

第3章 进程和处理器管理

- 1 进程
- 2 进程控制
- 3 线程
- 4 进程互斥和同步
- 5 进程间通信
- 6 死锁问题
- 7 处理器调度
- 8 调度算法
- 9 windows2000/xp线程调度

程序的顺序执行和并发执行

- 程序的顺序执行

其执行过程可以描述为：

Repeat $IR \leftarrow M[pc]$

$pc \leftarrow pc + 1$

〈 Execute (instruction in IR)〉

Until CPU halt

程序的顺序执行特点：

- 程序执行的顺序性：
- 程序执行的封闭性：
- 程序结果的可再现性：

Ø 顺序执行的特征

- Ø 顺序性：按照程序结构所指定的次序
- Ø 封闭性：计算机的状态只由于该程序的控制逻辑所决定
- Ø 可再现性：初始条件相同则结果相同。

Ø 并发执行的特征

- Ø 间断(异步)性："走走停停"，一个程序可能走到中途停下来，失去原有的时序关系；
- Ø 失去封闭性：共享资源，受其他程序的控制逻辑的影响。
如：一个程序写到存储器中的数据可能被另一个程序修改，失去原有的不变特征。
- Ø 失去可再现性：失去封闭性 → 失去可再现性；外界环境在程序的两次执行期间发生变化，失去原有的可重复特征。

begin integer N;

N:=0;

cobegin

program A:

begin

L1:;

N:=N+1;

goto L1;

end;

coend;

program B:

begin

L2:;

print(N);

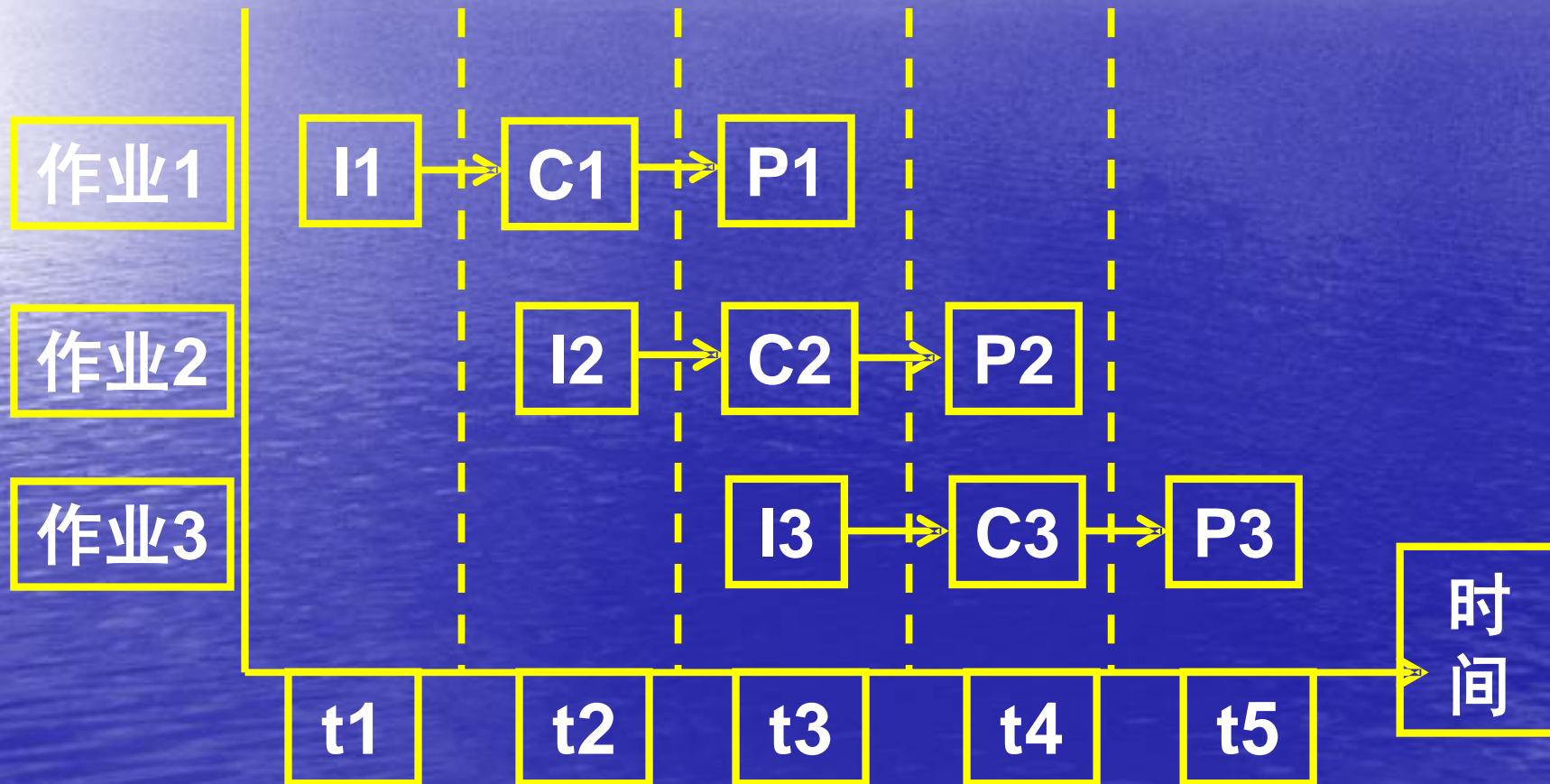
N:=0;

