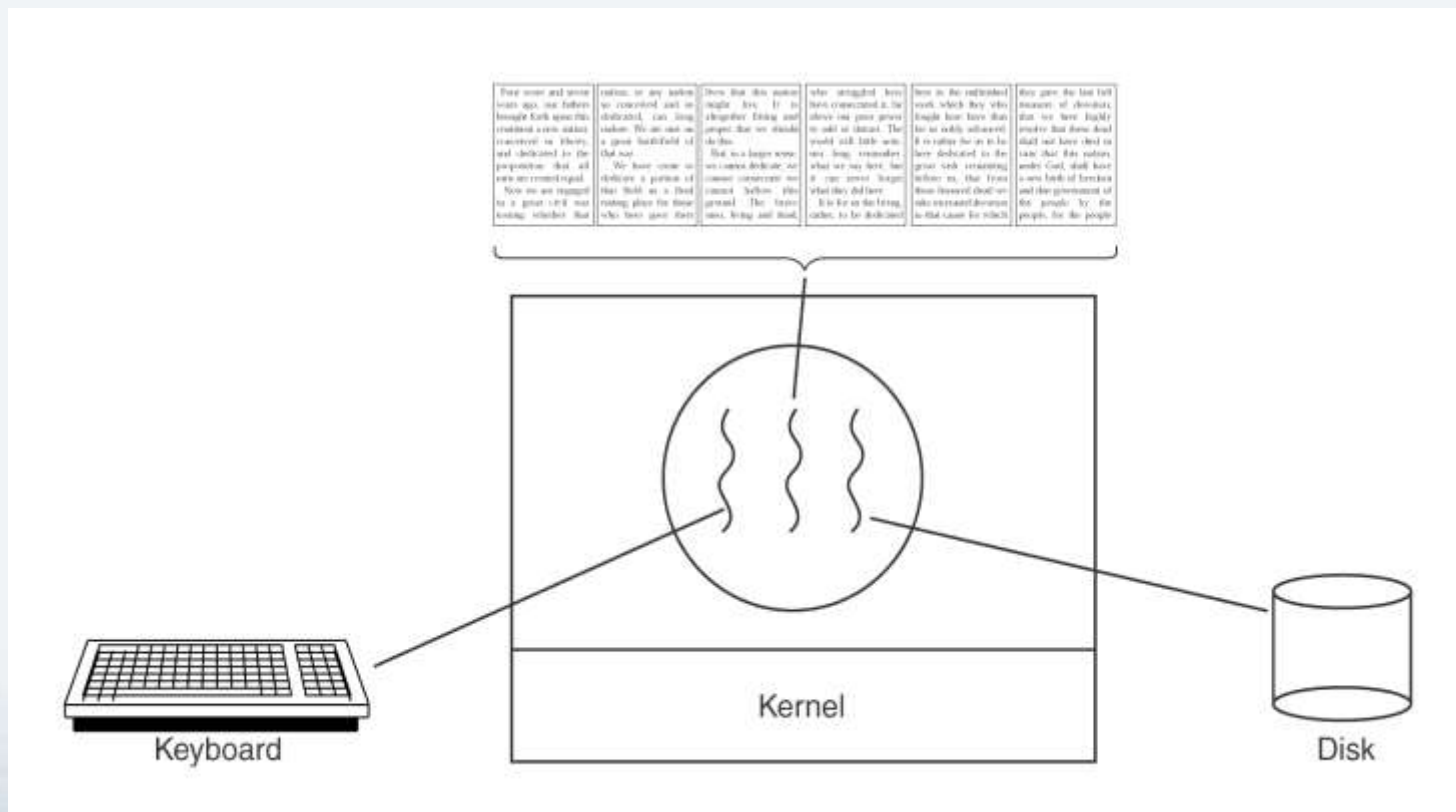
- 共享进程资源：它可与同属一个进程的其它线程共享进程所拥有的全部资源。
- 轻型实体：线程自己基本不拥有系统资源，只拥有少量必不可少的资源：程序计数器、一组寄存器、栈。
- 独立调度和分派的基本单位：在引入线程的OS中，线程是进程中的一个实体，是被系统独立调度和分派的基本单位。
- 可并发执行：同一进程中的多个线程之间可以并发执行，不同进程中的线程也能并发执行。

引入线程的好处

- 创建一个新线程花费时间少（结束亦如此）；
- 两个线程的切换花费时间少；
- 因为同一进程内的线程共享内存和文件，因此它们之间相互通信无须调用内核；
- 适合多处理机系统。

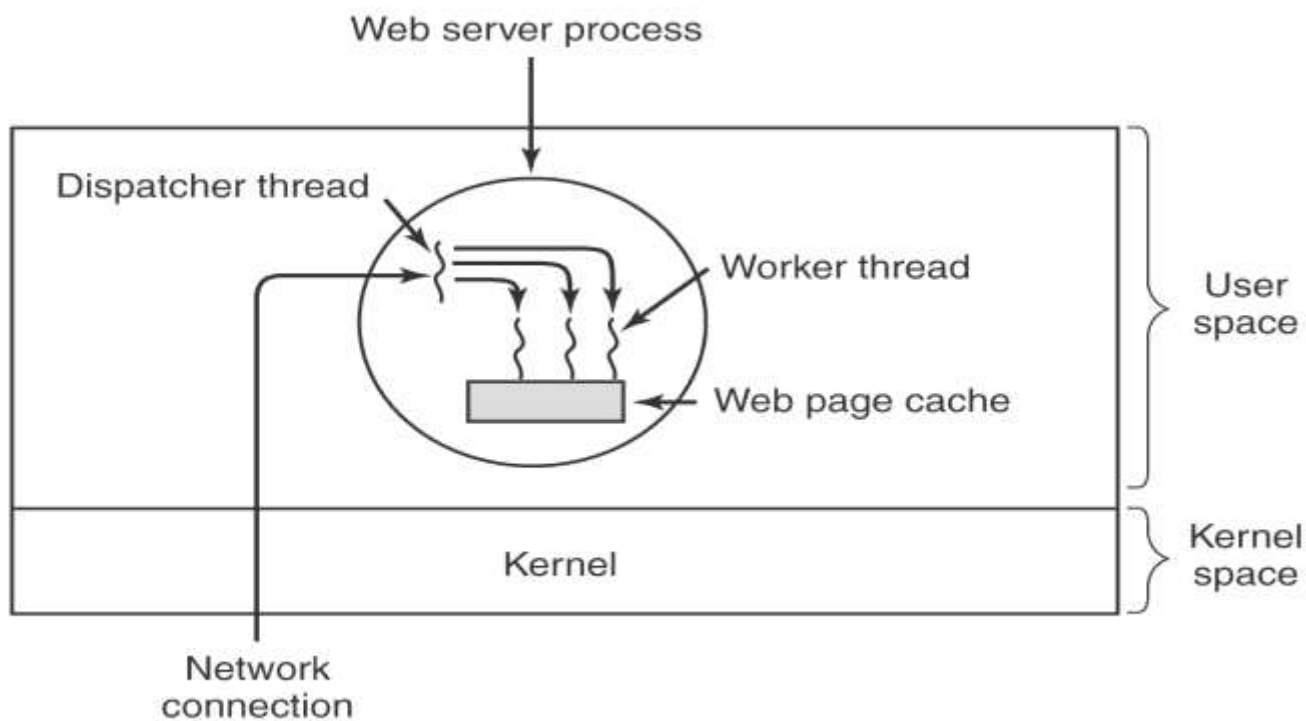
2.2.2 线程的使用

- 例1: 有三个线程的字处理程序



2.2.2 线程的使用

- 例2：一个多线程的Web服务器



2.2.2 线程的使用

```
while (TRUE) {  
    get_next_request(&buf);  
    handoff_work(&buf);  
}
```

(a)

分派线程

```
while (TRUE) {  
    wait_for_work(&buf)  
    look_for_page_in_cache(&buf, &page);  
    if (page_not_in_cache(&page))  
        read_page_from_disk(&buf, &page);  
    return_page(&page);  
}
```

(b)

工作线程

2.2.2 线程的使用

线程与进程的比较：

- ① 调度
- ② 并行性
- ③ 拥有资源
- ④ 系统开销

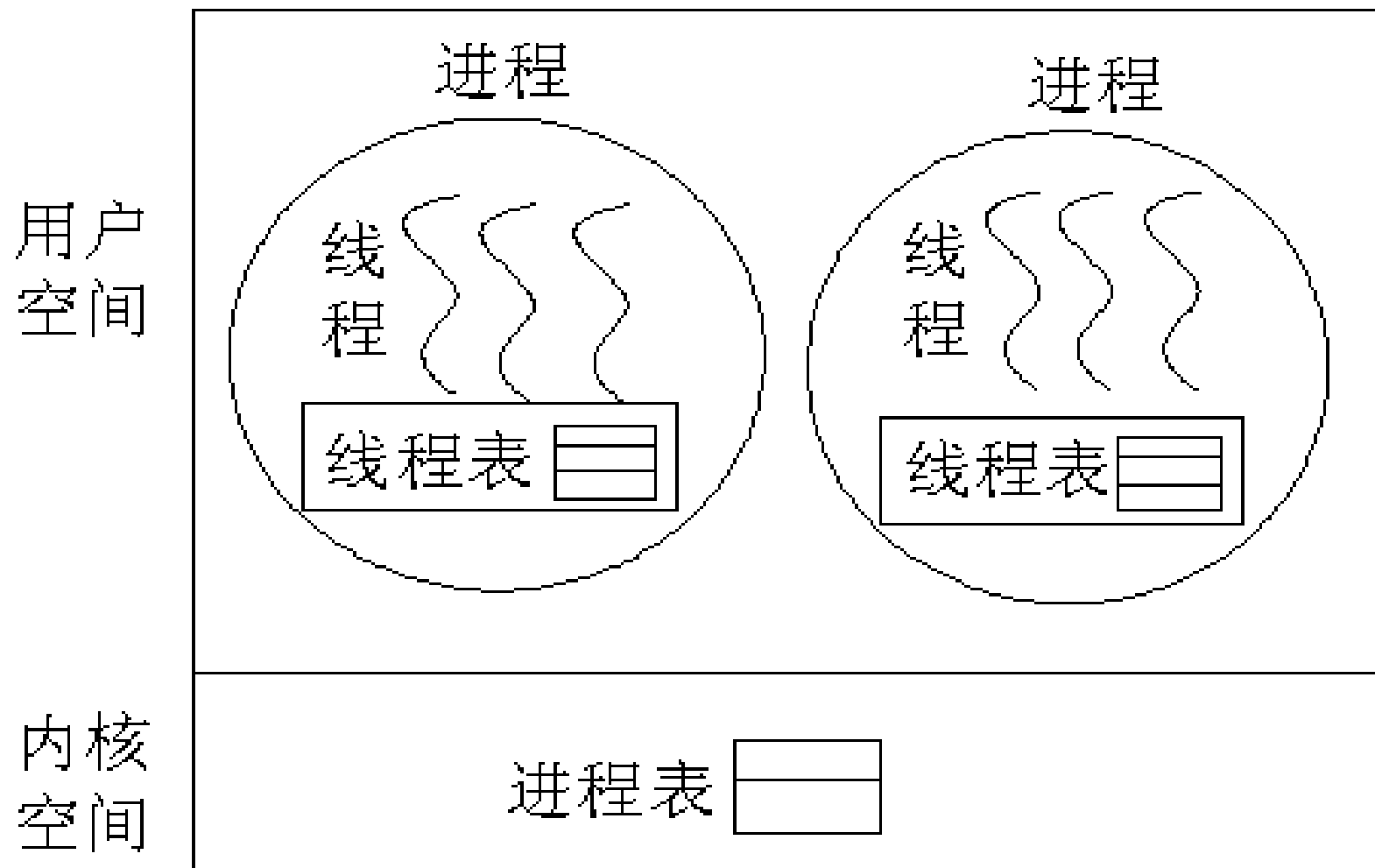
线程的实现机制

线程的三种实现机制：

- 用户级线程
- 核心级线程
- 两者结合方法

2.2.3 在用户空间中实现线程

- 由应用程序完成所有线程的管理 ---- 通过线程库
- 核心不知道线程的存在，因此调度仍以进程为单位进行
- 线程切换不需要核心态特权



用户级线程

ULT

线程库：

- 创建、撤消线程
- 在线程之间传递消息和数据
- 调度线程执行
- 保护和恢复线程上下文

ULT

对用户级线程的核心活动：

- 核心不知道线程的活动，但仍然管理线程的隶属进程的活动
- 当线程调用系统调用时，整个进程阻塞
- 但对线程库来说，线程状态是与进程状态独立的

ULT

优点:

- 线程切换不调用核心
- 调度是应用程序特定的: 可以选择最好的算法
- ULT可运行在任何操作系统上(只需要线程库)

ULT

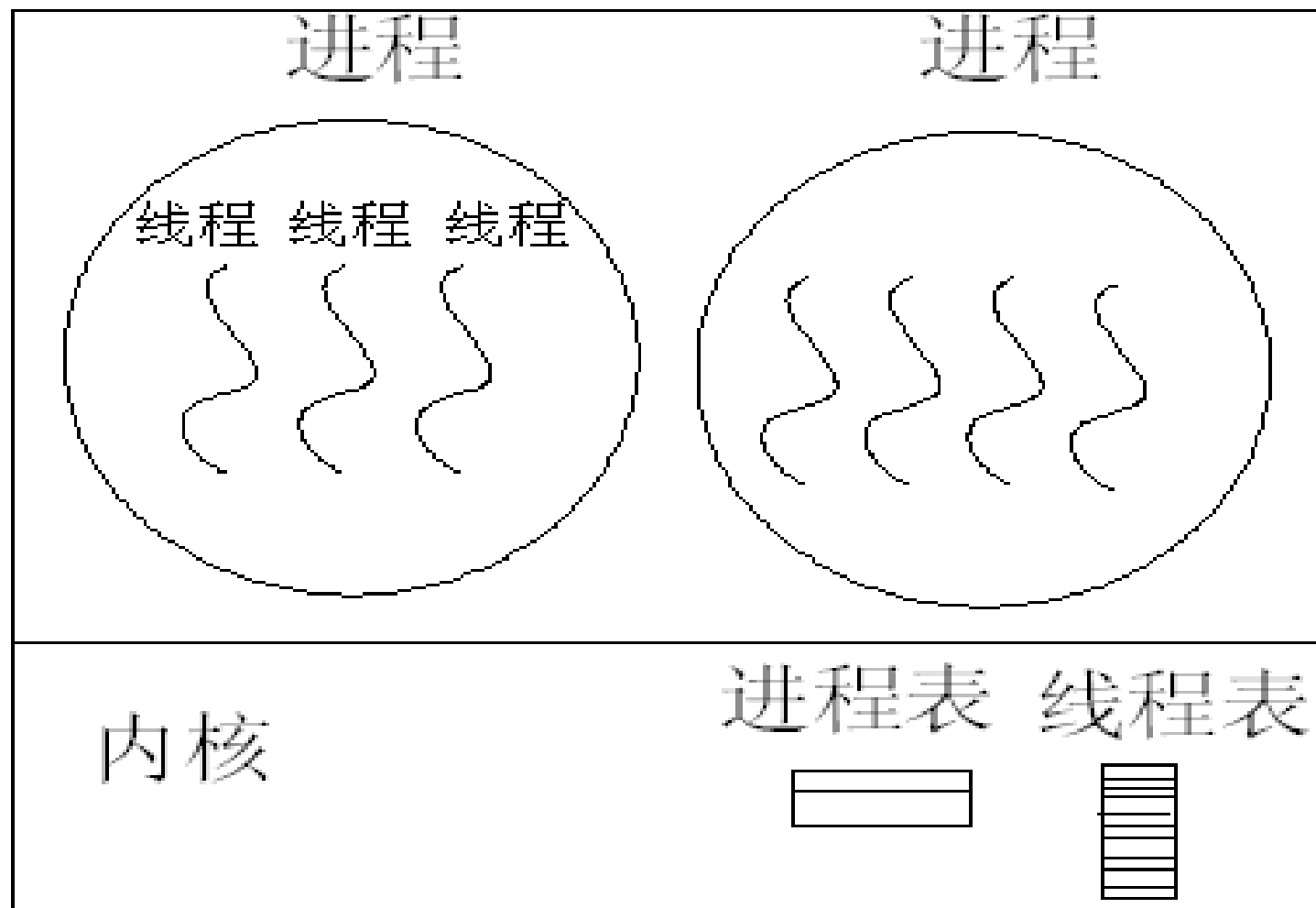
缺点：

- 大多数系统调用是阻塞的，核心阻塞进程，故进程中所有线程将被阻塞。
- 核心只将处理器分配给进程，同一进程中的两个线程不能同时运行于两个处理器上。

2.2.4 在内核中实现线程

核心级线程（KLT）：

- 所有线程管理由核心完成：利用系统调用
- 没有线程库，但对核心线程工具提供API
- 核心维护进程和线程的上下文
- 线程之间的切换需要核心支持
- 以线程为基础进行调度
- 例子：Windows NT



核心级线程

KLT

优点：

- 对多处理器，核心可以同时调度同一进程的多个线程
- 阻塞是在线程一级完成
- 核心例程是多线程的

缺点：

- 在同一进程内的线程切换调用内核，导致速度下降

ULT VS KLT

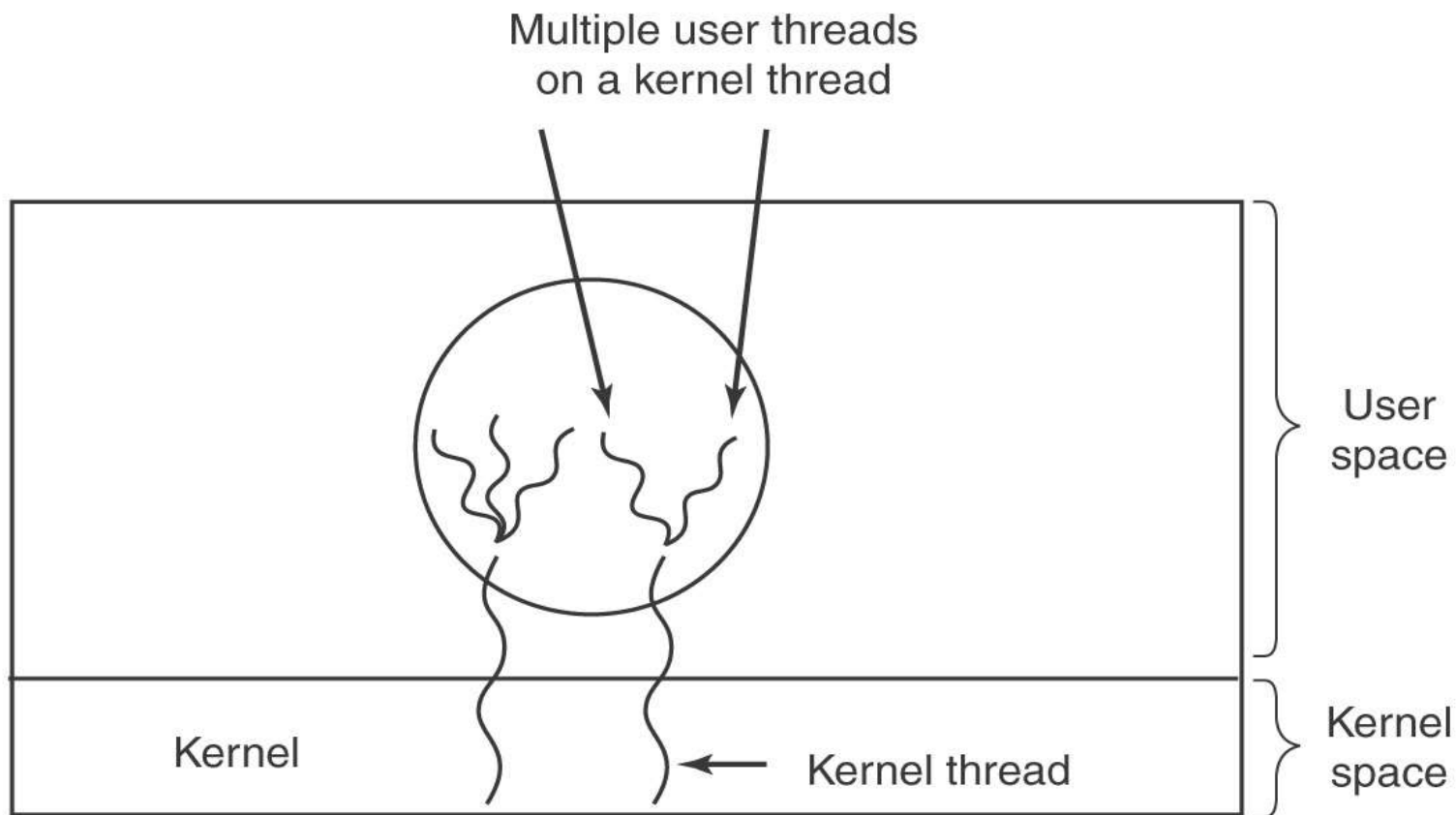
- 线程的调度和切换速度
- 系统调用的阻塞
- 线程执行时间

2.2.5 混合方式

ULT和KLT结合方法：

- 线程创建在用户空间完成
- 大量线程调度和同步在用户空间完成
- 程序员可以调整KLT的数量

2.2.5 混合方式



2.2.5 混合方式

- 内核只识别内核级线程，并对其进行调度。
- 其中一些内核级线程会被多个用户级线程多路复用。
- 如同在没有多线程能力操作系统中某个进程中的用户级线程一样，可以创建、撤销和调度这些用户级线程。
- 在这种模型中，每个内核级线程有一个可以轮流使用的用户级线程集合。