goto L2;

end;

程序A和B是并发执行的，它们根据运行环境的情况按照各自独立的速度运行。由于它们共享一个变量N，而N的值是由这两个程序共同确定的。所以一个程序的执行结果，与它和另一个程序的相对执行速度有密切的关系。

程序的并发执行



1966年Bernstein 提出了可以并发执行的条件:

(这里没有考虑执行速度的影响。)

- 将程序中任一语句 S_i 划分为两个变量的集合 $R(S_i)$ 和 $W(S_i)$ 。其中
- $R(S_i)=\{a_1 a_2 \dots a_m\}$ 表示 S_i 在执行期间所有引用变量的集合;
- $W(S_i)=\{b_1 b_2 \dots b_n\}$ 表示 S_i 在执行期间要改变值的全部变量;

① $R(S_1) \cap W(S_2) = \{ \phi \},$

② $W(S_1) \cap R(S_2) = \{ \phi \},$

③ $W(S_1) \cap W(S_2) = \{ \phi \}$

同时成立, 则语句 S_1 和 S_2 是可以并发执行的。

例 有四条语句

S1: $a=x+y$

S2: $b=Z+1$

S3: $c=a-b$

S4: $w=c+1$

能否并发执行？

程序的并发执行所带来的影响

- 如果并发执行的各程序段中语句或指令满足上述Bernstein 的三个条件，则认为并发执行不会对执行结果的封闭性和可再现性产生影响。
- 但在一般情况下，系统要判定并发执行的各程序段是否满足Bernstein 条件是相当困难的。

进程的定义和描述

进程的定义

一个具有一定独立功能的程序在一个数据集合上的一次动态执行过程。

- 它对应虚拟处理机、虚拟存储器和虚拟外设等资源的分配和回收；
- 引入多进程，提高了对硬件资源的利用率，但又带来额外的空间和时间开销，增加了OS 的复杂性；

进程的特征

- 动态性：进程具有动态的地址空间（数量和内容），地址空间上包括：
 - 代码（指令执行和CPU状态的改变）
 - 数据（变量的生成和赋值）
 - 系统控制信息（进程控制块的生成和删除）
- 独立性：各进程的地址空间相互独立，除非采用进程间通信手段；
- 并发性、异步性："虚拟"
- 结构化：代码段、数据段和核心段（在地址空间中）；程序文件中通常也划分了代码段和数据段，而核心段通常就是OS核心（由各个进程共享，包括各进程的PCB）

Windows 2000/XP进程的组成如下：

- 一个可执行程序，它定义了初始代码和数据。
- 一个专用的“虚拟地址空间”，它是进程能够使用的一组虚拟内存地址。
- 系统资源，例如信号量、通信端口和文件，它们是在程序执行过程中，当线程打开这些资源时操作系统分配给进程的。
- 称作“进程ID”的唯一标识符（在内部被称作“客户ID”）。
- 至少一个执行线程。

进程与程序的区别

- 进程是动态的，程序是静态的：程序是有序代码的集合；进程是程序的执行。通常进程不可在计算机之间迁移；而程序通常对应着文件、静态和可以复制。
- 进程是暂时的，程序的永久的：进程是一个状态变化的过程，程序可长久保存。
- 进程与程序的组成不同：进程的组成包括程序、数据和进程控制块（即进程状态信息）。
- 进程与程序的对应关系：通过多次执行，一个程序可对应多个进程；通过调用关系，一个进程可包括多个程序。

进程控制块

进程控制块是由OS维护的用来记录进程相关信息的一块内存。

- 在创建一个进程时，应首先创建其 PCB，然后才能根据PCB 中信息对进程实施有效的管理和控制。
- PCB包含一个进程的描述信息、控制信息及资源信息，有些系统中还有进程调度等待所使用的现场保护区