小结-----引入线程的原因

- ① 应用需求：在许多应用中同时发生着多种活动。其中某些活动随着时间的推移会被阻塞。通过将这些应用程序分解成可以并行运行的多个顺序线程，程序设计模型会变得更简单。

小结----引入线程的原因

- ② 线程比进程更轻量级，所以它们比进程更容易（即更快）创建，也更容易撤销。在许多系统中，创建一个线程较创建一个进程要快10~100倍。在有大量线程需要动态和快速修改时，具有这一特性是很有用的。

小结----引入线程的原因

- ③ 性能方面。若多个线程都是CPU密集型的，那么并不能获得性能上的增强，但是如果存在着大量的计算和大量的I/O处理，拥有多个线程允许这些活动彼此重叠进行，从而会加快应用程序执行的速度。

小结-----引入线程的原因

- ④ 在多CPU系统中，多线程是有益的，在这样的系统中，真正的并行有了实现的可能。

2.3 进程间通信



2.3 进程间通信

问题的提出

- 在飞机订票系统中的两个进程为不同的客户试图争夺飞机上的最后一个座位，如何保证正确性？
- 如果进程A产生数据而进程B打印数据，那么B在打印之前必须等待，直到A已经产生一些数据，如何保证执行的顺序？

2.3 进程间通信

- 并发执行的合作进程，由于它们都是合力完成一个共同的作业，所以必须保持一定的联系，以便协调地完成任务。这种联系就是指在进程间交换一定数量的信息。

2.3 进程间通信

2.3.1 同步与互斥

2.3.2 忙等待的互斥

2.3.3 休眠与唤醒

2.3.4 信号量

2.3.5 管程

2.3.6 消息传递

2.3.1 同步与互斥

两种形式的制约关系

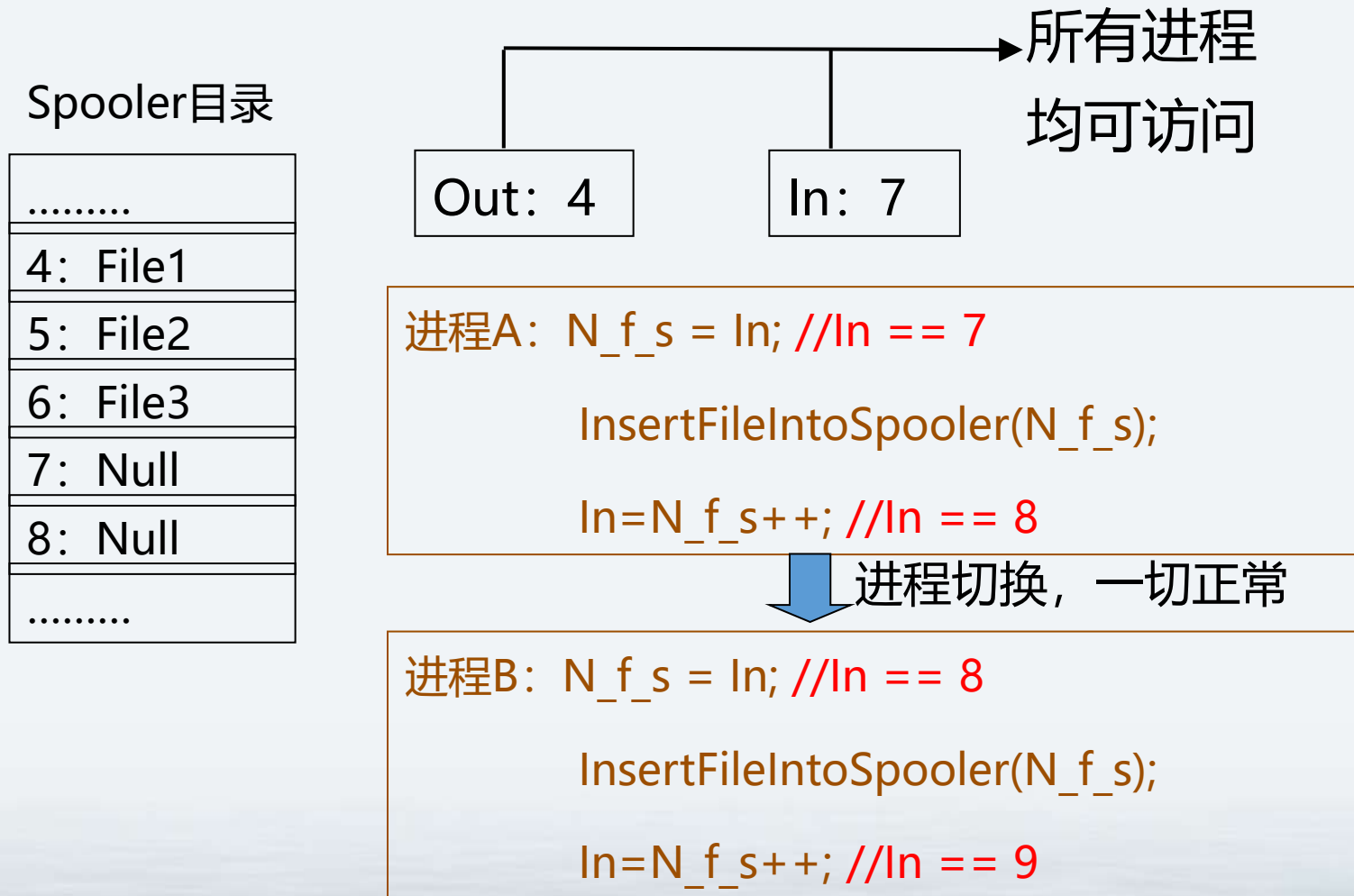
- 间接相互制约关系：进程 - 资源 - 进程

互斥关系，资源共享关系

- 直接相互制约关系：进程 - 进程

同步关系，相互合作关系

1. 互斥问题----- Spooler目录问题



Spooler 目录问题（互斥）

Out: 4

In: 7

Spooler目录

| |
|----------|
| |
| 4: File1 |
| 5: File2 |
| 6: File3 |
| 7: Null |
| 8: Null |
| |

进程A: $N_f_s = In; //In == 7$

↓ 进程切换

进程B: $N_f_s = In; //In == 7$
 $InsertFileIntoSpooler(N_f_s);$
 $In = N_f_s++; //In == 8$

↓ 进程切换, 进程B数据丢失

进程A:
 $InsertFileIntoSpooler(N_f_s);$
 $In = N_f_s++; //In == 8$

概念引入

- 竞争条件

- 两个或者多个进程读写共享数据，而运行结果取决于进程运行时的精确时序，则这种情况称之为竞争条件。
- 当竞争条件存在时，进程处理结果可能失效甚至发生错误。

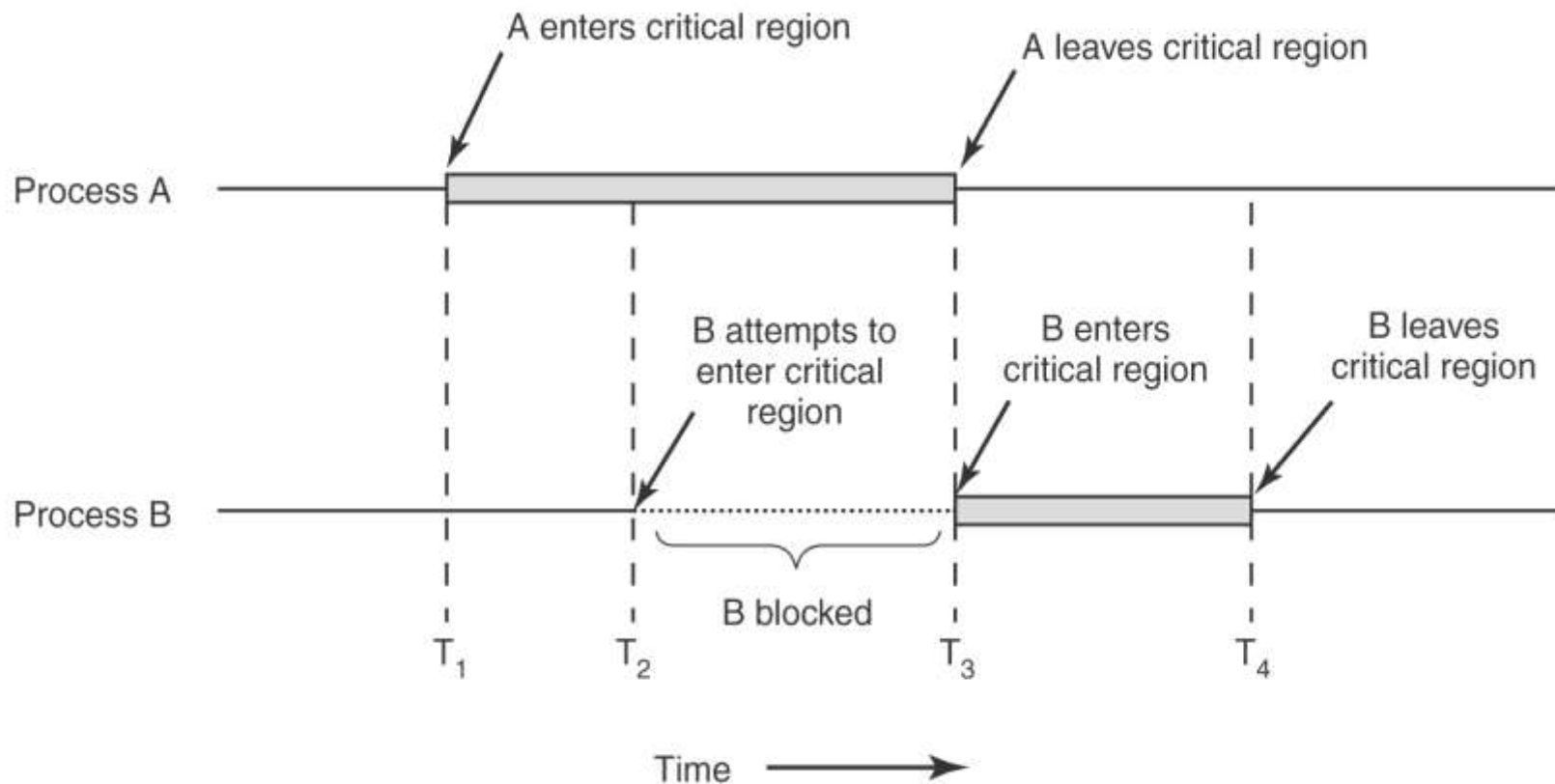
临界资源（Critical Resource）

- 一次仅允许一个进程访问的资源。如：进程 A、B 共享一台打印机，若让它们交替使用则得到的结果肯定不是我们希望的。
- 临界资源可能是硬件，如打印机；也可能是软件：变量，数据，表格，队列等。
- 并发进程对临界资源的访问必须作某种限制，否则就可能出现错误。

互斥问题

- 临界区：每个进程中，访问临界资源的那段代码称为临界区。
- 互斥：防止多个进程同时进入临界区

使用临界区的互斥



进程通信问题的解决原则

- 互斥原则：任何两个进程不能同时处于临界区
- 通用性原则：不对CPU的速度和数目进行任何假设
- 有效性原则：临界区外的进程不应阻塞其他进程
- 合理性原则
 - 不得使进程在临界区外无限制的等待
 - 当进程无法进入临界区时，应放弃CPU资源

2. 同步问题---“司机 - 售票员”问题

司机进程

While(True)

```
{  
    启动公车;  
    驾驶公车;  
    停止公车;  
}
```

售票员进程

While(True)

```
{  
    关车门;  
    卖车票;  
    开车门;  
}
```

正确运行过程

While(True)

```
{  
    (司机) 启动公车;  
    (售票员) 关车门;  
    (司机) 驾驶公车;  
    (售票员) 卖车票  
    (司机) 停止公车;  
    (售票员) 开车门;  
}
```

问题分析

两个独立进程如何保持同步?

现实应用中存在大量的类似实例

概念的引入（同步）

- 进程同步(Synchronization of multiple processes)是指对多个相关进程在执行次序上进行协调。
- 进程同步的主要任务是使并发执行的各进程之间能有效的共享资源和相互合作，从而使程序的执行具有可再现性。

概念的引入（同步）

- 同步关系分析

- 两个或者多个进程在运行顺序上存在固定的规则
- 由于分时操作系统的不确定性，导致同步关系无法保持
- 必须进行显式的程序控制，保证同步关系的正确

同步与互斥的差别

- 互斥：体现为排他性，可表现为“0 - 1”关系
- 同步：体现为时序性，比互斥更加复杂和多变

2.3.2 忙等待的互斥

- 基本思想：
 - 设定一个变量，标识临界区的状态
 - 互斥进程均检查这个变量，只有当其满足条件时方可进入临界区

2.3.2 忙等待的互斥

- 访问临界资源的循环进程描述如下：

REPEAT

ENTRY SECTION

CRITICAL SECTION

EXIT SECTION

REMAINDER SECTION

UNTIL FALSE

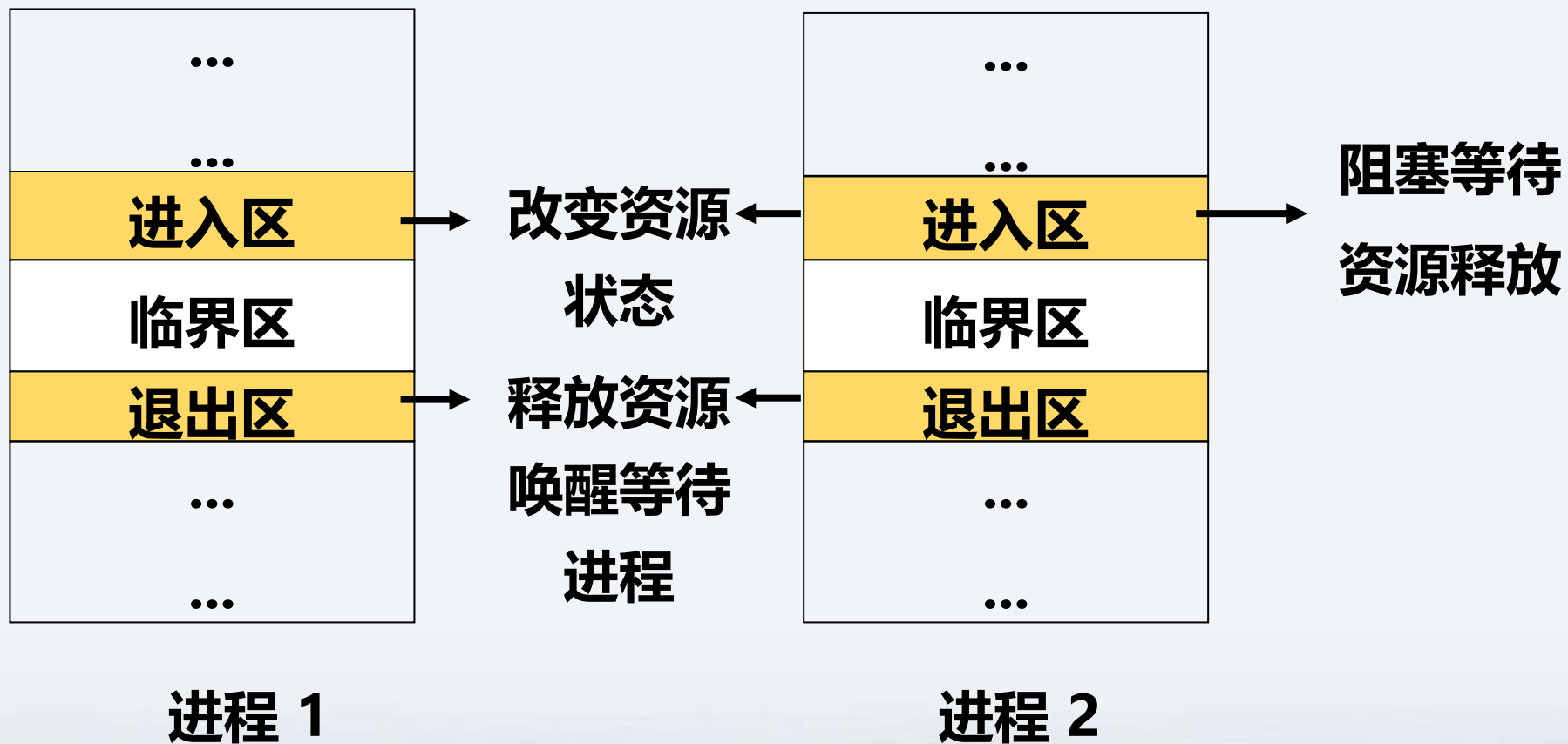
进入区

临界区

退出区

剩余区

2.3.2 忙等待的互斥



模型方法

软件方法

- 锁变量
- 严格轮转
- Peterson解决方案

硬件方法

- 关中断
- TSL

软件方法-----锁变量

lock = 0

进程A

While(lock != 0);

lock = 1;

Critical_region();

lock = 0;

进程B

While(lock != 0);

lock = 1;

Critical_region();

lock = 0;

锁变量的缺陷示例

lock = 0

进程A
While(lock != 0);

进程B
While(lock != 0);

进程A
lock = 1;
Critical_Region();

进程B
lock = 1;
Critical_Region();

发生进程调度，互斥的进程开始运行

发生进程调度，A并不知道B也进入临界区

发生进程调度，B并不知道A也进入临界区

锁 变 量

- 解决思想：设定监控变量（lock），通过其值变化控制进程操作
- 方法分析
 - 依然存在竞争条件，不能根本解决互斥问题
 - 致命缺陷：不具有操作的原子性

软件方法-----严格轮转法

turn = 0

进程A

```
While(TRUE)
{
    While(turn != 0);
    Critical_region();
    turn = 1;
    Nocritical_region();
}
```

进程B

```
While(TRUE)
{
    While(turn != 1);
    Critical_region();
    turn = 0;
    Nocritical_region();
}
```

严格轮转法的缺陷示例

turn = 0

进程A
进入 - 退出临界区
Nocritical_region();

进程B
进入 - 退出临界区
完成非临界区操作
等待进入临界区

第一次运行时，可
严格保证互斥关系

进程B需要第二次进入
临界区时，必须等待进
程A将turn值改为1。
但是进程A的非临界区
操作很慢

两进程必须交替进入临界区!!!

在一个进程比另一个慢很多的情况下，
轮流进入临界区并不是一个好方法

严格轮转法

- 方法分析
 - 忙等待浪费CPU时间
 - 存在边界情况，违反解决原则第三条：
进程可能被一个临界区之外的进程阻塞

软件方法----- Dekker 算法

```
Pturn = FALSE;  
Qturn = FALSE;  
enum turn;  
初值为1或2
```

进程P

```
While(TRUE)  
{  
  Pturn = TRUE;  
  While(Qturn) {  
    if(turn == 2){  
      Pturn = FALSE;  
      While(turn == 2);  
      Pturn = TRUE;}}  
  Critical_region();  
  turn = 2;  
  Pturn = FALSE;}
```

进程Q

```
While(TRUE)  
{  
  Qturn = TRUE;  
  While(Pturn){  
    if(turn == 1){  
      Qturn = FALSE;  
      While(turn == 1);  
      Qturn = TRUE;}}  
  Critical_region();  
  turn = 1;  
  Qturn = FALSE;}
```

我想进入临界区
如果对方也想，则
如果P坚持，则
我暂时让步
等待对方退出
然后我再表明意愿

现在允许P进入

Dekker 算法

- 基本思想：
 - 进程P(或Q)进入自己的临界区时，把自己的标志位Pturn(或Qturn)置为true，并检查对方标志位
 - 如果对方不在也不想进入临界区，进程P可立即进入临界区；如果双方都想进入，咨询指示器turn，若turn为1(或为2)，P(或Q)知道应该自己进入

Dekker 算法

- 若不使用turn?

进程P

...

Pturn = TRUE;

While(Qturn);

临界区;

Pturn = FALSE;

...

1 a

2 b

3 c

4 d

进程Q

...

Qturn = TRUE;

While(Pturn);

临界区;

Qturn = FALSE;

...

1

a

2

b

软件方法----- Peterson 方法

```
#define FALSE 0
#define TRUE 1
#define N      2
int turn
int interested[N]
```

```
void enter_region(int iProcessID)
{
    int other;
    other = 1 - iProcessID;
    interested[iProcessID] = TRUE;
    turn = iProcessID;
    while((turn == iProcessID)
        &&(interested[other] = TRUE);
}

void leave_region(int iProcessID)
{
    interested[iProcessID] = FALSE;
}
```

若为真，则表明此时已有进程在等待进入（或已进入）临界区，则当前进程等待

方法分析

克服严格轮转法的缺陷
有何不足？

Peterson 方 法

- 基本思想：
 - 用对turn的置值和while语句来限制每次只有一个进程进入临界区；
 - 进程执行完临界区程序后，修改interested[i]状态使等待进入临界区的进程可在有限时间内进入。

Peterson方法的使用

进程P

While(TRUE)

{

enter_region(PID);

Critical_region;

leave_region(PID);

}

进程Q

While(TRUE)

{

enter_region(QID);

Critical_region;

leave_region(QID);

}

硬件方法-----关中断

进程A

Close_INT;

Critical_region();

Open_INT;

进程B

Close_INT;

Critical_region();

Open_INT;

关 中 断

- 解决思想
 - 在临界区中防止发生进程调度
 - 保证临界区操作的完整性
- 方法分析
 - 用户控制系统中断是非常危险的
 - 本方法对多个CPU系统将失去作用

硬件方法----- TSL

- 锁变量法的问题：管理临界区的标志时要用到两条指令，而这两条指令在执行过程中有可能被中断。
- 测试和上锁这两个动作不能分开，以防两个或多个进程同时测试到允许进入临界区的状态。

测试并加锁：硬件TSL指令

TSL RX, LOCK

- 将内存单元LOCK的值送入寄存器RX中，并将LOCK的值置为非0；
- 执行TSL时，CPU将锁住内存总线，以禁止其它CPU在本指令结束之前访问内存。

测试并加锁：硬件TSL指令

TSL RX, LOCK

硬件实现锁变量机制

保证LOCK读写原子性

enter_region:

TSL REGISTER, LOCK

CMP REGISTER, #0

JNE enter_region

RET

leave_region:

MOVE LOCK, #0

RET

方法分析

硬件实现提高速度，适用于多处理机情况

依然存在忙等待的问题

测试并加锁：硬件TSL指令

- 锁住存储总线不同于屏蔽中断。屏蔽中断，然后在读内存字之后跟着写操作并不能阻止总线上的第二个处理器在读操作和写操作之间访问该内存字。事实上，在处理器1上屏蔽中断对处理器2根本没有任何影响。让处理器2远离内存直到处理器1完成的惟一方法就是锁住总线，这需要一个特殊的硬件设施。

忙等待模型汇总分析

- 优点分析
 - 实现机制简单易懂，可有效保证互斥
- 缺点分析
 - 只适用于两个进程间互斥，不具有通用性
 - 忙等待严重浪费CPU资源，降低硬件效率

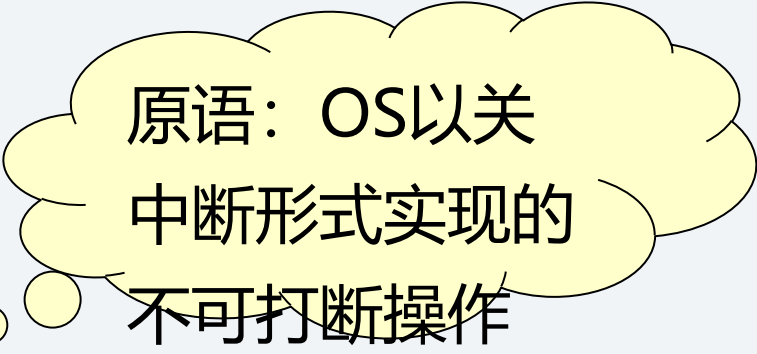
忙等待模型汇总分析

- 缺点分析

- “优先级调度” + “忙等待” = 优先级反转
 - 进程H和L，H优先级比L优先级高
 - L先进入临界区，H后进入临界区
 - H开始忙等待，而L由于无法获得CPU也不能离开临界区

2.3.3 休眠与唤醒

- 模型思想



原语：OS以关
中断形式实现的
不可打断操作

- OS提供系统调用原语 (Atomic Action) , 改变进程状态
- 无法进入临界区的进程转为阻塞态, 条件满足则被唤醒
- 可有效克服忙等待造成的资源浪费
- 更重要的优点: 可同时实现同步与互斥

2.3.3 休眠与唤醒

模型方法

- 简单的睡眠 - 唤醒方法
- 信号量机制
- 管程方法

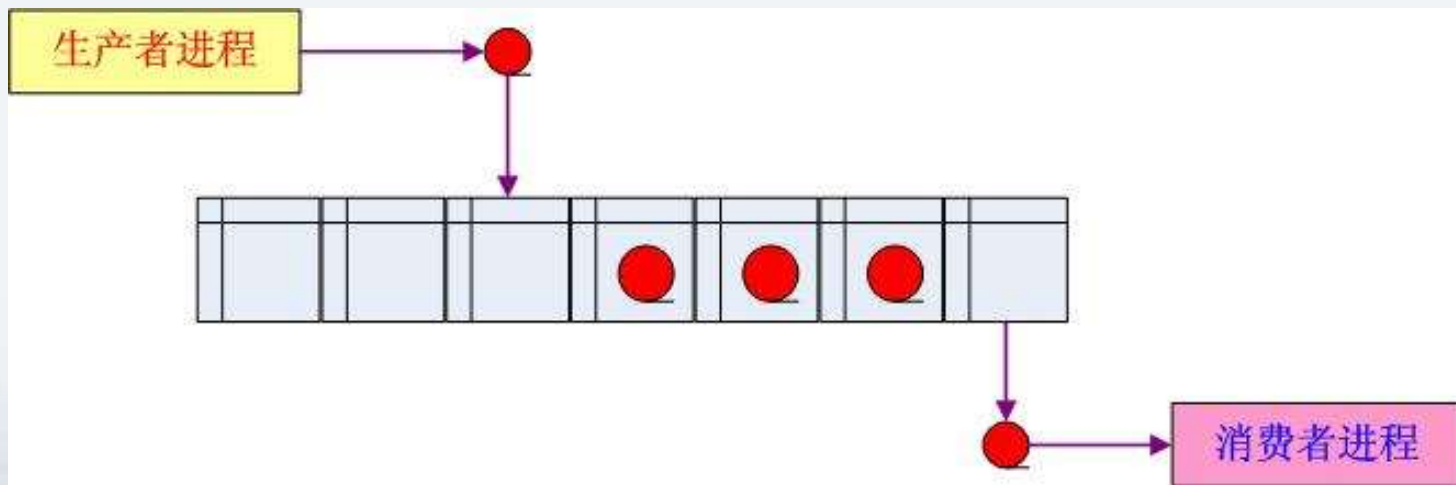
简单的睡眠 - 唤醒方法

- 操作系统原语设计
 - Sleep(): 调用该原语的进程将变为阻塞态
 - Wakeup(ID): 该原语将唤醒ID标识的进程
- 原语使用思想
 - 进入临界区前检查竞争条件, 如不满足则睡眠
 - 离开临界区后唤醒互斥进程

经典问题：生产者 - 消费者问题

问题描述

- 一个有限空间的共享缓冲区，负责存放货物
- 生产者向缓冲区中放物品，缓冲区满则不能放
- 消费者从缓冲区中拿物品，缓冲区空则不能拿



生产者—消费者问题的解法

```
# define N 100  
int count = 0
```

并不用
while循环
进行忙等

Producer进程

```
While(TRUE)  
{  
    Produce-Item();  
    if(count == N)  
        sleep();  
    Enter-item();  
    count = count + 1;  
    if(count == 1)  
        wakeup(consumer);  
}
```

Comsumer进程

```
While(TRUE)  
{  
    if(count == 0)  
        sleep();  
    Remove-Item();  
    count = count - 1;  
    if(count == N-1)  
        wakeup(producer);  
    Consume-item();  
}
```

潜在的竞争条件

```
# define N 100  
int count = 0
```

两个进程都
将永远睡眠

Comsumer进程

... ..

if(count == 0)

发生进程调度，导致C
进程并未进行Sleep

Producer进程

... ..

wakeup(consumer);

P进程认为C在睡眠，
试图唤醒C

Comsumer进程

sleep();

由于错过了Wakeup信
号，C进入了睡眠状态

Producer进程

... ..

P进程一直运行下去，填
满缓冲区后也睡眠了

Producer进程

sleep();

普通的睡眠-唤醒解决方法

- 临界情况
 - count的访问未加限制，形成竞争条件
 - 必须增加唤醒等待位，以解决互斥问题
- 方法分析
 - 较忙等待更进一步，有效节省CPU资源
 - 存在竞争条件，需要额外处理
 - 当互斥进程数增加时，方法有效性下降

现有各种处理方法分析

- 根本缺陷
 - 总是通过共享的变量来控制互斥关系
 - 停留在进程间协商层次，OS缺少宏观调控
- 改进办法
 - 由OS提供宏观调控管理机制
 - 彻底消除调度顺序引起的错乱
 - OS实现更高层次的原语级操作

2.3.4 信号量机制

- 核心思想
 - 引入新的数据结构定义：信号量
 - 利用信号量，提供更为复杂的原语级操作
 - 从根本上解决“潜在竞争条件”问题
 - 可以方便的推广到一般情况，适用于多进程的互斥与同步

2.3.4 信号量机制

- 信号量机制是荷兰学者Dijkstra在1965年提出的一种卓有成效的同步工具，其基本思想是在多个相互合作的进程之间使用简单的信号来同步。
- 一个信号量通常对应于一类临界资源，在使用前，信号量必须经过定义并赋适当的初值，初值表示系统中某类资源的数目。

2.3.4 信号量机制

```
struct semaphore
```

```
{    int value;————→ •资源使用情况  
    PCB *queue;————→ •信号量等待队列  
};
```

value:

>0: 表示系统中该类资源当前可用的数目,

=0: 表示资源已被占用, 无其它进程等待

<0: 表示该类资源已分配完毕, 其绝对值表示系统中因请求该类资源而阻塞在queue队列上等待的进程数目

2.3.4 信号量机制

除初始化外，仅能通过down()、up()操作(P、V操作)来访问信号量。

- P(s)：将信号量s减去1，若结果小于0，则调用P(s)的进程被置成等待信号量s的状态。
- V(s)：将信号量s加1，若结果不大于0，则释放一个等待信号量s的进程。

该进程状态置为阻塞状态；放弃处理机；将该进程的PCB插入链表S->queue中。

down(semaphore *s)

{
 •
 •

S->value = S->value - 1;

if S->value < 0 then block(S->queue)

P(S)

}

当 S.value < 0 时，进程会立即将自己阻塞，因此解决了“忙等”问题，做到了“让权等待”。

up(semaphore *s)

{

S->value = S->value+1;

if S->value ≤ 0 then wakeup(S->queue)

}

V(S)

唤醒S->queue链表中的
第一个等待进程；改变状
态为就绪态；将其插入就
绪队列中。

2.3.4 信号量机制

- 每执行一次down()操作就意味着请求分配一个单位的该类资源给执行down() 操作的进程使用
- 而执行一次up()操作就意味着进程释放出一个单位的该类可用资源

2.3.4 信号量机制

- 为确保信号量能正确工作，最重要的是要采用一种不可分割的方式来实现它。
- 通常是将up和down作为系统调用实现，而且操作系统只需在执行以下操作时暂时屏蔽全部中断：测试信号量、更新信号量以及在需要时使某个进程睡眠。由于这些动作只需要几条指令，所以屏蔽中断不会带来什么副作用。

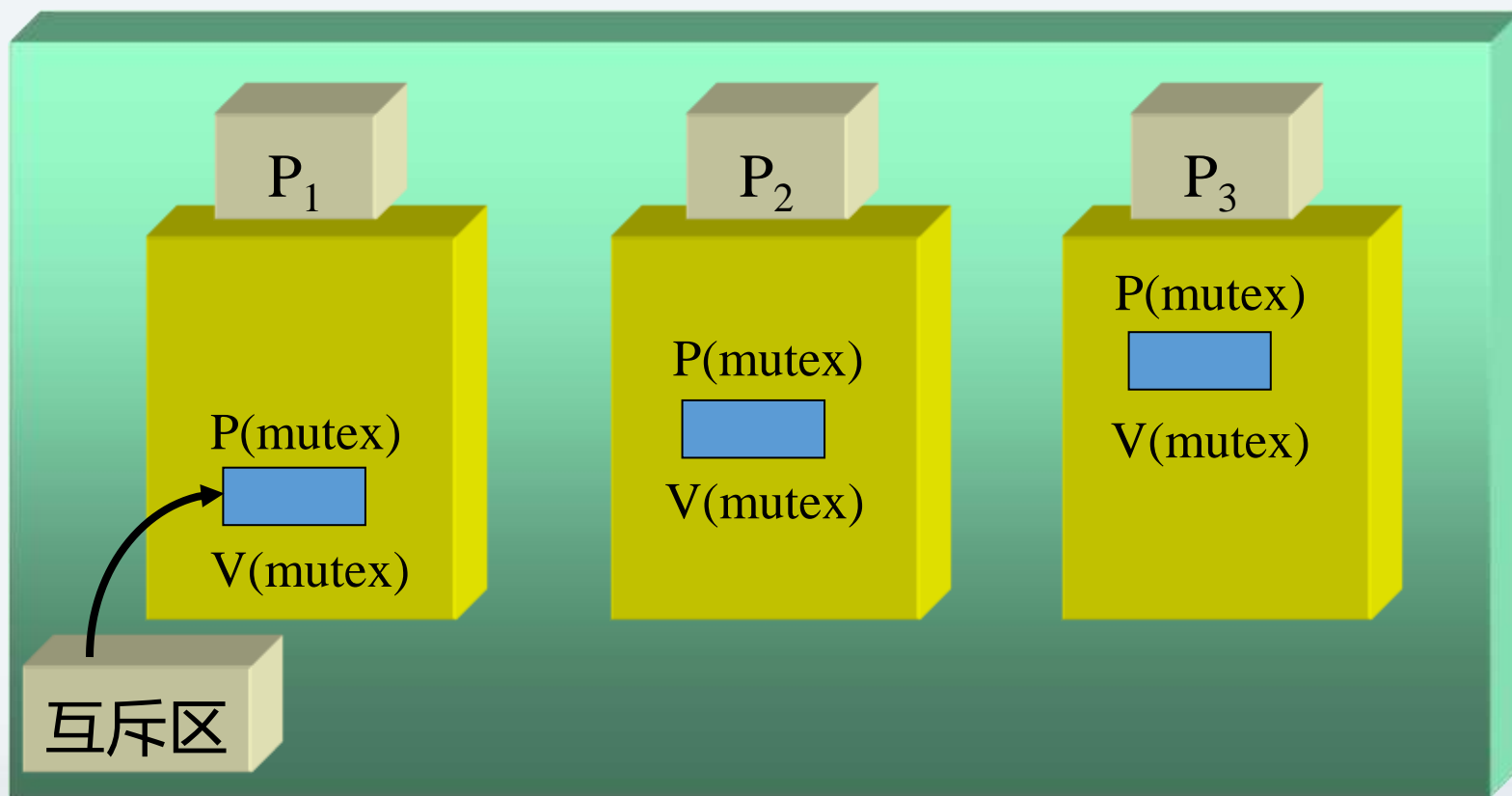
2.3.4 信号量机制

- 如果使用多个CPU，则每个信号量应由一个锁变量进行保护。通过TSL或XCHG指令来确保同一时刻只有一个CPU在对信号量进行操作。

用信号量实现进程互斥

- 为使多个进程能互斥地访问某临界资源，只需为该资源设置一个互斥信号量mutex，并设其初值为1，然后将各进程的临界区CS置于P(mutex)和V(mutex)操作之间即可。

用信号量实现进程互斥



最基本互斥机制的实现

```
semaphore mutex ;  
mutex.value= 1
```

```
进程A  
while(TRUE)  
{  
    nocritical_region();  
    Down(&mutex);  
    critical_region();  
    Up(&mutex);  
    nocritical_region();  
}
```

```
进程B  
while(TRUE)  
{  
    nocritical_region();  
    Down(&mutex);  
    critical_region();  
    Up(&mutex);  
    nocritical_region();  
}
```

用信号量实现进程互斥

模拟执行：

| 序号 | 进程A | 进程B | mutex | 说明 |
|----|----------|----------|-------|-------------|
| 0 | | | 1 | |
| 1 | P(mutex) | | 0 | A进入，占资源 |
| 2 | | P(mutex) | -1 | B想进入，无资源，阻塞 |
| 3 | V(mutex) | | 0 | A释放资源，PB解封 |
| 4 | | V(mutex) | 1 | B释放资源 |
| 0 | | | | 可反复执行 |

用信号量实现进程互斥

- 对于两个并发进程，互斥信号量的值仅取1、0 和 -1三个值：若 $\text{mutex} = 1$ 表示没有进程进入临界区；若 $\text{mutex} = 0$ 表示有一个进程进入临界区；若 $\text{mutex} = -1$ 表示一个进程进入临界区，另一个进程等待进入。

用信号量实现进程互斥

- 当实现 n 个进程并发时，互斥信号量的取值为1、0、-1、...、 $-(n-1)$ 。
- P、V操作必须成对出现：遗漏P(mutex)将不能保证互斥访问，遗漏V(mutex)将不能在使用临界资源之后将其释放（给其他等待的进程）。

用信号量实现同步

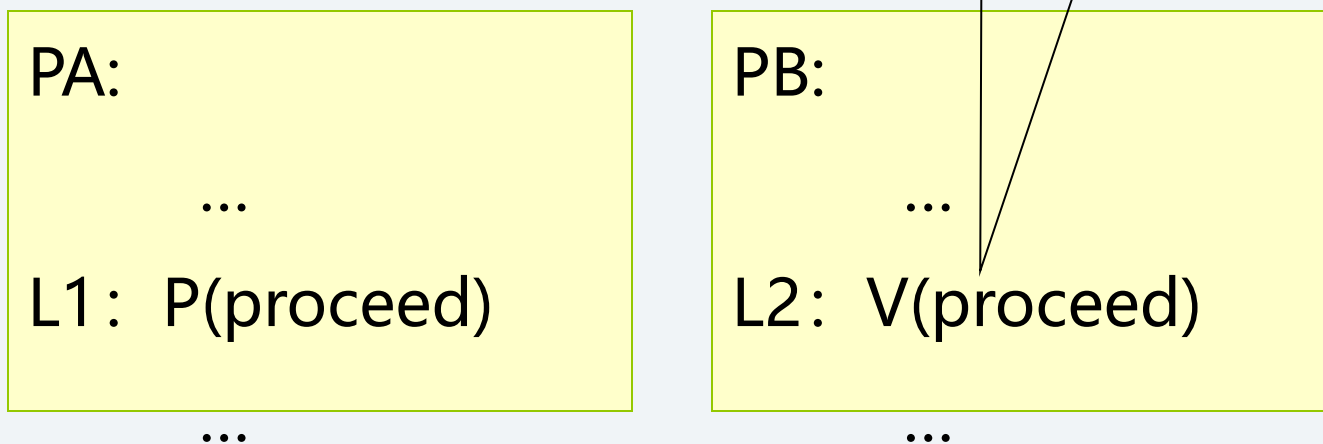
- 异步环境下的一组并发进程，因直接制约关系而互相发送消息而进行相互协作、相互等待，使得各进程按一定的速度执行的过程称为进程间的同步。

用信号量实现同步

- 进程同步的信号量与进程互斥的信号量在含义上是有着明显的不同，进程同步的信号量只与制约进程及被制约进程有关，而不是与整组并发进程有关。

用信号量实现同步

模型



- 信号量proceed用于实现同步。 PA在L1点必须与PB在L2点同步， PA受PB的制约，而 PB不受PA的制约，是非对称的。
- P、V操作必须成对出现， 当用于实现互斥时，它们同处于同一进程；当用于实现同步操作时，则不在同一进程中出现。

用信号量实现同步

- 【例】：生产者每次生产一件物品（数据）存入缓冲区，消费者每次从缓冲区取一件物品消费。假定缓冲区只能存放一件物品。

用信号量实现同步

- 我们可为生产者进程和消费者进程设置相应的信号量empty、full，empty表示生产者能否将物品存入缓冲区，full表示生产者告诉消费者能否从缓冲区中取物品。开始时，缓冲区是空的。显然，empty初值为1，full初值为0。

semaphore empty, full;
empty.value = 1 ; full.value = 0 ;

process producer

```
{  
    int data;  
    生产一件物品;  
    P(&empty);  
    将物品放入缓冲区;  
    V(&full);  
}
```

process consumer

```
{  
    int data;  
    P(&full);  
    从缓冲区中取物;  
    V(&empty);  
    消费物品;  
}
```