进程控制块的内容

(1) 描述信息

- ① 进程名或进程标识号
- ② 用户名或用户标识号
- ③ 家族关系

(2) 控制信息

- ① 进程当前状态

进程在活动期间可分为就绪态、执行态和等待状态。

- ② 进程优先级

进程优先级是选取进程占有处理机的重要依据。与进程优先级有关的PCB表项有：

- a. 占有CPU时间；
- b. 进程优先级偏移；
- c. 占据内存时间，等。

③ 程序开始地址

④ 各种计时信息

给出进程占有和利用资源的有关情况。

⑤ 通信信息

通信信息用来说明该进程在执行过程中与别的进程所发生的信息交换情况。

(3) 资源管理信息

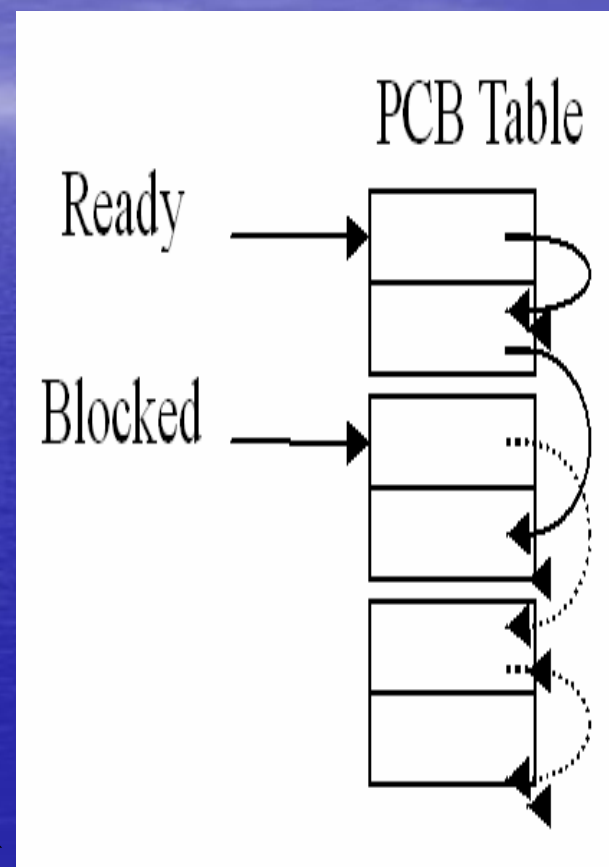
PCB 中包含最多的是资源管理信息，包括有关存储器的信息、使用输入输出设备的信息、有关文件系统的信息等。这些信息有：

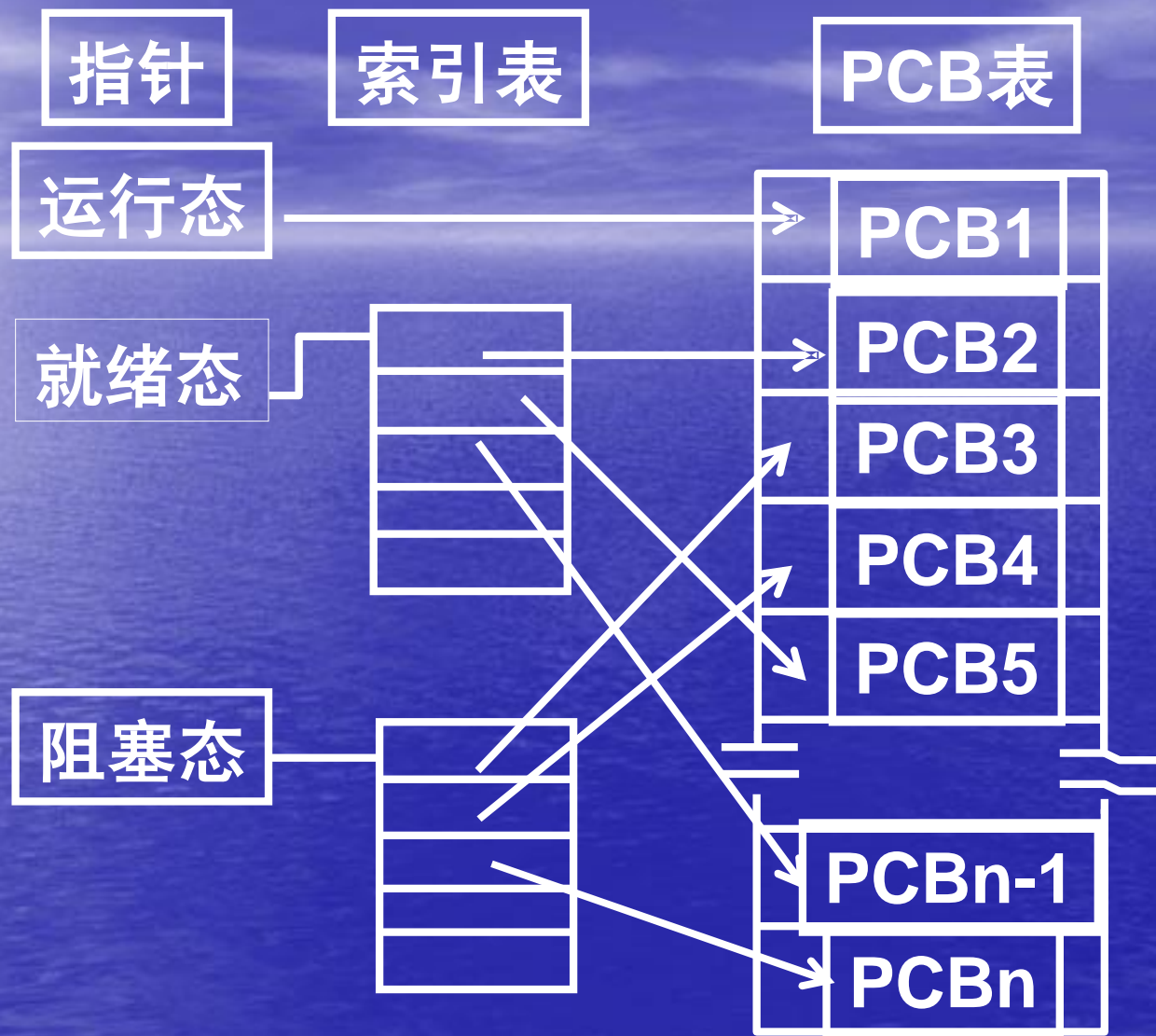
① 占用内存大小及其管理用数据结构指针，例如后述内存管理中所用到的进程页表指针等。