将问题一般化，现有 m 个生产者和 n 个消费者，它们共享可存放 k 件物品的缓冲区。

两个信号量够吗？

生产者 - 消费者问题分析

- 问题分析：这是一个同步与互斥共存的问题
- 互斥关系分析
 - 任何时刻，只能有一个进程在缓冲区中操作
 - 引入互斥信号量（二元信号量） mutex，其初值为1
 - 信号量为0，表明已有进程进入临界区；

生产者 - 消费者问题分析

- 同步关系分析

- 对于“生产者”而言，缓冲区满则应等待。引入同步信号量“empty”，表明空缓冲区的数目，初值为有界缓冲区的大小N，为0表示缓冲区满。
- 对于“消费者”而言，缓冲区空则应等待。引入同步信号量“full”，表明已用缓冲区的数目，为0表示缓冲区空

生产者 - 消费者信号量解法

```
# define N 100  
semaphore mutex,  
empty, full;  
mutex.value = 1;  
empty.value = N;  
full.value = 0;
```

Producer进程

```
int item;  
While(TRUE)  
{  
    Produce-Item(&item);  
    down(&empty);  
    down(&mutex);  
    Enter-item(item);  
    up(&mutex);  
    up(&full);  
}
```

Comsumer进程

```
int item;  
While(TRUE)  
{  
    down(&full);  
    down(&mutex);  
    Remove-item(&item);  
    up(&mutex);  
    up(&empty);  
    Consume-item(item);  
}
```

【 思考题 】

down操作的顺序能否颠倒？

【 问 题 分 析 】

- 用信号量机制解决进程同步问题需在程序中正确设置信号量和在适当位置安排P、V操作。
- 例如生产者-消费者问题中，用于互斥进入临界区的P(mutex)、V(mutex)语句需紧靠着临界区前和后，而生产者用于同步的P(empty)和V(full)(消费者为P(full)和V(empty))语句相对临界区就在外面一层。

【 问 题 分 析 】

- 如在以上问题中生产者的P(empty)和P(mutex)先后位置对调，相应消费者的P(full)和P(mutex)先后位置也对调，则生产者和消费者在并发执行时，就有可能会出现问題。

| 事件 | 操作 | | 操作引起（信号量）变化 | | | 操作引起（队列）变化 | | | |
|----------------|------------|------------|-------------|-------|------|------------|-----|-----|-----|
| | 生产者 (P) | 消费者 (C) | mutex | empty | full | RL | QmL | QeL | QfL |
| 0 | / | / | 1 | n | 0 | P、C | / | / | / |
| C ₁ | / | P(mutex) | 0 | n | 0 | P | / | / | / |
| C ₂ | / | P(full) | 0 | n | -1 | P | / | / | C |
| P ₁ | P(empty) | | 0 | n-1 | -1 | / | / | / | C |
| P ₂ | P(mutex) | | -1 | n-1 | -1 | / | P | / | C |

RL：就绪队列

QmL、QeL、QfL：分别等待mutex、empty、full
信号量而阻塞的队列

同步与互斥问题

- 互斥信号量
 - mutex: 防止多个进程同时进入临界区
- 同步信号量
 - empty和full: 保证事件发生的顺序
 - 缓冲区满时, Producer停止运行
 - 缓冲区空时, Consumer停止运行
- 概念差别——互斥与同步（并发的两个要素）
 - 互斥: 保护临界区, 防止多个进程同时进入
 - 同步: 保证进程运行的顺序合理

同步/互斥信号量的使用方法

- 互斥信号量：必定成对出现：进入临界区——临界区——退出临界区
- 同步信号量：未必成对出现，依赖于同步关系的性质
- 同步信号量和互斥信号量的操作顺序
 - 基本原则：互斥信号量永远紧邻临界区

信号量模型汇总分析

- 优点分析

- 彻底解决忙等待弊端，提高OS的管理层次
- 可实现复杂的同步与互斥情况，特别是多进程间通信
- 可最大限度的保证并发效率

- 缺点分析

- 实现机制复杂，互斥和同步关系分析困难
- 存在死锁陷阱，需要谨慎小心严密的设计

2.3.5 管程

为什么引入管程？

- 把分散在各进程中的临界区集中起来进行管理；
- 防止进程有意或无意的违法同步操作；
- 提高可读性，便于程序正确性验证。

2.3.5 管程

- 管程是一种集中式同步机制，它的基本思想是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。

2.3.5 管程

管程的基本概念：

- 一个管程定义了一个数据结构和能为并发进程所执行（在该数据结构上）的一组操作，这组操作能同步进程和改变管程中的数据。

2.3.5 管程

管程的四个组成部分：

- 名称
- 局部于管程的数据说明
- 对该数据结构进行操作的一组过程/函数
- 初始化语句

```
monitor example  
  integer i;  
  condition c;  
  
  procedure producer();  
  .  
  .  
  .  
  end;  
  
  procedure consumer();  
  .  
  .  
  .  
  end;  
end monitor;
```

局部于管程的
共享变量说明

对该数据结构
进行操作的一
组过程

2.3.5 管程

管程的特点：

- 管程内的局部变量只能被局部于管程内的过程所访问；反之亦然，即局部于管程内的过程只能访问管程内的变量。
- 任何进程只能通过调用管程提供的过程入口进入管程。
- 任一时刻，最多只能有一个进程在管程中执行。

2.3.5 管程

- 管程相当于围墙，它把共享变量和对它进行操作的若干过程围了起来，所有进程要访问临界资源时，都必须经过管程(相当于通过围墙的门)才能进入，而管程每次只准许一个进程进入管程，从而实现了进程互斥。

2.3.5 管程

- 管程是一种编程语言的构件，所以编译器知道它们很特殊，并可以采用与其他过程调用不同的方法来处理它们。对进入管程实现互斥由编译器负责，所以出错的可能性要小得多。

2.3.5 管程

管程提供了一种实现互斥的简便途径

NOT ENOUGH!!!

我们还需要一种方法使得进程在无法继续运行时被阻塞

2.3.5 管程

- 条件变量：

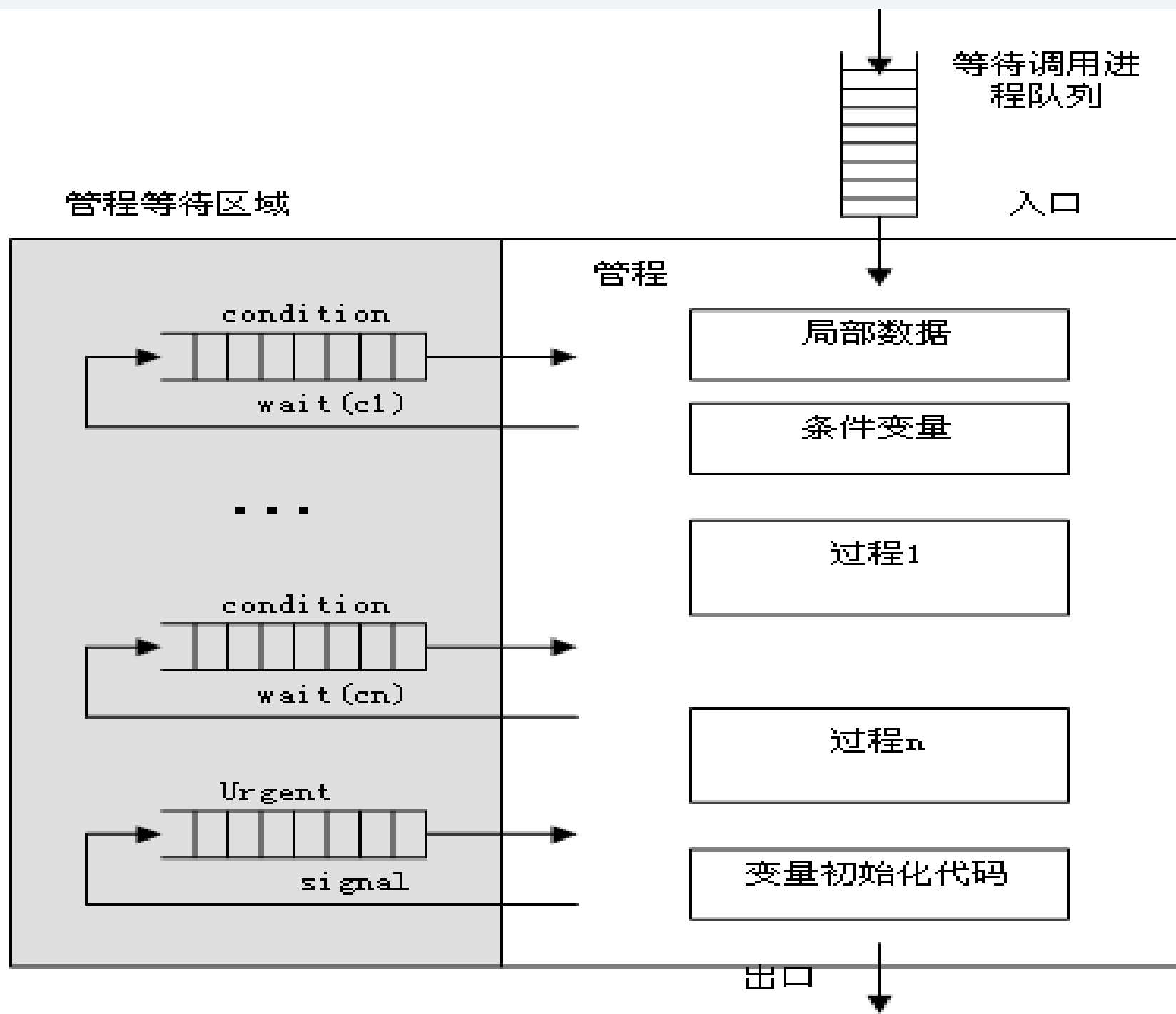
当进程通过管程请求临界资源未能满足时，将排在队列上等待。等待的原因可能有多，为了区分它们，引入条件变量 `condition`。每个 `condition` 表示一种等待原因。

2.3.5 管程

对条件变量进行操作的两个函数过程:

- WaitC(C): 将调用此函数的进程挂起并阻塞在与条件变量C相应的队列中, 同时使其它进程可以进入管程。
- SignalC(C): 恢复某个由于在条件变量C上执行WaitC操作而被挂起阻塞的进程执行。若没有被挂起的进程, 则执行空操作, 即什么也不做。

注意对比与up()的不同。



利用管程解决生产者-消费者问题

PC管程描述：

- 用管程实现生产者和消费者问题。仍然使用有K个缓冲区的环形缓冲区，每个缓冲区可容纳一个数据记录。
- 用full表示缓冲区已满的条件变量，用empty表示缓冲区已空的条件变量，用count表示当前缓冲区未取走的数据记录数。

利用管程解决生产者-消费者问题

Monitor PC

{

product buffer[K] ;

Condition full, empty ;

static int in = 0, out = 0 ,count = 0;

/*初始化*/

利用管程解决生产者-消费者问题

```
public put( product x)
{
    if (count == K ) WaitC(full); //缓冲区已满, 等待
    buffer[in] = x ;
    in = (in+1) % K;
    count ++;
    SignalC( empty );
}
```


利用管程解决生产者-消费者问题

```
public get( product *x)
{
    if (count == 0 ) WaitC(empty); //缓冲区已空, 等待
    *x = buffer[out] ;
    out = (out+1) % K;
    count --;
    SignalC( full );
}
```

利用管程解决生产者--消费者问题

- 生产者 and 消费者描述:

Producer

```
{  
    product x;  
    生产一件物品x;  
    PC.put(x); /*调用管程过程, 将物品放入缓冲区*/  
}
```

Consumer

```
{  
    product y;  
    PC.get(&y); /*调用管程过程, 从缓冲区中取物品*/  
    消费一件物品y;  
}
```

JAVA 中的 实现

- 利用关键字 `synchronized`
- JAVA保证一旦某个线程执行该方法，就不允许其他线程执行该类中的任何 `synchronized`方法。

```

public class ProducerConsumer {
    static final int N = 100;           // constant giving the buffer size
    static producer p = new producer(); // instantiate a new producer thread
    static consumer c = new consumer(); // instantiate a new consumer thread
    static our_monitor mon = new our_monitor(); // instantiate a new monitor
    public static void main(String args[]) {
        p.start();                      // start the producer thread
        c.start();                      // start the consumer thread
    }
    static class producer extends Thread {
        public void run() {              // run method contains the thread code
            int item;
            while (true) {               // producer loop
                item = produce_item();
                mon.insert(item);
            }
        }
        private int produce_item() { ... } // actually produce
    }
    static class consumer extends Thread {
        public void run() {              // run method contains the thread code
            int item;
            while (true) {               // consumer loop
                item = mon.remove();
                consume_item(item);
            }
        }
        private void consume_item(int item) { ... } // actually consume
    }
    static class our_monitor {           // this is a monitor
        private int buffer[] = new int[N];
        private int count = 0, lo = 0, hi = 0; // counters and indices
        public synchronized void insert(int val) {
            if (count == N) go_to_sleep(); // if the buffer is full, go to sleep
            buffer[hi] = val;              // insert an item into the buffer
            hi = (hi + 1) % N;             // slot to place next item in
            count = count + 1;             // one more item in the buffer now
            if (count == 1) notify();      // if consumer was sleeping, wake it up
        }
        public synchronized int remove() {
            int val;
            if (count == 0) go_to_sleep(); // if the buffer is empty, go to sleep
            val = buffer[lo];              // fetch an item from the buffer
            lo = (lo + 1) % N;             // slot to fetch next item from
            count = count - 1;             // one less item in the buffer
            if (count == N - 1) notify();  // if producer was sleeping, wake it up
            return val;
        }
        private void go_to_sleep() { try{wait();} catch(InterruptedException exc) {};}
    }
}

```

2.3.5 管程

- 核心思想
 - 实现一种包含过程、变量、数据结构的独立模块
 - 任何时刻，只能有一个进程在管程内
 - 由编译器提供支持，实现进程间的互斥
- 方法分析
 - 优点：实现了进程互斥的自动化
 - 缺点：依赖于编译器，无法通用或普及

小结

- 管程和信号量，都被设计用来解决访问一块公共存储器的一个或多个CPU上的互斥问题。
- 但是：信号量太低级，而管程在少数几种编程语言以外无法使用。同时，这些原语均未提供机器间的信息交换方法，所以还需要其他的方法。

2.3.6 消息传递模型

- 进程间用消息来交换信息。一个正在执行的进程可以在任何时刻向另一个正在执行的进程发送一个消息；一个正在执行的进程也可以在任何时刻向正在执行的另一个进程请求一个消息。

2.3.6 消息传递模型

- 设计要点
 - 如何保证消息传递中不丢失？（ACK机制）
 - 如何命名进程，使得其地址唯一并确认身份？
 - 如何规范消息格式，降低处理时间？

2.3.6 消息传递模型

- 模型思想
 - 提供Send和Receive原语，用来传递消息
 - 进程通过发送和接收消息来实现互斥与同步
 - 重要的优点：不仅实现了同步，还能够传送大容量信息

消息传递系统的分类：

(1) 直接通信方式

- 发送进程利用OS提供的发送命令直接把消息发送给接收进程，并将它挂在接收进程的消息缓冲队列上。
- 接收进程利用OS提供的接收命令直接从消息缓冲队列中取得消息。

直接通信方式

- 要求发送进程和接收进程都以显示的方式提供对方的标识符，通常系统提供下述两条通信原语：
- 原语send (P, 消息)：把一个消息发送给进程P
- 原语receive (Q, 消息)：从进程Q接收一个消息

生产者 - 消费者消息传递解法

```
# define N 100
```

Producer进程

```
int item;  
message m;  
While(TRUE)  
{  
    Produce-Item(&item);  
    send(consumer,&m);  
}
```

Comsumer进程

```
int item;  
message m;  
While(TRUE)  
{  
    receive(producer,&m);  
    Consume-item(item);  
}
```

直接通信方式

基本原理：

- 在内存中开设缓冲区，发送进程将消息送入缓冲区，接收进程接收传递来的缓冲区。

Send 过程

- 在操作系统空间设置一组缓冲区；
- 当发送进程需要发送消息时，执行send系统调用，产生自愿性中断，进入操作系统；
- 操作系统为发送进程分配一个空缓冲区，并将所发送的消息从发送进程copy到缓冲区中；
- 将该载有消息的缓冲区连接到接收进程的消息链链尾，如此就完成了发送过程；
- 发送进程返回到用户态继续执行。

设信号量mutex为控制对缓冲区访问的互斥信号量，其初值为1。设SM为接收进程的资源信号量，表示等待接收的消息个数，其初值为0。

Send(m):

begin

向系统申请一个消息缓冲区

将发送区消息m送入新申请的消息缓冲区

P(mutex)

把消息缓冲区挂入接收进程的消息队列

V(mutex)

V(SM)

end

Receive 过程

- 当接收进程执行到receive接收原语时，也产生自愿性中断进入操作系统；
- 由操作系统将载有消息的缓冲区从消息链中取出，并把消息内容copy到接收进程空间；
- 收回缓冲区，如此就完成了消息的接收；
- 接收进程返回到用户态继续进行。

Receive 过程

Receive(n):

begin

P(SM)

P(mutex)

摘下消息队列中的消息n

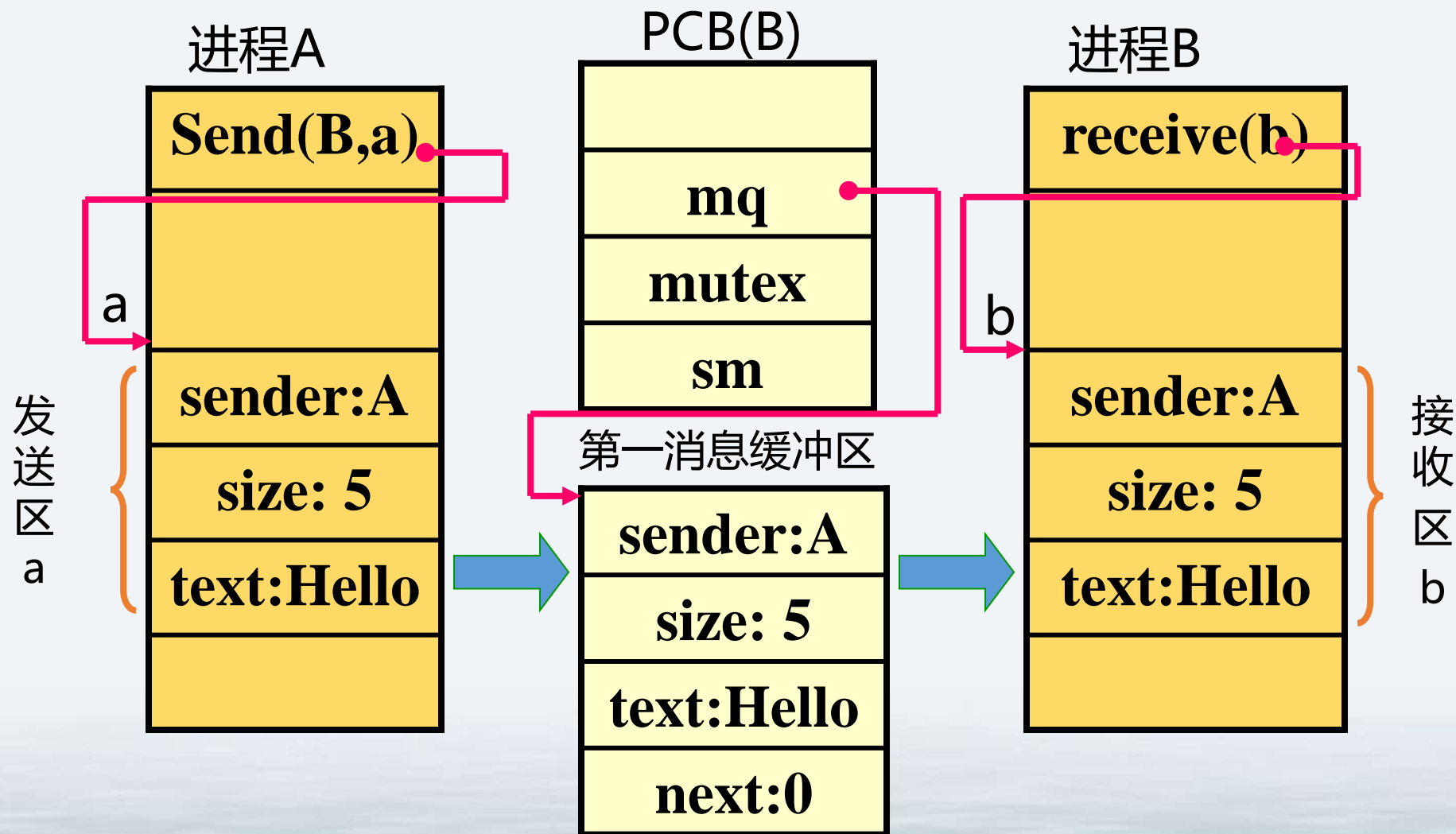
V(mutex)

将消息n从缓冲区复制到接收区

释放缓冲区

end

消息缓冲队列通信机制



(2) 间 接 通 信 方 式

- 进程间发送或接收消息通过信箱进行，消息可被理解成信件。也称信箱通信。
- 由发送进程申请建立一与接收进程链接的信箱。发送进程把消息送往信箱，接收进程从信箱中取出消息，从而完成进程间信息交换。

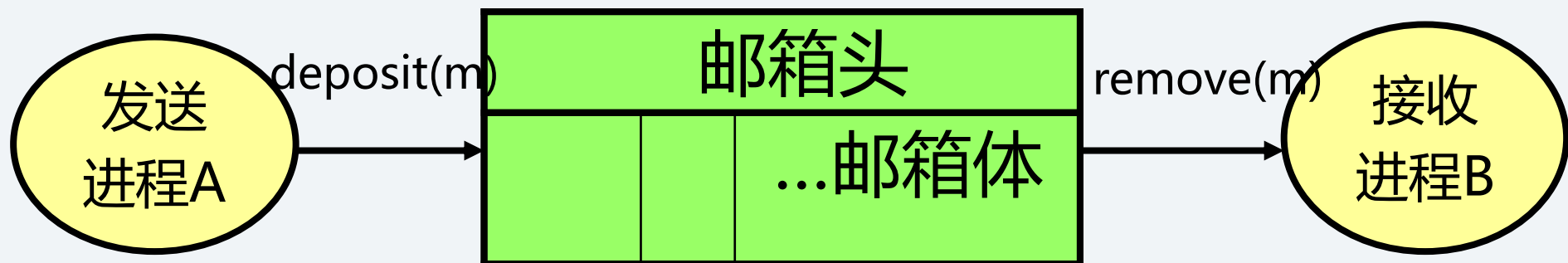
间接通信方式

两条通信原语：

- Send (mailbox, message) ;
- Receive (mailbox, message) ;

间接通信方式

- 注意：用户不必写出接收进程标识符，从而可以向不知名的进程发送消息；另外，消息在信箱中可以安全的保存，只允许核准的目标用户随时读取，可实现非实时通信。



- 信箱由信箱头和由若干格子组成的信箱体组成。
- 信箱中每个格子存放一封信，信箱中格子的数目和每格的大小在创建信箱时确定。

间接通信方式

进程间的通信要满足如下条件：

- 发送进程发送消息时，邮箱中至少要有有一个空格存放该消息。
- 接收进程接收消息时，邮箱中至少要有有一个消息存在。

2.4 经典的IPC问题



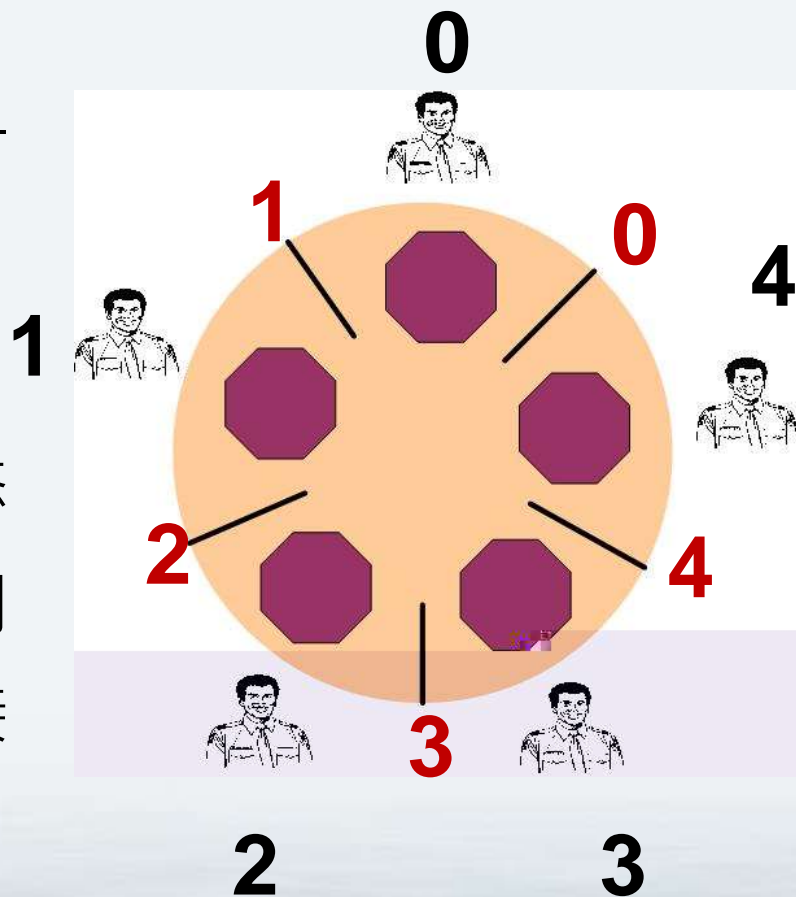
2.4 经典的IPC问题

- 2.4.1 哲学家就餐问题
- 2.4.2 读者-写者问题
- 2.4.3 睡眠理发师问题

经典问题：哲学家就餐问题

- 问题描述

- 五个哲学家坐在圆桌前，每人一份炒面
- 每个哲学家两侧各有一支筷子
- 哲学家处于吃面和思考两种状态
- 为了吃面，每个哲学家必须拿到两只筷子，并且每个人只能直接从自己的左边或右边去取筷子。



哲学家就餐问题

- 互斥关系分析
 - 筷子：一个筷子同一时刻只能被一个哲学家拿起
- 同步关系分析
 - 就餐：只有获得两个筷子后才能进餐

哲学家就餐问题

- 特殊情况考虑
 - 死锁：如果每个哲学家都拿起一只筷子，都饿死
 - 并行程度：五只筷子允许两人同时进餐

哲学家就餐问题

- 能不能只设置一个信号量，其初值为5，用以表示5支筷子？

第一种方案

- 用一个信号量表示一支筷子，由这5个信号量构成信号量数组

semaphore stick[5];
所有信号量初值为1。

```
semaphore stick[5]={1,1,1,1,1};  
void philosopher( int i )  
{  
    while(true){  
        思考;  
        P(stick[i]);  
        P(stick[i+1]%5);  
        进餐;  
        V(stick[i]);  
        V (stick[i+1]%5);  
    }  
}
```

第二种方案

• 极端情况，3号哲学家得到两支筷子（3和4），而此时第0位和第4位哲学家同时拿起右边筷子（1和0），则此时只会有一位哲学家能进餐

```
if ( i %2 == 1 )
```

```
{
```

```
    P(stick[ i %5 ] ) ;
```

```
    P(stick[ (i+1)%5] ) ;
```

```
    Eating();
```

```
    V(stick[ (i+1)%5] ) ;
```

```
    V(stick[ i %5 ] ) ;
```

```
}
```

```
{
```

```
    P(stick[ (i+1)%5] ) ;
```

```
    P(stick[ i %5 ] ) ;
```

```
    Eating();
```

```
    V(stick[ (i+1)%5] ) ;
```

```
    V(stick[ i %5 ] ) ;
```

```
}
```

哲学家就餐问题的直观解法

哲学家进程

```
#define    N    5
void philosopher(int i)
{
    While(TRUE) {
        think () ;
        take_forks(i);
        take_forks((i + 1) % N);
        eat();
        put_forks(i);
        put_forks((i + 1) % N);
    }
}
```

思考1：这样的解法有何问题？

思考2：对左右的筷子是否可用进行验证，这样的修改有何优缺点？

思考3：引入互斥信号量mutex？

思考4：需要引入几个信号量才能实现最优化的解法呢？

哲学家就餐问题的信号量解法

```
Semaphore mutex;  
#define N    5  
#define LEFT(i)  i % N  
#define RIGHT(i) (i+1) % N  
#define THINKING 0  
#define HUNGRY 1  
#define EATING 2  
int state[N];  
Mutex.value = 1;  
semaphore s[N]; //初始为0
```

哲学家进程

```
void philosopher(int i)  
{  
    While(TRUE){  
        think ();  
        take_forks(i);  
        eat();  
        put_forks(i);  
    }  
}
```

哲学家就餐问题的信号量解法

取叉子函数

```
void take_forks (int i)
{
    P(mutex);
    state[i] = HUNGRY;
    test(i);
    V(mutex);
    P(s[i]);
}
```

放叉子函数

```
void put_forks (int i)
{
    P(mutex);
    state[i] = THINKING;
    test(LEFT(i));
    test(RIGHT(i));
    V(mutex);
}
```

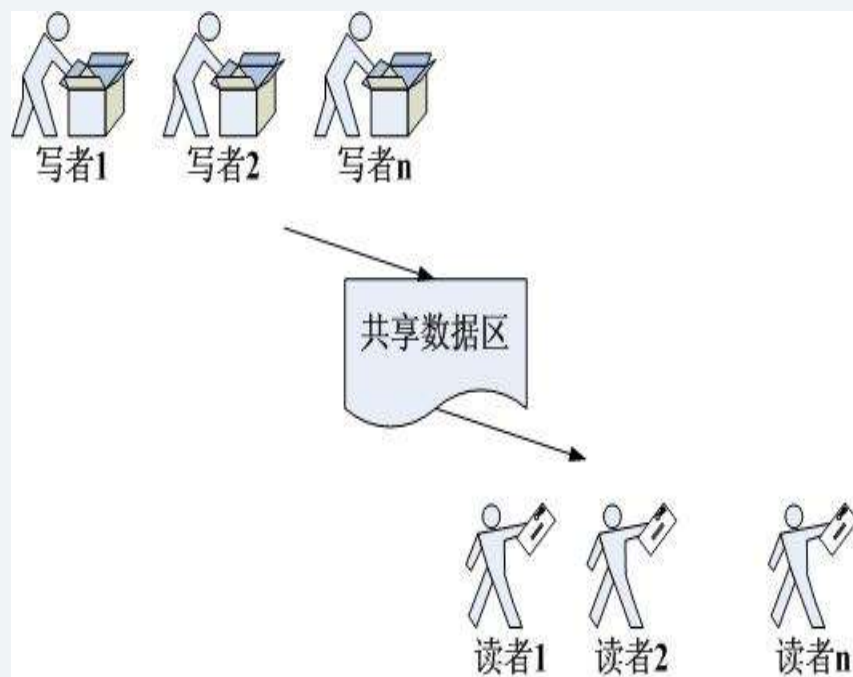
Test函数

```
void test (int i)
{
    if((state[i] == HUNGRY)
        && (state[ LEFT(i) ] ! = EATING)
        && (state[ RIGHT(i) ] != EATING))
    {
        state[i] = EATING;
        V(s[i]);
    }
}
```

经典问题：读者-写者问题

问题描述

- 写者向数据区放数据，读者从数据区获取数据
- 多个读者可同时读取数据
- 多个写者不能同时写数据，任一写者在完成写操作之前不允许其它读者或写者读写数据区



读者 - 写者问题的信号量解法

- 互斥关系分析
 - 读者和写者不能同时进入共享数据区
 - 多个写者不能同时进入共享数据区
 - 多个读者可以同时进入共享数据区

读者 - 写者问题的信号量解法

- 同步关系分析

- 读者进入缓冲区，写者必须等待
- 写者进入缓冲区，读者必须等待
- 读者优先：一旦有读者进入，则后续读者均可进入
- 写者优先：只要有写者等待，则后续读者必须等待

读者优先的信号量解法

- 整形变量rcount表示正在读的读者数目，初值为0。
- 信号量write用于保证读者和写者、写者和写者之间的互斥，初值为1。
- 信号量mutex用于保证对rcount 这个临界资源的互斥修改，初值为1。

读者优先的信号量解法

```
Semaphore mutex,  
write;  
  
Mutex.value = 1;  
Write.value = 1;  
  
int rcount = 0;
```

```
Reader进程  
While(TRUE)  
{  
    down(&mutex);  
    rcount++;  
    if(rcount == 1)  
        down(&write);  
    up(&mutex);  
    Read_Action();  
    down(&mutex);  
    rcount--;  
    if(rcount == 0)  
        up(&write);  
    up(&mutex);  
}
```

```
Writer进程  
While(TRUE)  
{  
    down(&write);  
    Write_Action();  
    up(&write);  
}
```


写者优先的信号量解法

- 信号量Concur表示当前是否有写者，若有，则读者必须等到写者完成，用此控制写者优先
- 信号量write用于保证读者和写者、写者和写者之间的互斥，初值为1。
- 整形变量rcount表示正在读的读者数目，初值为0。
- 信号量rmutex用于保证对rcount 这个临界资源的互斥修改，初值为1。
- 整形变量wcount表示当前的写者数目，初值为0。
- 信号量wmutex用于保证对wcount 这个临界资源的互斥修改，初值为1。

写者优先的信号量解法

```
Semaphore  
rmutex, wmutex,  
writem, concur;  
  
Rmutex.value = 1;  
wmutex.value = 1  
Write.value = 1;  
Concur.value = 1;  
  
int rcount = 0;  
int wcount = 0;
```

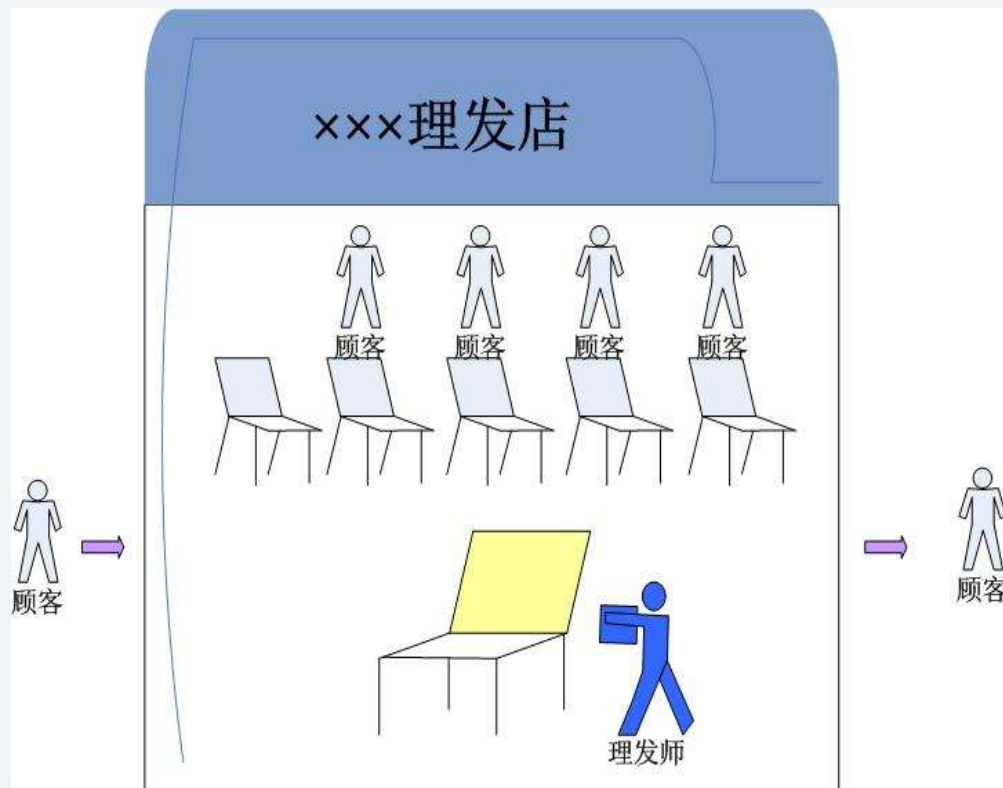
```
Reader进程  
While(TRUE)  
{  
    P(concur);  
    P(rmutex);  
    rcount++;  
    if(rcount == 1)  
        P(write);  
    V(rmutex);  
    V(concur);  
    Read_Action();  
    P(rmutex);  
    rcount--;  
    if(rcount == 0)  
        V(write);  
    V(rmutex);  
}
```

```
Writer进程  
While(TRUE)  
{  
    P(wmutex);  
    wcount++;  
    if(wcount == 1)  
        P(concur);  
    V(wmutex);  
    P(Write);  
    Write_Action();  
    V(write);  
    P(wmutex);  
    wcount--;  
    if(wcount == 0)  
        V(concur);  
    V(wmutex);  
}
```

经典问题：睡眠理发师问题

问题描述

- 一把理发椅，N把等待座位
- 理发师为理发椅上的顾客理发，没有顾客就在理发椅上睡觉
- 有一个顾客时需要叫醒理发师
- 多个顾客时需要在等待座位上等候



睡眠理发师问题的信号量解法

- 互斥关系分析
 - 理发椅上只能有一位顾客
 - 等待座位是有限缓冲区
- 同步关系分析
 - 只要存在顾客，理发师就不能睡觉
 - 所有理发师都在理发，则顾客就要等待

睡眠理发师问题的信号量解法

引入3个信号量和一个控制变量：

- 控制变量waiting用来记录等候理发的顾客数，初值均为0;
- 信号量customers用来记录等候理发的顾客数，并用作阻塞理发师进程，初值为0;
- 信号量barbers用来记录正在等候顾客的理发师数，并用作阻塞顾客进程，初值为0;
- 信号量mutex用于互斥访问waiting，初值为1。

睡眠理发师问题的信号量解法

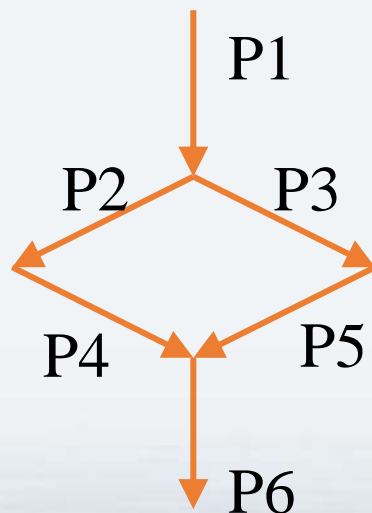
```
#define CHAIRS  N  
  
Semaphore customer ,  
barer , mutex ;  
  
Customer.value = 0;  
  
barer.value = 0;  
  
mutex.value = 1;  
  
int waiting = 0;
```

```
barber进程  
While(TRUE)  
{  
    P(customer);  
    P(mutex);  
    Waiting --;  
    V(barbers);  
    V(mutex);  
    cut_hair();  
}
```

```
customer进程  
While(TRUE)  
{  
    P(mutex);  
    if(waiting < CHAIRS)  
    {  
        waiting += 1;  
        V(customer);  
        V(mutex);  
        P(barber);  
        get_haircut();  
    }  
    Else  
        {V(mutex);}  
}
```

【思考题 1】

- 设有6个进程P1、P2、P3、P4、P5、P6，它们有如图所示的并发关系。试用P、V操作实现这些进程间的同步。（假设每个进程只运行一次）



【 问题的解 】

Semaphore b23=0, b4=0, b5=0, b6=0;

Main() {

P1();

P2();

P3();

P4();

P5();

P6();

}


```
P1( )  
{  
    ...  
    V(b23);  
    V(b23);  
}
```

```
P2( )  
{  
    P(b23);  
    ...  
    V(b4);  
}
```

```
P3( )  
{  
    P(b23);  
    ...  
    V(b5);  
}
```

```
P4( )  
{  
    P(b4);  
    ...  
    V(b6);  
}
```

```
P5( )  
{  
    P(b5);  
    ...  
    V(b6);  
}
```

```
P6( )  
{  
    P(b6);  
    P(b6);  
    ...  
}
```

【思考题 2】

- 桌上有一空盘，最多允许存放一只水果。爸爸可向盘中放一个苹果或放一个桔子，儿子专等吃盘中的桔子，女儿专等吃苹果。
- 试用P、V操作实现爸爸、儿子、女儿三个并发进程的同步与互斥。
- **【提示】** 设置一个信号量表示可否向盘中放水果，一个信号量表示可否取桔子，一个信号量表示可否取苹果。

【思考题 2】

- 是否需要设置一个信号量mutex，用于对盘子操作的互斥？

【问题的解】

设置三个信号量empty、orange、apple，初值分别为1、0、0。分别表示可否向盘中放水果，可否取桔子，可否取苹果。

```
Father( )  
{  
  while(true)  
  {  
    P(empty);  
    将水果放入盘中;  
    if 桔子 then V(orange);  
    else V(apple);  
  }  
}
```

```
Son( )  
{  
  while(true)  
  {  
    P(orange);  
    取桔子;  
    V(empty);  
    吃桔子;  
  }  
}
```

```
Daughter( )  
{  
  while(true)  
  {  
    P(apple);  
    取苹果;  
    V(empty);  
    吃苹果;  
  }  
}
```

【思考题 3】

- 某打印机服务器采用假脱机技术为用户服务，已知磁盘打印缓冲目录长度为100，如下图:进程A或进程B接收用户的打印请求，并将其存入缓冲目录中，其存放位置由内存中的变量in指示，并修改in的值，进程printer负责从缓冲目录中取出打印请求，送向打印机，取请求的位置由内存中的变量out指示。假设当前in=7，out=4，请信号量的PV操作实现这三个进程的并发执行。

【 问 题 分 析 】

信号量设置：

- mutex：可否访问主存变量in，初值
mutex=1;
- Sin：缓冲目录中当前空余的个数，初值
Sin=97;
- Sout：缓冲区中打印请求的个数：Sout=3;