- ② 在某些复杂系统中，还有对换或覆盖用的有关信息，如对换程序段长度，对换外存地址等。这些信息在进程申请、释放内存中使用。
- ③ 共享程序段大小及起始地址。
- ④ 输入输出设备的设备号，所要传送的数据长度、缓冲区地址、缓冲区长度及所用设备的有关数据结构指针等。这些信息在进程申请释放设备进行数据传输中使用。
- ⑤ 指向文件系统的指针及有关标识等。进程可使用这些信息对文件系统进行操作。

PCB的组织方式

- 链表：同一状态的进程其PCB成一链表，多个状态对应多个不同的链表
 - 各状态的进程形成不同的链表：就绪链表、阻塞链表
- 索引表：同一状态的进程归入一个index表（由index指向PCB），多个状态对应多个不同的index表
 - 各状态的进程形成不同的索引表：就绪索引表、阻塞索引表

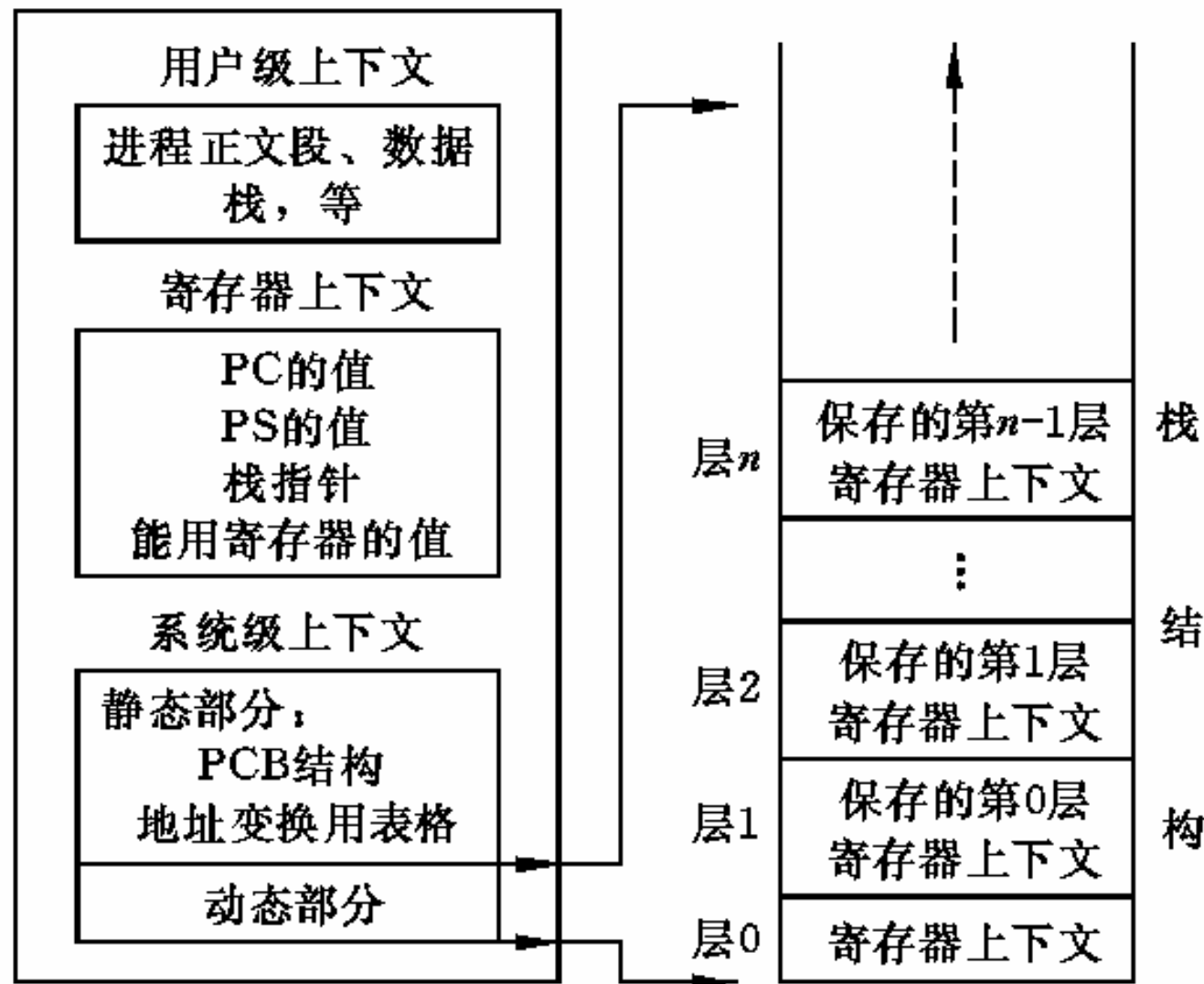




进程上下文

进程上下文是对进程执行活动全过程的静态描述。进程上下文由进程的用户地址空间内容、硬件寄存器内容及与该进程相关的核心数据结构组成。

- 用户级上下文：进程的用户地址空间（包括用户栈各层次），包括用户正文段、用户数据段和用户栈；
- 寄存器级上下文：程序寄存器、处理机状态寄存器、栈指针、通用寄存器的值；
- 系统级上下文：
 - 静态部分（PCB和资源表格）
 - 动态部分：核心栈（核心过程的栈结构，不同进程在调用相同核心过程时有不同核心栈）