【问题的解】

Process A()

```
{  
    收到打印请求;  
    P(Sin);  
    P(mutex);  
    将打印请求存入in指示  
    的目录位置;  
    In=(in+1)%100;  
    V(mutex);  
    V(Sout);  
}
```

Process B()

```
{  
    收到打印请求;  
    P(Sin);  
    P(mutex);  
    将打印请求存入in指示  
    的目录位置;  
    In=(in+1)%100;  
    V(mutex);  
    V(Sout);  
}
```

Process printer()

```
{  
    P(Sout);  
    P(mutex);  
    从缓冲out位置取一个  
    打印请求;  
    out=(out+1)%100;  
    V(mutex);  
    V(Sin);  
}
```

经验总结

- 进程间互斥：信号量初值为1，对同一信号量的P、V操作在一个进程中。
- 进程间同步：信号量初值为0（不绝对，由实际情况决定），对同一信号量的P、V操作在不同进程中。
- 进程前趋关系：信号量初值为0，对同一信号量的P、V操作在不同进程中。P操作在受限进程的最前面，V操作在前趋进程的最后。

2.5 调度



2.5 调度

- CPU资源管理——多道程序设计面临的挑战
 - 批处理系统：如何安排内存中多个作业的运行顺序？
 - 交互式系统：如何更好应对不同的交互式请求？
 - 实时系统：如何保证实时服务的高质量？

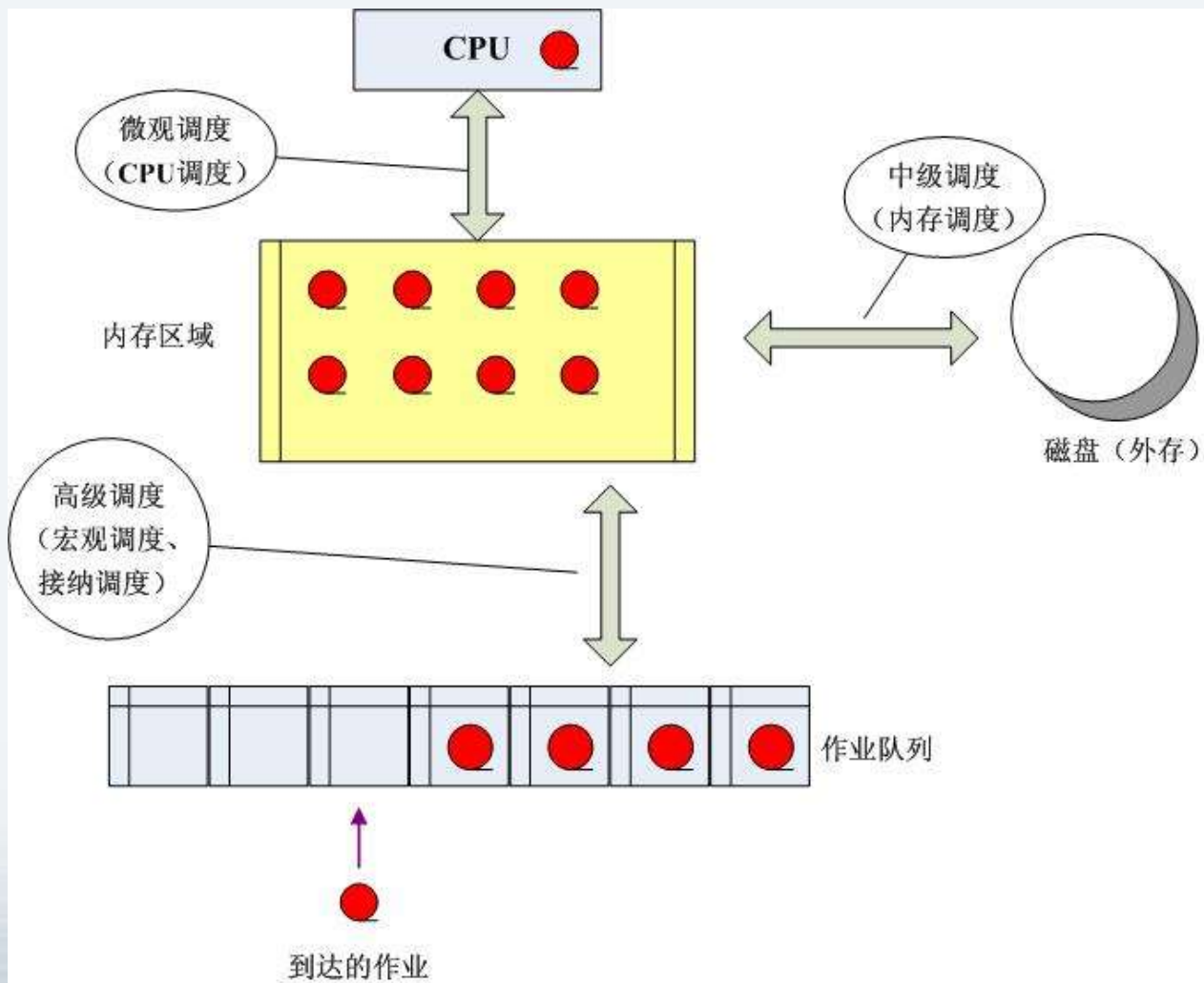
2.5 调度

- 进程调度——有效的管理CPU资源
 - When: 何时进行进程调度?
 - How: 遵循何种规则完成调度?
 - What: 调度过程中需要完成哪些工作?

2.5.1 调度介绍

- 调度的级别
 - 高级调度：也称宏观调度，决定哪些程序可以进入系统
 - 中级调度：也称内存调度，决定内存中程序的位置和状态
 - 低级调度：也称微观调度，决定CPU资源在就绪进程间的分配

调度的三个层次



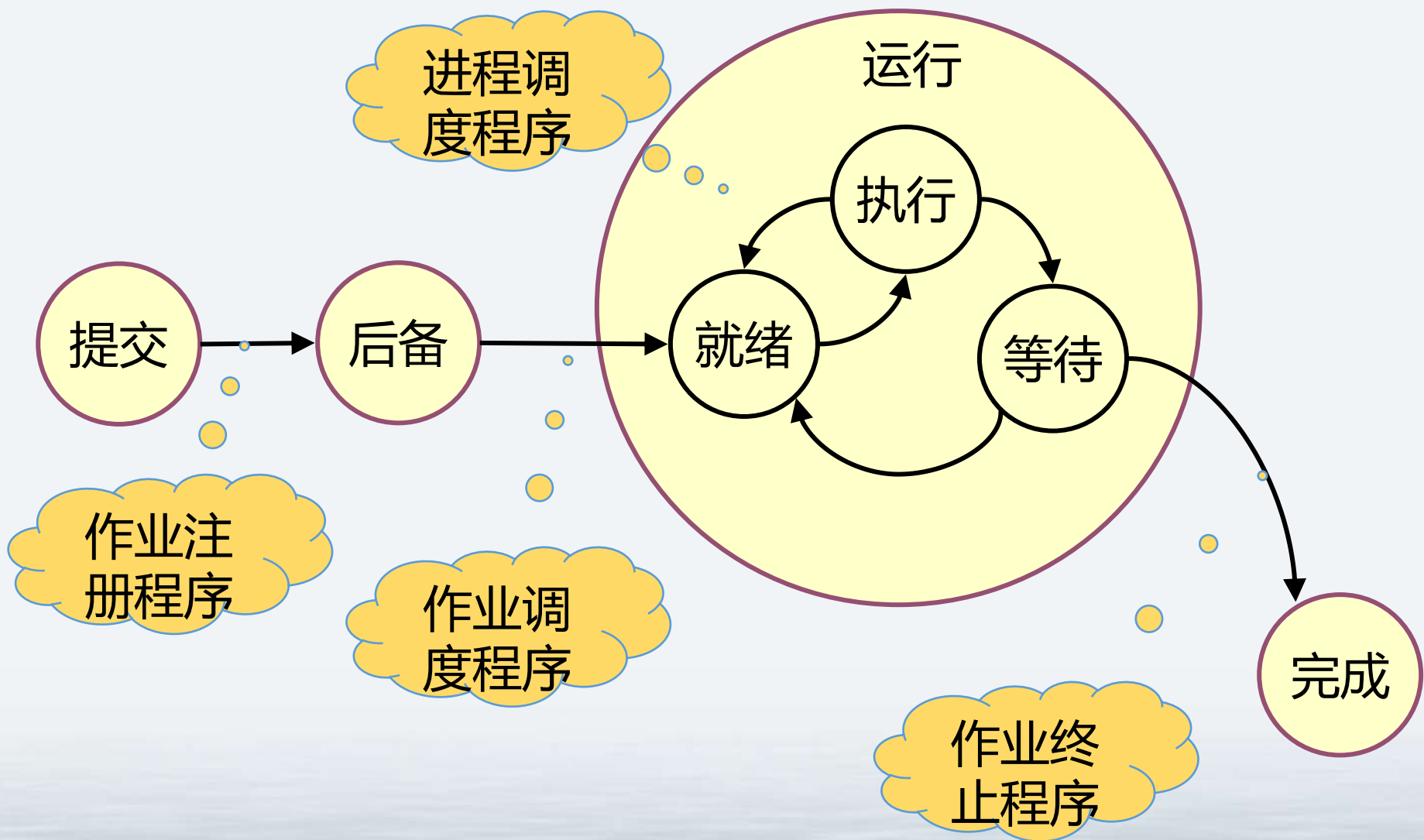
高级调度

- 又称为作业调度或长程调度。
- 用于决定把后备队列中的哪些作业调入内存，为它们创建进程、分配必要的资源，再将新创建的进程排在就绪队列上，准备执行。
- 在批处理系统中，大多配有作业调度，但在分时和实时系统中，却往往不配置作业调度。作业调度的运行频率较低，通常为几分钟一次。

高级调度

- 作业从进入到运行结束，一般需要经历四个阶段
 - 提交状态：作业从输入设备进入外存储器时的状态
 - 后备状态：作业已输入到磁盘，等待调入内存
 - 运行状态：作业已进入内存，对应进程的各种状态
 - 完成状态：当进程正常运行结束或因发生错误而终止时，作业进入完成状态

作业的状态转换



高级调度

执行作业调度时，需要解决：

- 一次接纳多少作业：即允许多少个作业同时在内存中运行。
- 接纳哪些作业：即应将哪些作业调入内存，取决于所采用的算法。
 - 比如先来先服务调度算法；或者是短作业优先调度算法；还有基于作业优先权的调度算法等。

低级调度

- 又称为进程调度、短程调度。
- 用于决定就绪队列中的哪个进程能获得处理器, 并将处理机分配给该进程。
- 进程调度程序是操作系统最为核心的部分, 进程调度策略的优劣直接影响到整个系统的性能。
- 三种类型的操作系统中, 都必须配置此级调度。

进 程 调 度

有两类进程调度方式：

① 非抢占方式

一旦把处理机分配给某个进程后，让该进程一直执行，直到该进程完成或者发生某事件而阻塞。

进程调度--非抢占方式

- 优缺点：非抢占调度方式的优点是实现简单、系统开销小。缺点是它难于满足紧迫任务立即执行的要求，可能造成难以预料后果，因此，在实时要求比较严格的实时系统中不宜采用非抢占调度方式。

进 程 调 度

② 抢占方式

当一进程正在处理机上执行时，系统可根据某种原则暂停它的执行，并将已分配给它的处理机重新分配给另一个进程。

进程调度--抢占调度

- 优缺点：抢占调度方式的优点是可以为全体进程提供更好的服务，防止一个进程长期占用处理机。缺点是开销较大。抢占调度方式适用于分时系统和大多数实时系统

进程调度

抢占的原则有：

- 优先权原则：就绪的高优先权进程有权抢占低优先权进程的 CPU。
- 短作业优先原则：就绪的短作业(进程)有权抢占长作业(进程)的 CPU。
- 时间片原则：一个时间片用完后，系统重新进行进程调度。

调度算法分类

- 批处理：通常采用非抢占式算法
- 交互式：通常采用抢占式算法
- 实时：通常采用抢占式算法

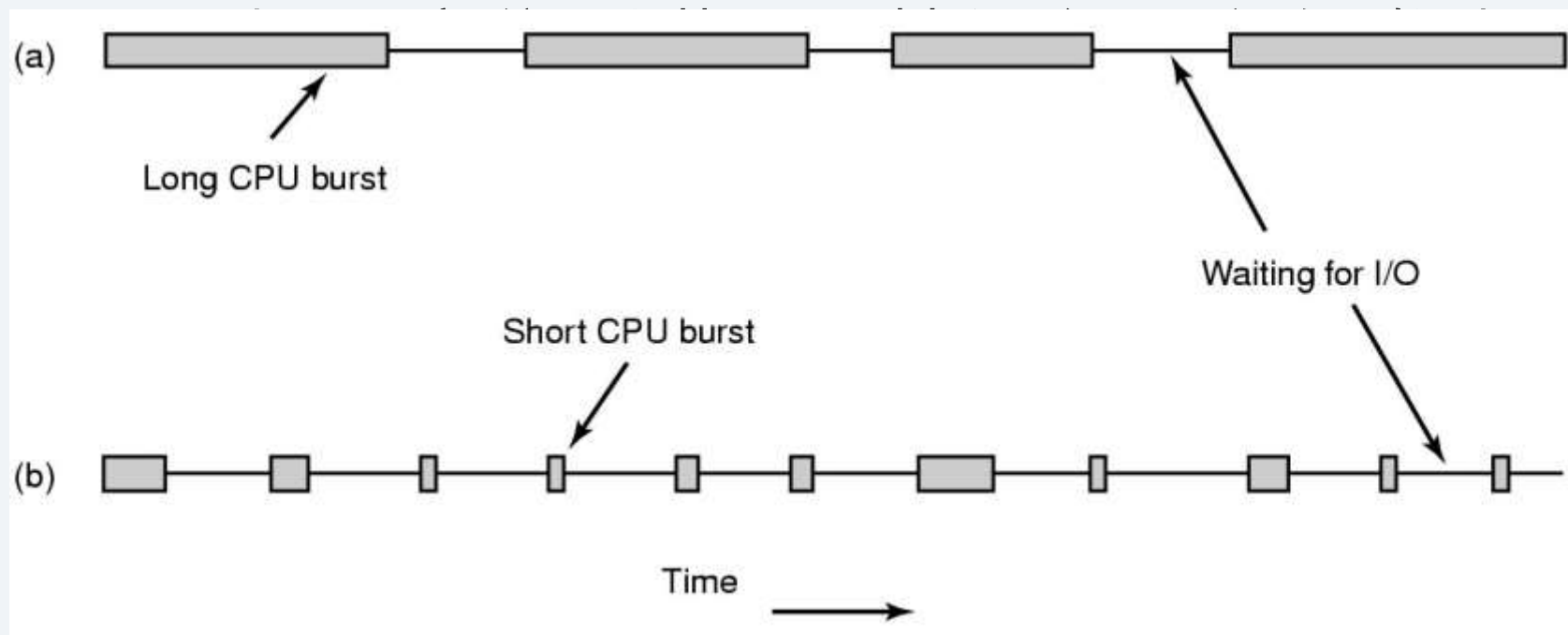
中级调度

- 又称平衡负载调度、中程调度。
- 目的是为了提_高内存利用率和系统吞吐量。
- 实质是进程的内外存对换功能：将外存中已具备运行条件的进程换入内存，而将内存中暂不能运行的某些进程换出至外存。

调度的三个层次

- 在三种调度中，进程调度的运行频率最高，作业调度的周期较长，中级调度的运行频率在上述两者之间。

调度基本思想



When——何时调度？

引起进程调度的因素：

- 创建一新进程后，需决定是运行父进程还是子进程；
- 正在执行的进程执行完毕；
- 执行中的进程阻塞在I/O请求或信号量上；
- I/O中断发生时。

进程调度的考量标准

- 响应时间：进程自进入就绪队列开始至进程占用CPU之间的时间间隔
- 周转时间：进程自进入就绪队列开始至进程结束之间的时间间隔
- CPU吞吐量：单位时间内运行结束的进程个数

进程调度的原则

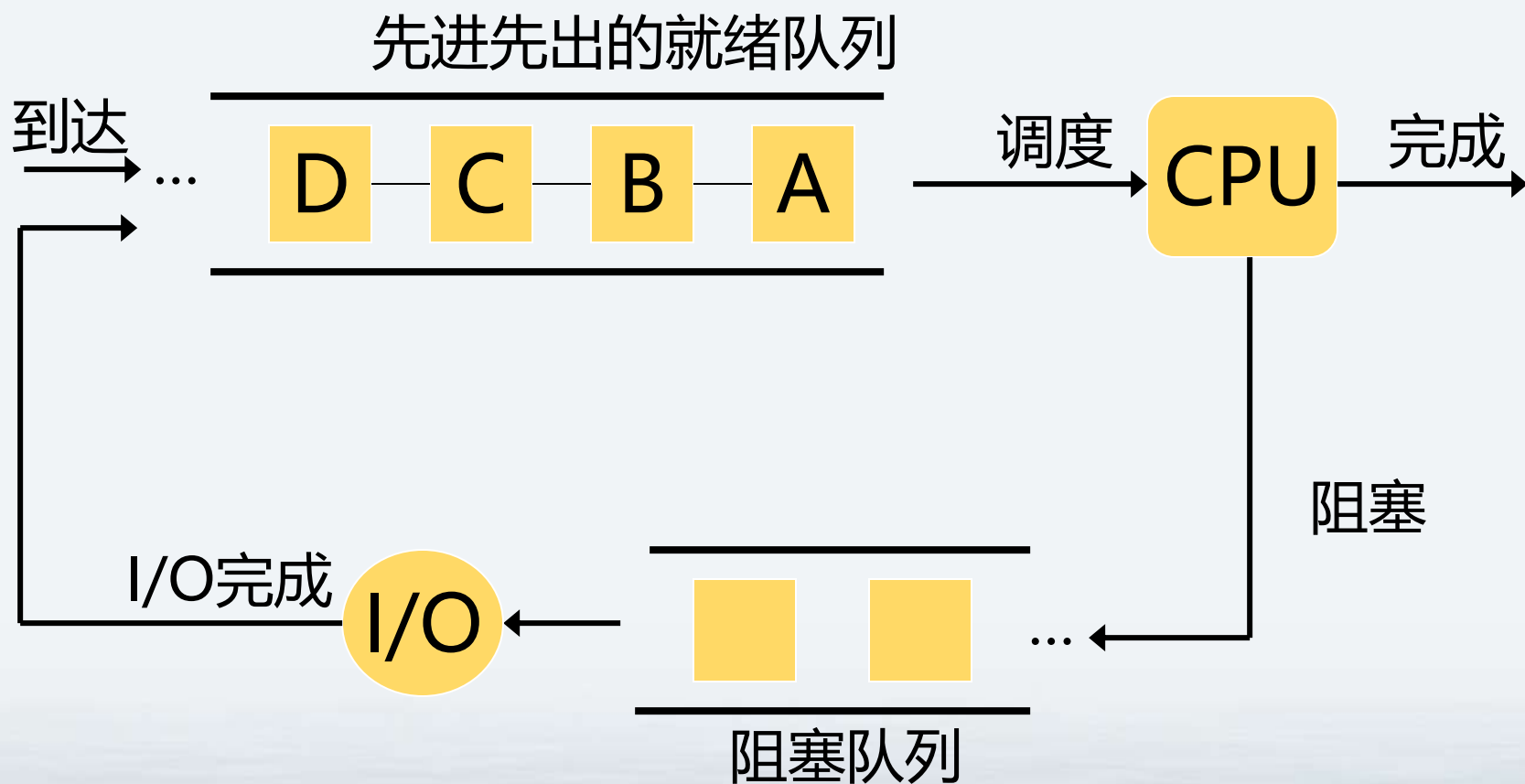
- 公平性原则：应保证每个进程获得合理的CPU份额
- 有效性原则：CPU资源应得到最大限度的利用
- 友好性原则：响应时间快，与用户（人）交互的时间应尽可能的短
- 快捷性原则：周转时间短。批处理作业的处理时间尽可能的短
- 广泛性原则：吞吐量。单位时间内完成的作业尽可能的多

2.5.2 批处理系统中的调度

1. 先来先服务调度算法 (FCFS)

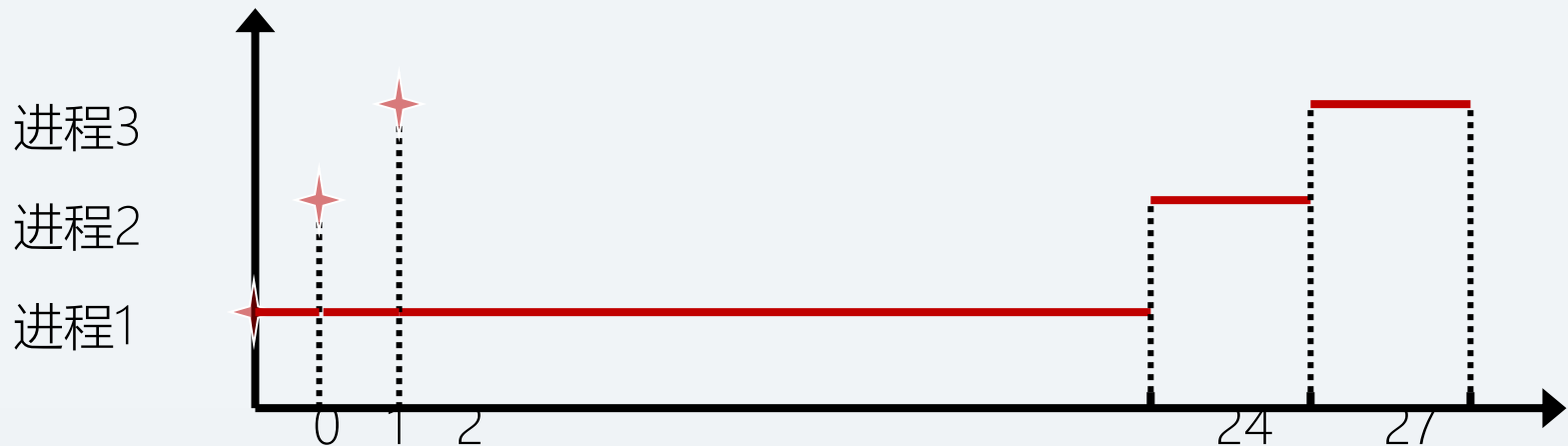
- 最简单的调度算法，可用于作业或进程调度。
- 用于作业调度时：从后备队列中选择最先进入队列的一个或几个作业，将它们调入内存，分配必要的资源，创建进程并放入就绪队列。
- 用于进程调度时：从就绪队列中选择最先进入该队列的进程，将处理机分配给它。
- 只顾及作业等候时间，没考虑作业要求服务时间的长短。

先来先服务(FCFS)



先来先服务(FCFS)

例：设有三个进程，编号分别为1, 2, 3。各进程依次到达，相差一个时间单位。下图是采用FCFS方式调度时这三个进程的执行情况：



先来先服务(FCFS)

周转时间/运行时间

| 进程 | 到达时间 | 运行时间 | 开始时间 | 完成时间 | 周转时间 | 带权周转时间 |
|--------------------------------|------|------|------|------|------|--------|
| 1 | 0 | 24 | 0 | 24 | 24 | 1 |
| 2 | 1 | 3 | 24 | 27 | 26 | 8.67 |
| 3 | 2 | 3 | 27 | 30 | 28 | 9.33 |
| 平均周转时间: 26 平均带权周转时间: 6.33 | | | | | | |

FCFS调度算法性能

先来先服务(FCFS)

- 根据进程到达就绪队列的时间来分配中央处理机，一旦一个进程获得了中央处理机，就一直运行到结束，先来先服务是非抢占调度。
- 有利于长进程（作业），不利于短进程（作业）；有利于CPU繁忙型进程（作业），不利于I/O繁忙型进程（作业）。
- 实现简单，但效率较低。

先来先服务(FCFS)

- 在当今系统中，先来先服务很少作为主要调度模式，而是常常嵌套在其它的调度模式中。
- 例如，许多调度模式根据优先级将处理机分配给进程，但具有相同优先级的进程却按先进先出进行分配。

2.5.2 批处理系统中的调度

2. 短作业(进程)优先调度算法 (SJF/ SPF)

- 用于作业调度时： SJF调度算法。从后备作业队列中选择估计运行时间最短的一个或几个作业，将它们调入内存，分配必要的资源，创建进程并放入就绪队列。
- 用于进程调度时： SPF调度算法。从就绪队列中选出估计运行时间最短的进程，将处理机分配给它。该进程一直运行下去，直到完成或因某种原因而阻塞。

短作业(进程)优先调度算法

| 调度算法 \ 作业情况 | 进程名 | A | B | C | D | E | 平均 |
|-------------|--------|----------------|----------------|-----------------|----------------|-----------------|------|
| | 到达时间 | 0 | 1 | 2 | 3 | 4 | |
| | 服务时间 | 4 | 3 | 5 | 2 | 4 | |
| FCFS (a) | 完成时间 | 4 | 7 | 12 | 14 | 18 | |
| | 周转时间 | 4 | 6 | 10 | 11 | 14 | 9 |
| | 带权周转时间 | 1 | 2 | 2 | 5.5 | 3.5 | 2.8 |
| SJF (b) | 完成时间 | 4 ^① | 9 ^③ | 18 ^⑤ | 6 ^② | 13 ^④ | 运行步骤 |
| | 周转时间 | 4 | 8 | 16 | 3 | 9 | |
| | 带权周转时间 | 1 | 2.67 | 3.1 | 1.5 | 2.25 | |

短作业(进程)优先调度算法

优缺点:

- 算法易于实现, 能有效降低作业的平均等待时间;
- 对长作业不利, 有可能导致长作业(进程)长期不被调度;
- 未能依据作业的紧迫程度来划分执行的优先级;
- 难以准确估计作业(进程)的执行时间, 未必能真正做到短作业优先, 从而影响调度性能。

2.5.3 交互式系统中的调度

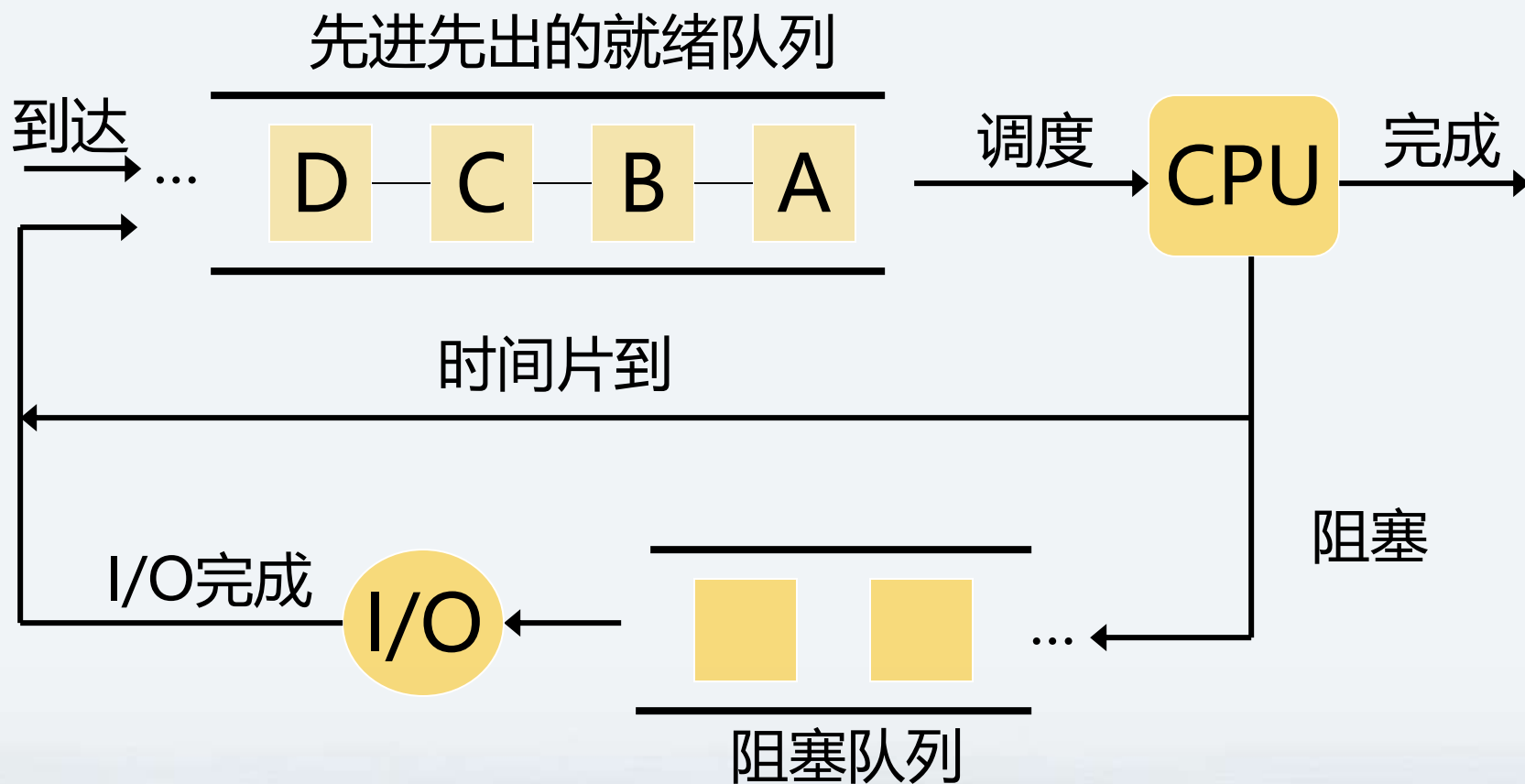
1. 时间片轮转法 (Round-Robin, RR)

- 核心思想：每个进程运行固定的时间片，然后调入下一个进程
- 实现机理：维护就绪进程队列，采用FIFO方式读取
- 特殊控制：时间片内发生阻塞或结束，则立即放弃时间片
- 抢占式调度算法

时间片轮转法(RR)

- 基本思想：系统将所有就绪进程按先进先出的原则排成一个队列。系统规定一个时间片，调度程序每次总是选出队首进程，让其在CPU上运行一个时间片的时间。在使用完一个时间片后，即使进程还未运行完毕，也必须释放CPU给下一个就绪的进程，它自己则返回到就绪队列末尾，重新排队等待再次运行。

时间片轮转法(RR)



问题的提出

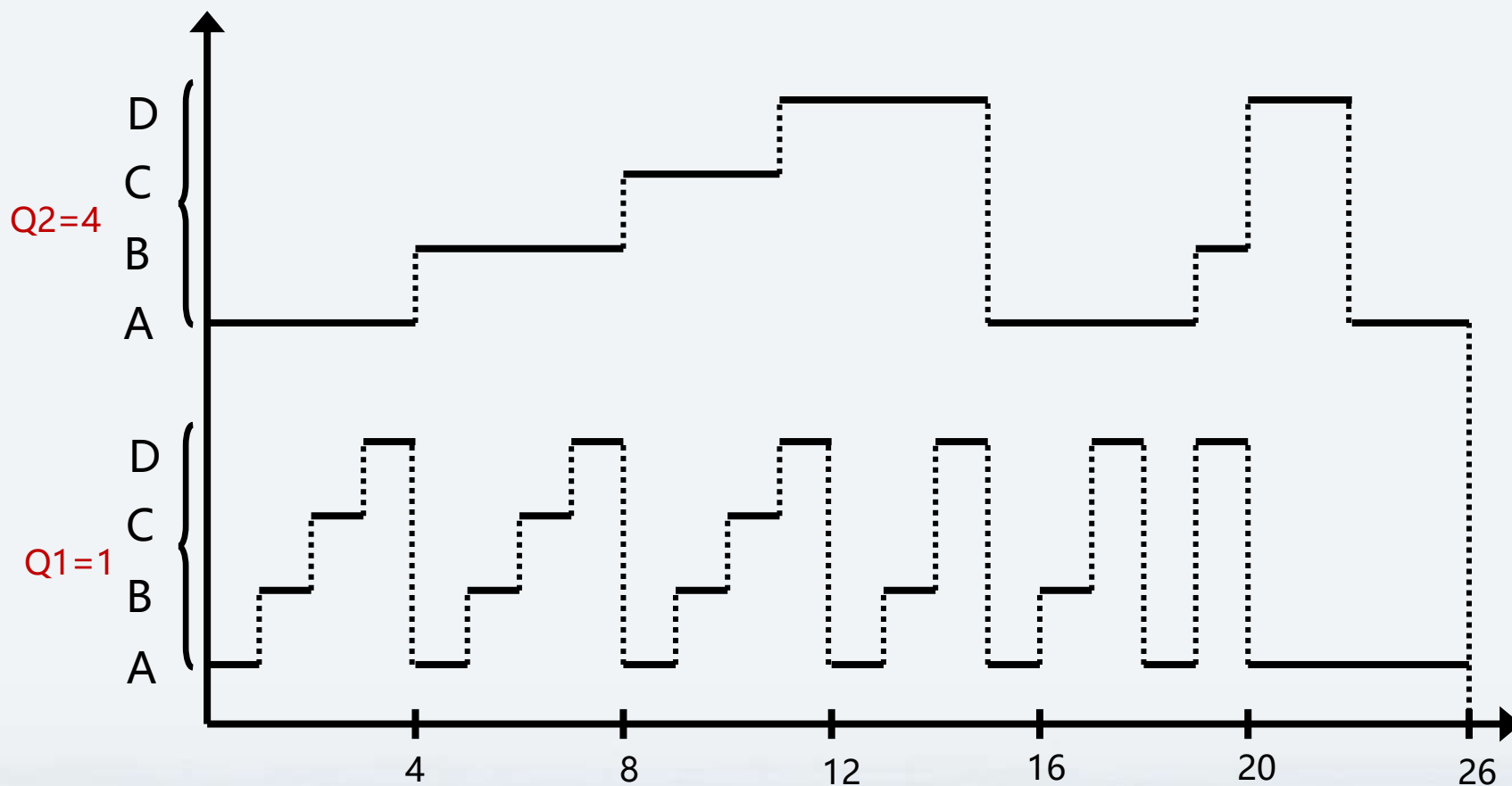
- 时间片的大小对系统的调度性能是否有影响？
- 时间片大小如何选取？

时间片轮转法(RR)

例：4个进程A, B, C, D依次进入就绪队列，但彼此相差时间很短，可以近似认为“同时”到达。4个进程分别需要运行12, 5, 3, 6个时间单位。

- 取两种不同大小时间片进行对比：时间片Q1 = 1个时间单位；时间片Q2 = 4个时间单位。

时间片轮转法(RR)



Q1=1, Q2=4时进程运行情况

RR调度算法的性能指标

| 时间片 | 进程名 | 到达时间 | 运行时间 | 开始时间 | 完成时间 | 周转时间 | 带权周转时间 |
|------|----------------------------------|------|------|------|------|------|--------|
| Q1=1 | A | 0 | 12 | 0 | 26 | 26 | 2.17 |
| | B | 0 | 5 | 1 | 17 | 17 | 3.4 |
| | C | 0 | 3 | 2 | 11 | 11 | 3.67 |
| | D | 0 | 6 | 3 | 20 | 20 | 3.33 |
| | 平均周转时间: 18.5 平均带权周转时间: 3.14 | | | | | | |

RR调度算法的性能指标

| 时间片 | 进程名 | 到达时间 | 运行时间 | 开始时间 | 完成时间 | 周转时间 | 带权周转时间 |
|------|---------------------------------|------|------|------|------|------|--------|
| Q1=4 | A | 0 | 12 | 0 | 26 | 26 | 2.17 |
| | B | 0 | 5 | 4 | 20 | 20 | 4 |
| | C | 0 | 3 | 8 | 11 | 11 | 3.67 |
| | D | 0 | 6 | 11 | 22 | 22 | 3.67 |
| | 平均周转时间：19.75 平均带权周转时间：3.38 | | | | | | |

时间片轮转法(RR)

在轮转法中，时间片长度的选取非常重要，时间片长度的选择会直接影响系统开销和响应时间

- 当时间片很大时，若每个进程得到比完成该进程多的处理机时间，此时轮转调度模式退化为先进先出模式。
- 当时间片非常小时，上下文切换开销就成了决定因素，系统性能降低，很多时间都消耗在处理器的转换上，处理器真正用于运行用户程序的时间将会减少。

决定时间片长短的主要因素

- 系统的响应时间。在进程数目一定时，时间片的长短直接正比于系统对响应时间的要求。
- 就绪队列进程的数目。当系统要求的响应时间一定时，时间片的大小反比于就绪队列中的进程数。
- 进程切换时间。若执行进程调度时的上下文切换时间为 t ，时间片为 q ，为保证系统开销不大于某个标准，应使比值 t/q 不大于某一数值，如 $1/10$ 。
- CPU运行指令速度。CPU运行速度快，则时间片可以短些；反之，则应取长些。

2.5.3 交互式系统中的调度

2. 优先级调度

- 核心思想：为每个进程赋予不同级别的优先级，优先级越高越优先
- 实现机理：维护一个优先级队列，自顶向下依次读取
- 特殊控制：静态优先级与动态优先级概念
- 优缺点分析
 - 优点：响应时间快，易于调整。最通用的方法。
 - 缺点：死规则，如何保证周转时间和吞吐量？

优先级调度

- 静态优先权：在创建进程时确定的，且在进程的整个运行期间保持不变。
- 静态优先权法简单易行，但不够精确，很可能出现优先权低的作业（进程）长期没有被调度的情况。

优先级调度

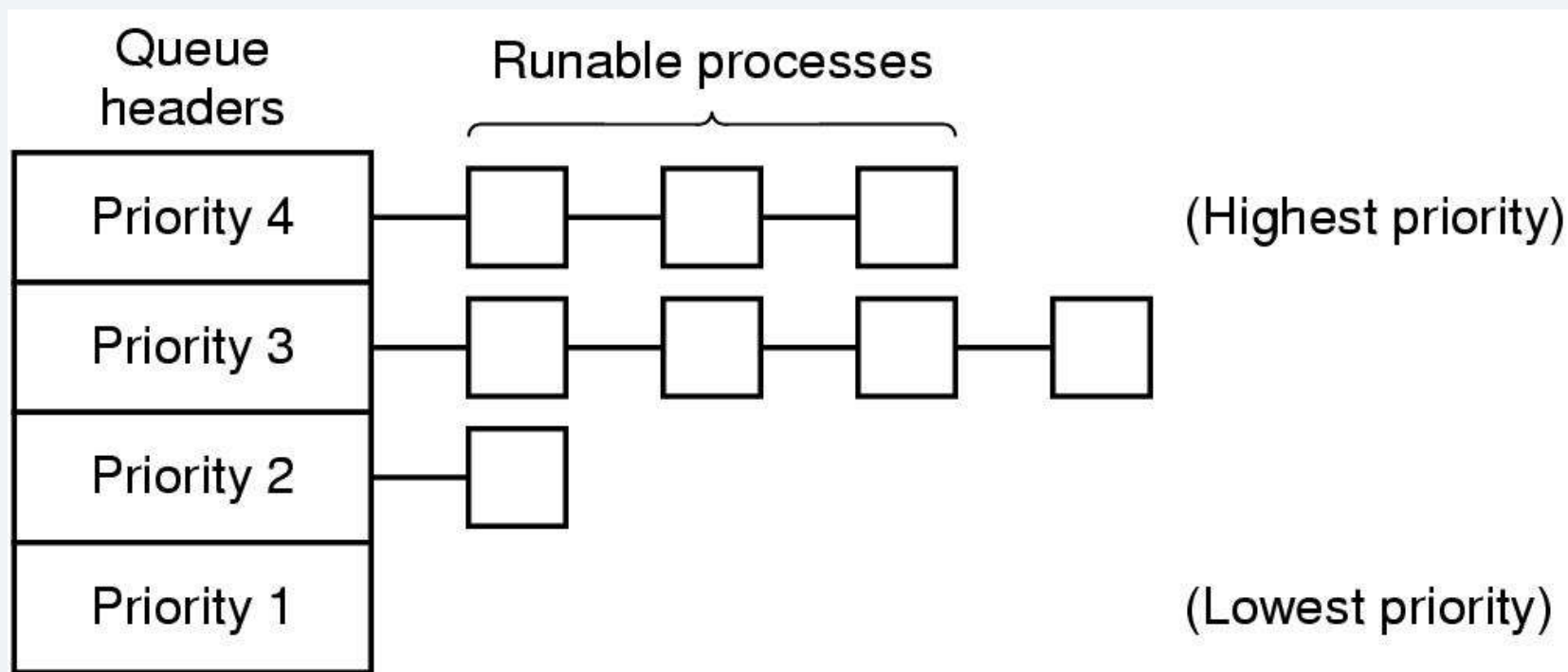
确定静态优先权的主要依据：

- 进程类型：如系统进程的优先级应高于用户进程。
- 进程对资源的要求：进程所申请的资源越多，估计运行时间越长，优先权越低。
- 用户要求：如由用户的紧迫程度及用户所付费用的多少来确定进程的优先权。

优先级调度

- 动态优先权：在创建进程时所确定的优先权，可以随着进程的推进而改变。
- 确定动态优先级的主要依据有：
 - 进程占有CPU时间的长短。
 - 就绪进程等待CPU时间的长短。