核心态和用户态

- 用户态时不可直接访问受保护的OS代码；
- 核心态时执行OS代码，可以访问全部进程空间。

进程的状态转换

两状态进程模型

五状态进程模型

挂起进程模型

五状态进程模型

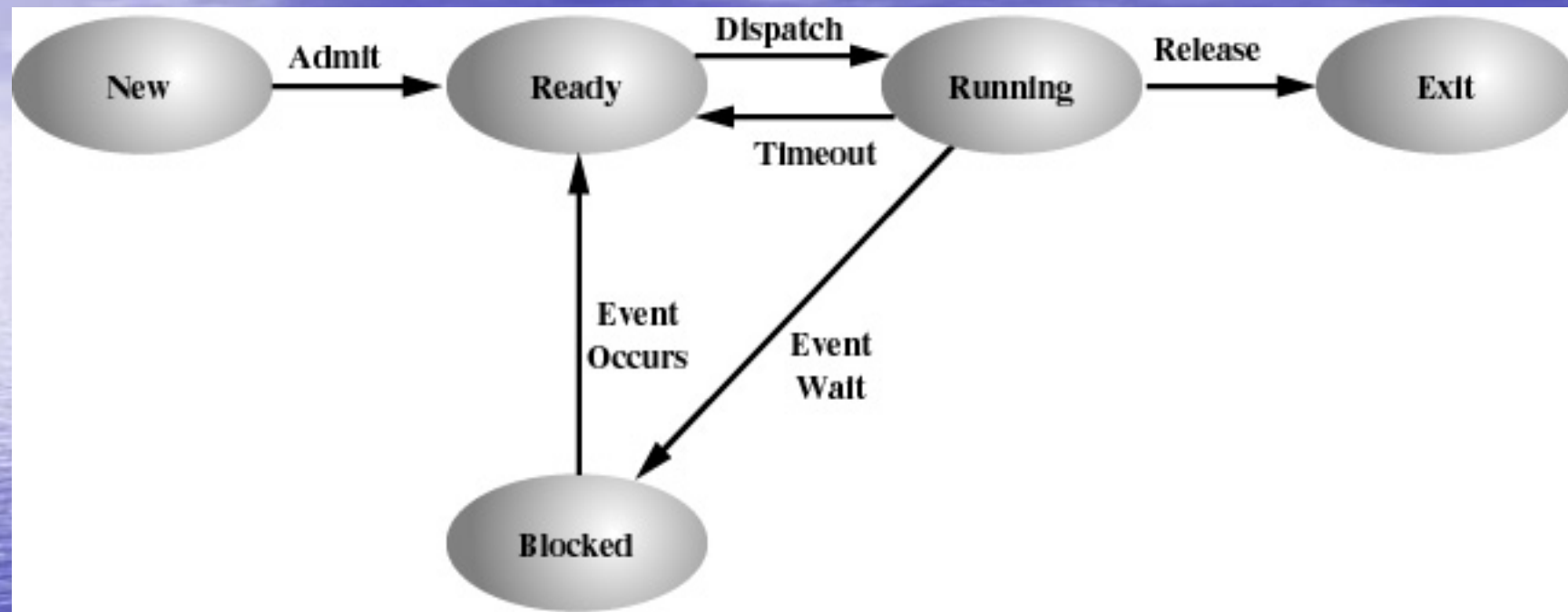
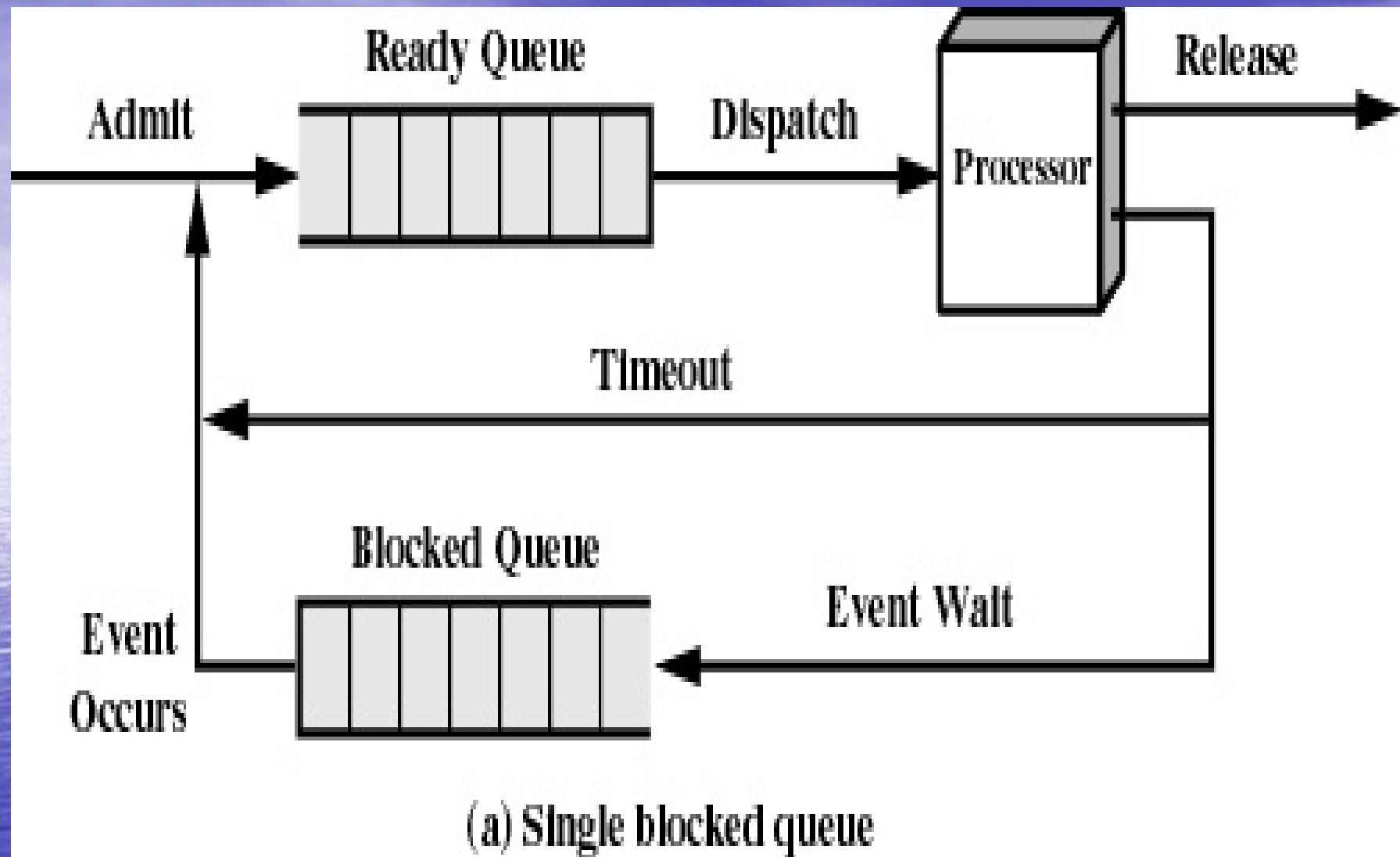
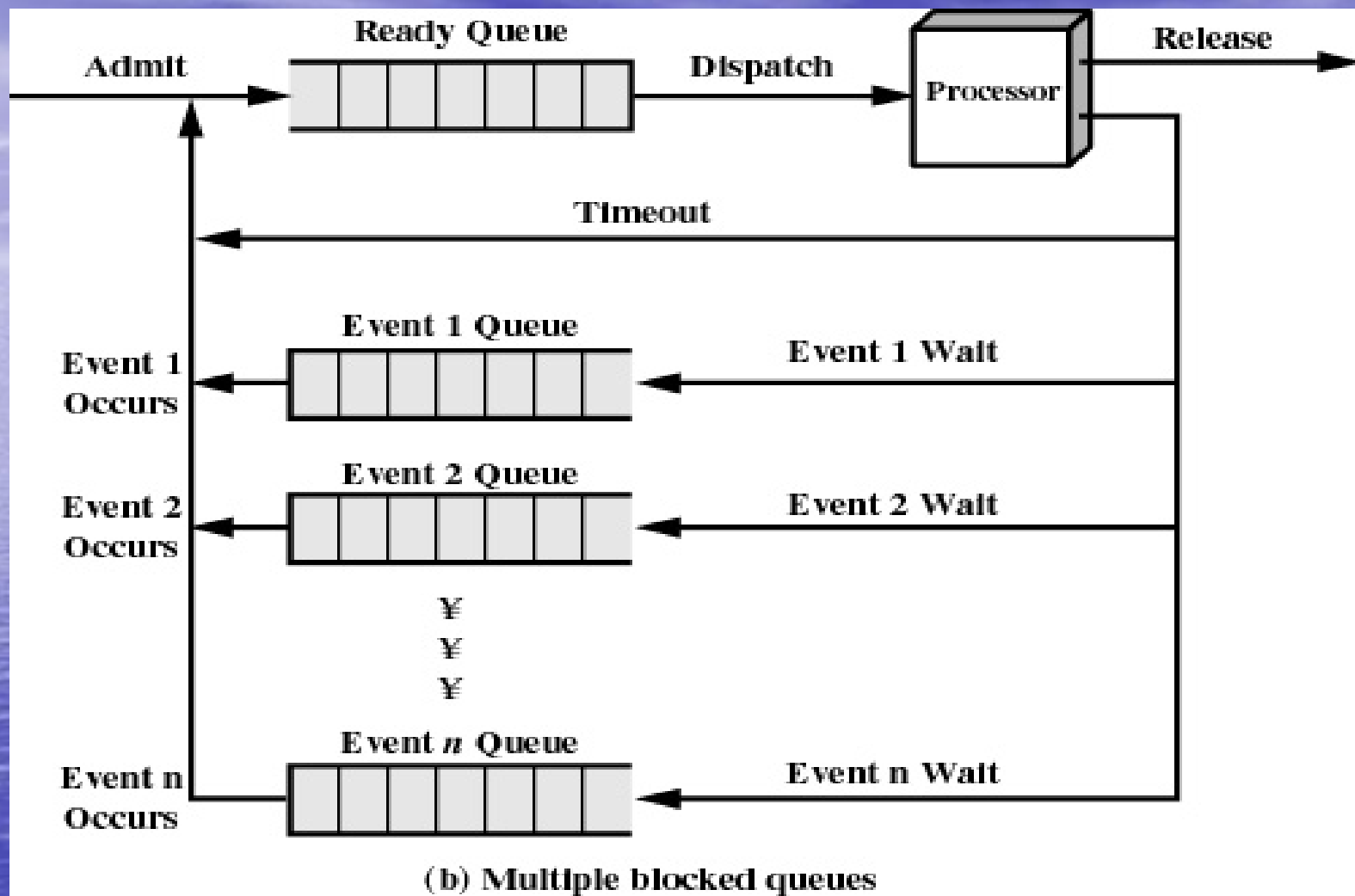