优先级调度

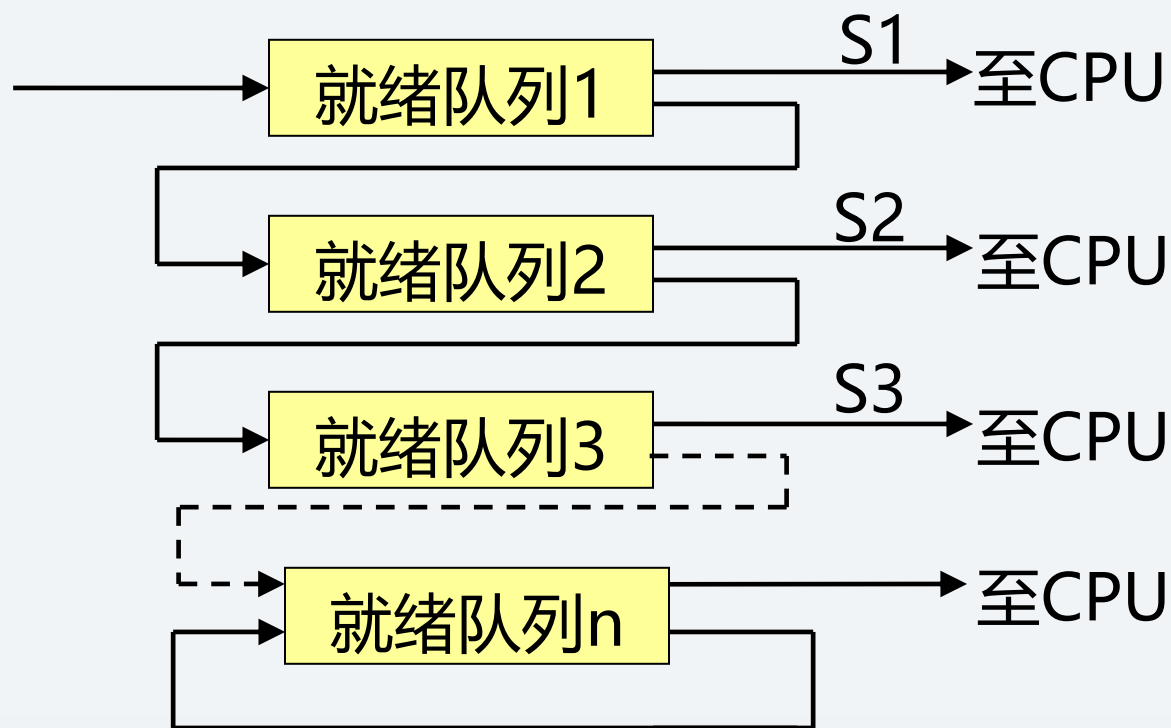


2.5.3 交互式系统中的调度

3. 多级反馈队列调度算法 (Feedback, FB)

- 将就绪队列分为N级，每个就绪队列分配给不同的时间片，**队列优先权越高，时间片越小**；
- 新进程进入内存后，放入第一队列末尾，按FCFS原则等待调度，如果在一个时间片结束时没完成，将该进程转入第二队列末尾重新等待调度执行..... 如此下去，当一个长作业从第一级队列降到最后一级队列时，便在该队列中采取时间片轮转方式运行。

多级反馈队列调度算法 (FB)



时间片: $S1 < S2 < S3$

多级反馈队列调度算法（FB）

- 仅当第一队列为空时，调度程序才调度第二队列中的进程运行.....类推之，仅当第 $1 \sim (i-1)$ 级队列均空时，才调度第 i 级队列上的进程执行。
- 如果处理机正在为某队列的进程服务，又有新进程插入到较高优先级的队列中，则新进程将抢占正在运行进程的处理机（抢占式），被抢占进程回到原来一级就绪队列末尾。
- 进程由于等待而放弃CPU后，进入等待队列，一旦等待的事件发生，则回到原来的就绪队列

多级反馈队列调度算法（FB）

特点：长、短作业兼顾，有较好的响应时间

- 短作业一次完成；
- 中型作业周转时间不长；
- 大型作业不会长期不处理。

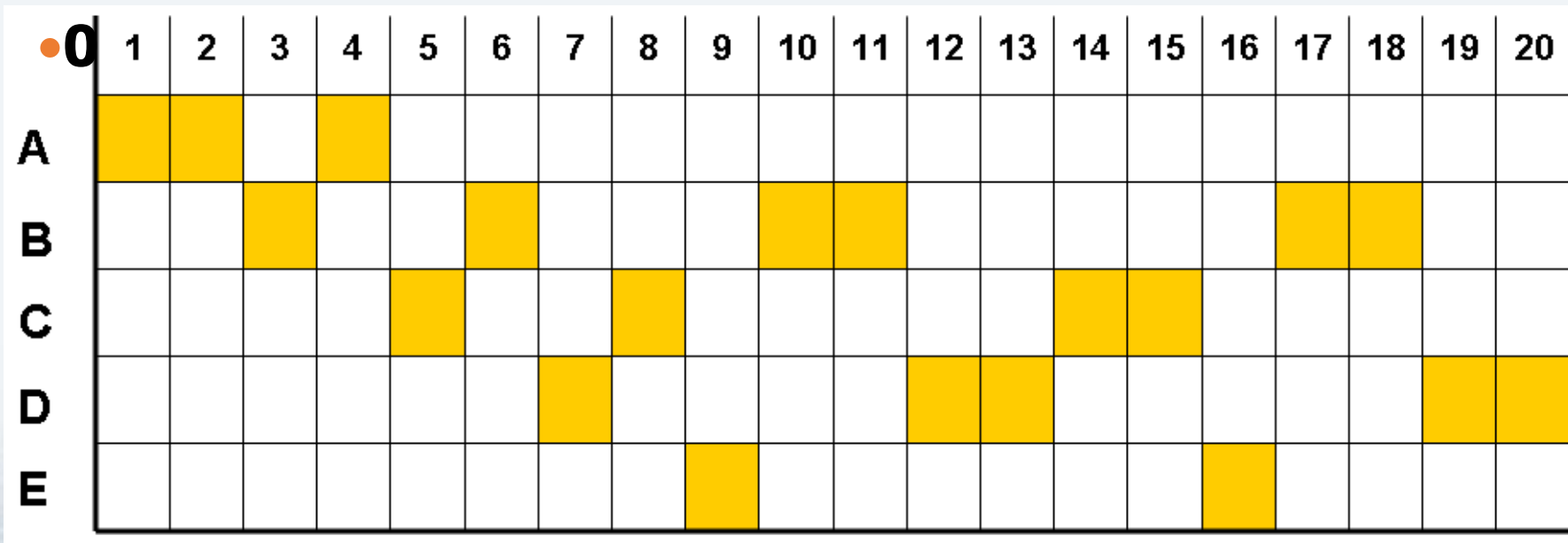
【 思考题 】

- 假设一个系统中有5个进程，它们的到达时间和服务时间如下表所示：

| 进程 | 到达时间 | 服务时间 |
|----|------|------|
| A | 0 | 3 |
| B | 2 | 6 |
| C | 4 | 4 |
| D | 6 | 5 |
| E | 8 | 2 |

【思考题】

- 忽略I/O及其他开销时间，若按FB(第*i*级队列的时间片为 2^{i-1})调度算法进行CPU调度，则运行过程如下表所示：



【 思考题 】

- 平均带权周转时间为：

| 进程 | A | B | C | D | E | 平均 |
|--------|------|------|------|------|------|------|
| 完成时间 | 4 | 18 | 15 | 20 | 16 | |
| 周转时间 | 4 | 16 | 11 | 14 | 8 | 10.6 |
| 带权周转时间 | 1.33 | 2.67 | 2.75 | 2.80 | 4.00 | 2.71 |

2.5.3 交互式系统中的调度

4. 彩票调度算法

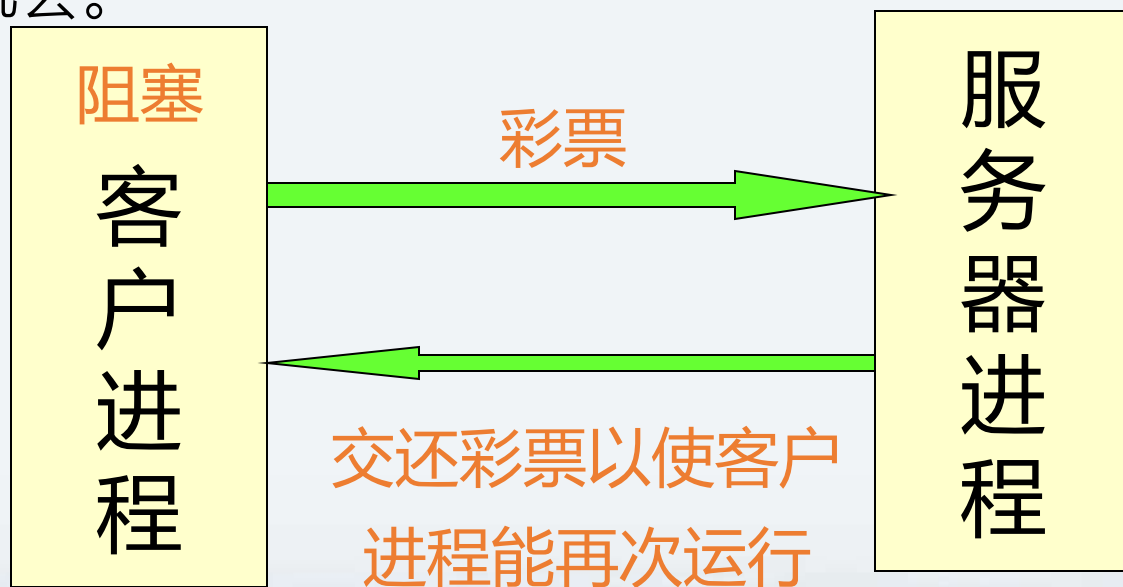
- 核心思想：为进程发放针对各种资源（如CPU时间）的彩票。调度程序随机选择一张彩票，持有该彩票的进程获得系统资源。

彩票调度算法的特点

- 平等且体现优先级：进程都是平等的，有相同的运行机会。如果某些进程需要更多的机会，可被给予更多彩票，增加其中奖机会。
- 易计算CPU的占有几率：某进程占用CPU的几率，与所持有的彩票数成正比例。该算法可实现各进程占用CPU的几率。
- 响应迅速
- 各个进程可以合作，相互交换彩票。
- 容易实现按比例分配: 如图象传输率, 10帧/s, 15帧/s, 25帧/s

彩票调度算法

- 一个客户进程向服务器发送一条消息并阻塞，阻塞后它可将其彩票交给服务器进程，以增加服务器进程下次运行的机会。



服务器进程结束后.....

2.5.4 实时系统中的调度

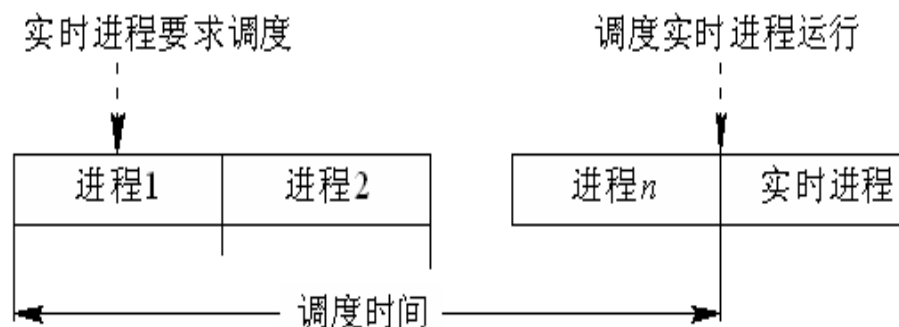
- 根据实时系统的特点，对实时系统中的调度有特殊的要求，故引入一种新的调度方式——实时调度。
- 实时系统的特点：
 - 有限等待时间（决定性）
 - 有限响应时间
 - 可靠性高
 - 系统出错处理能力强

2.5.4 实时系统中的调度

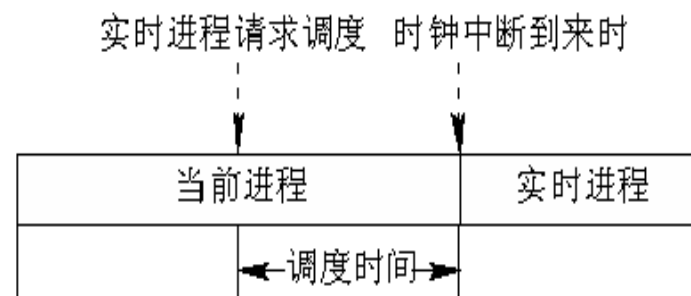
实现实时调度的基本条件：

- 提供必要的信息：如就绪时间、开始或完成截止时间、处理时间、资源要求、绝对或相对优先级（硬实时或软实时）
- 系统处理能力强
- 采用抢占式调度机制
- 具有快速切换机制：对外部中断的快速响应能力、快速的任務分派能力

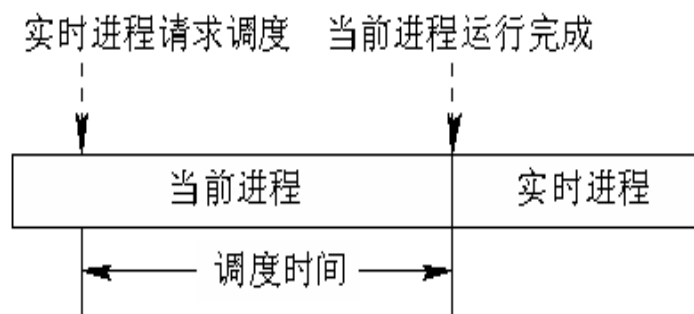
2.5.4 实时系统中的调度



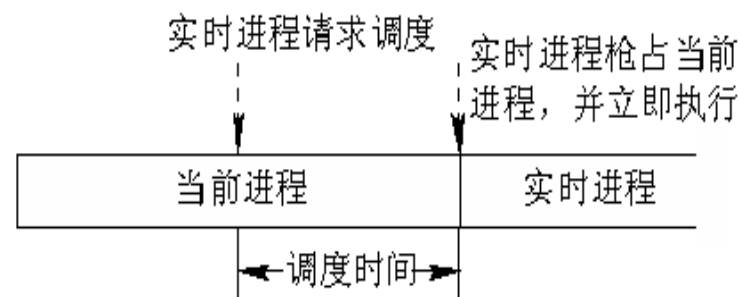
(a) 非抢占轮转调度



(c) 基于时钟中断抢占的优先权抢占调度



(b) 非抢占优先权调度



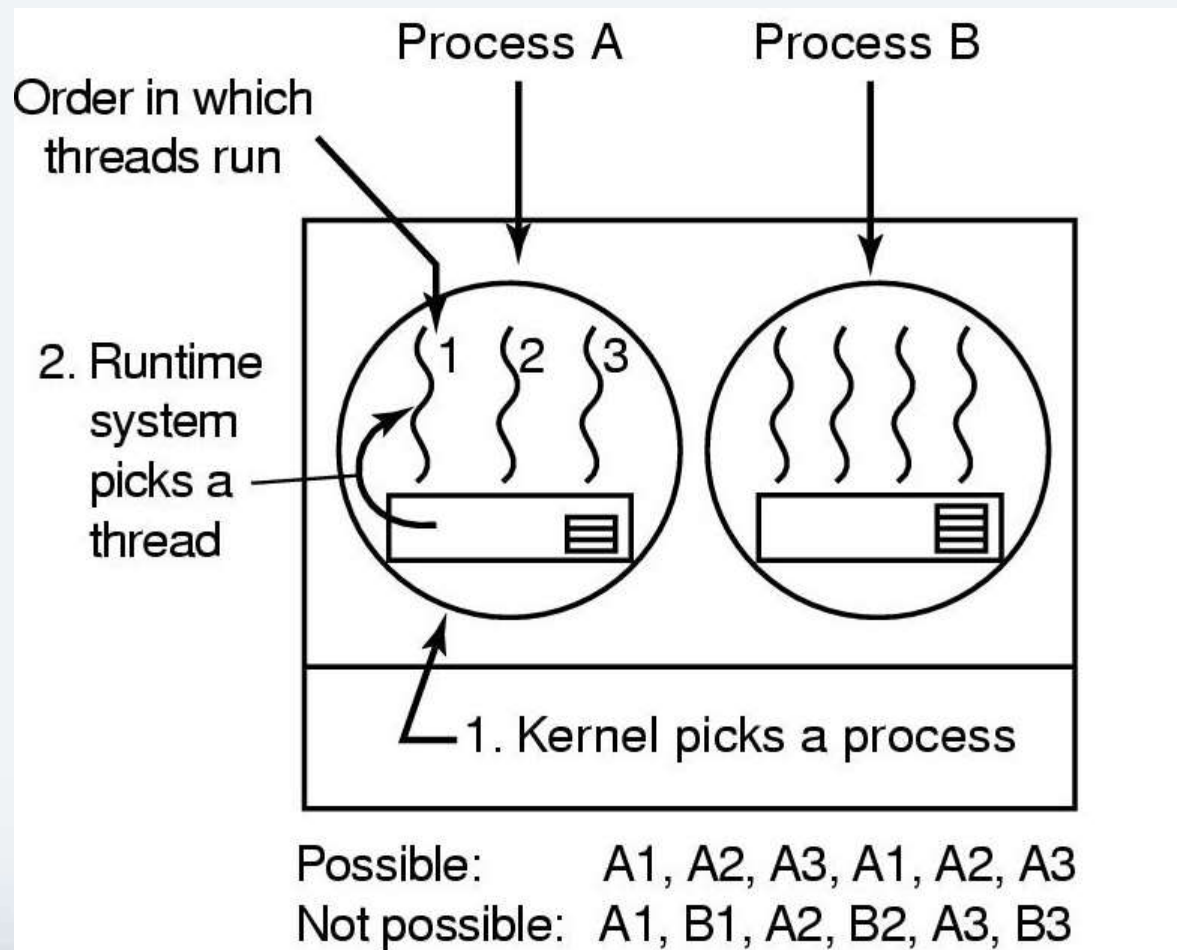
(d) 立即抢占的优先权调度

2.5.6 线程调度

用户级线程

- 内核进程调度程序选择一个进程运行
- 线程库中的线程调度程序选择一个线程运行

2.5.6 线程调度

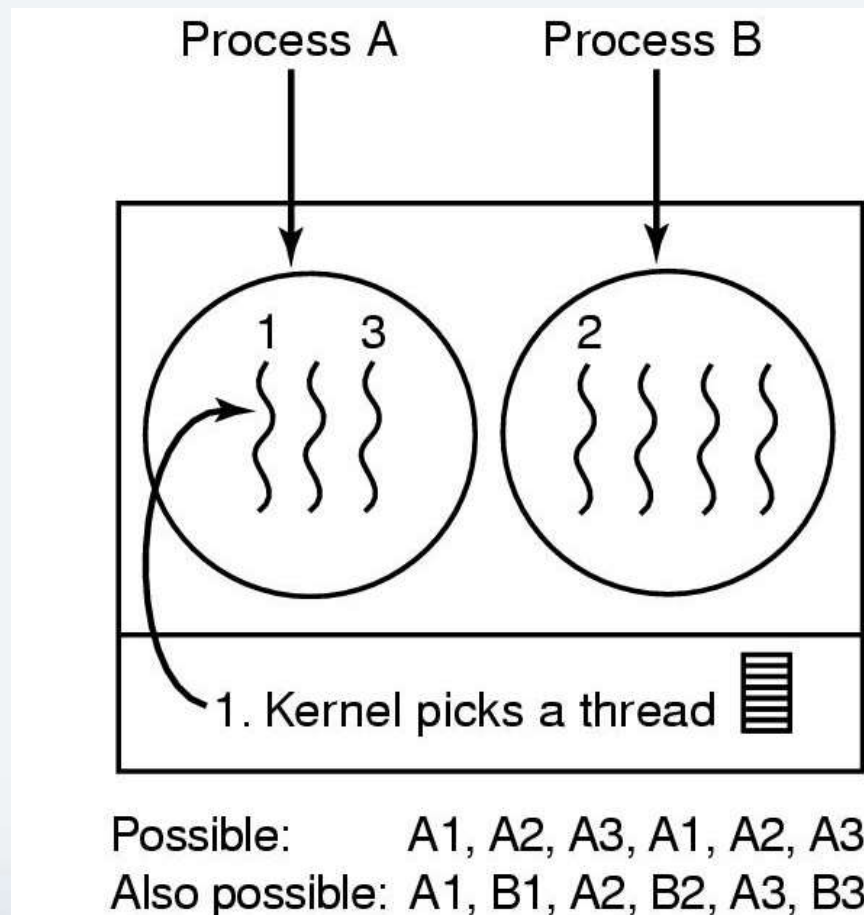


2.5.6 线程调度

内核级线程

- 线程是系统调度的单位，内核选择一个线程运行。

2.5.6 线程调度



The background is a soft, misty landscape. In the foreground, there is a calm body of water with two small, simple boats. One boat is on the left, and another is on the right. In the background, there are layers of misty mountains. A large, golden, brush-stroke-like shape is on the left side of the image. In the upper right, there are three small birds flying. The word "THANKS" is written in large, bold, black capital letters in the center-right area.

THANKS

JIANG LANFAN

2021/9/20