Figure 3.5 Five-State Process Model

五状态进程模型(状态变迁)



五状态进程模型(单队列结构)



五状态进程模型(多队列结构)

五状态模型

- 运行(Running): 该进程正在被执行。在本章中, 我们假设计算机只有一个处理器, 因此一次最多只有一个进程处于这个状态。
- 就绪(Ready): 进程做好了准备, 只要有机会就开始执行。
- 阻塞(Blocked): 进程在某些事件发生前不能执行, 如I / O操作完成。
- 新建(New): 刚刚创建的进程, 操作系统还没有把它加入到可执行进程组中, 通常是还没有加载到主存中的新进程。
- 退出(Exit): 操作系统从可执行进程组中释放出的进程, 或者是因为它自身停止个, 或者是因为某种原因被取消。

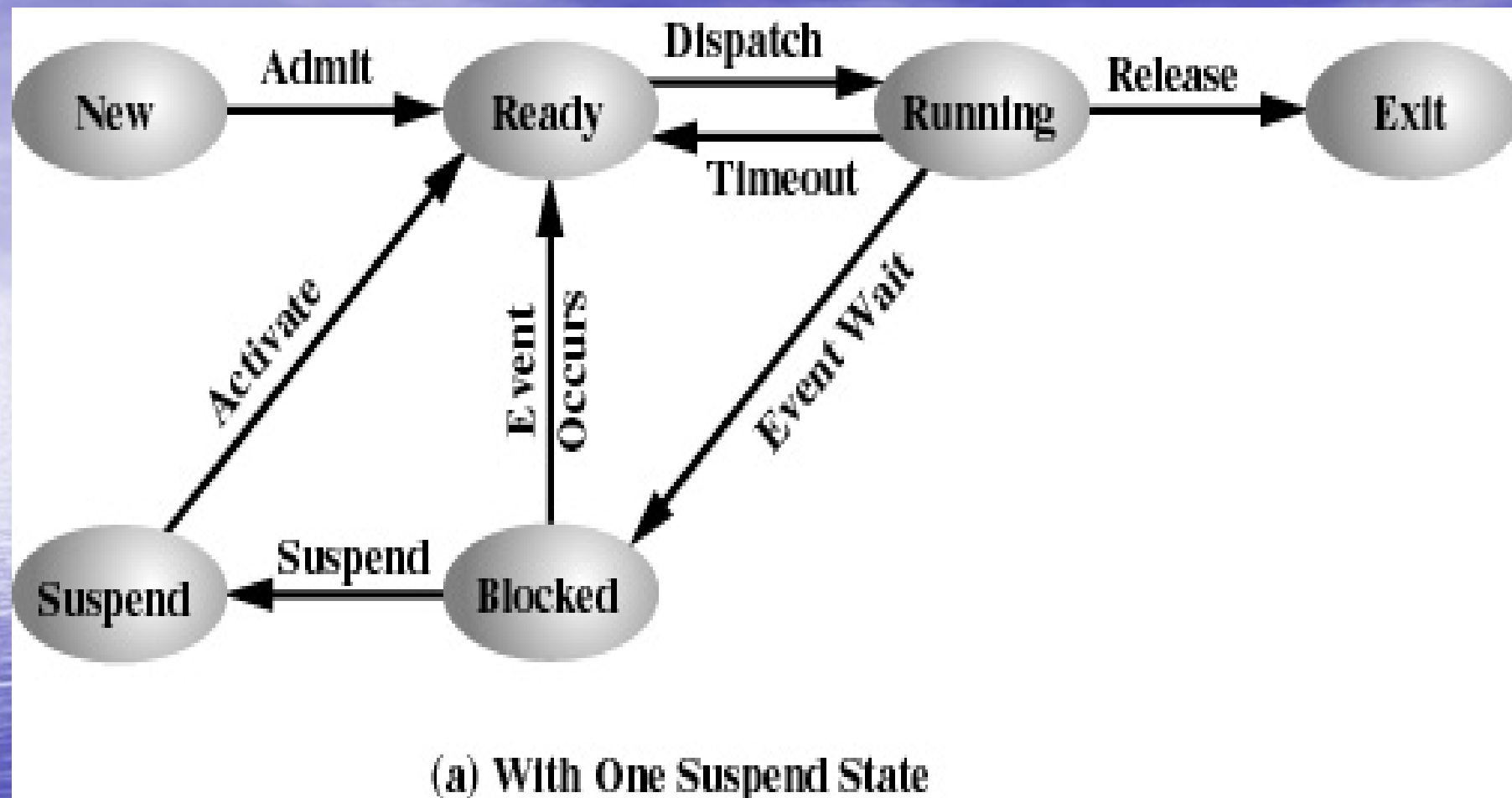
挂起进程模型

Ø这个问题的出现是由于进程优先级的引入，一些低优先级进程可能等待较长时间，从而被对换至外存。这样做的目的是：

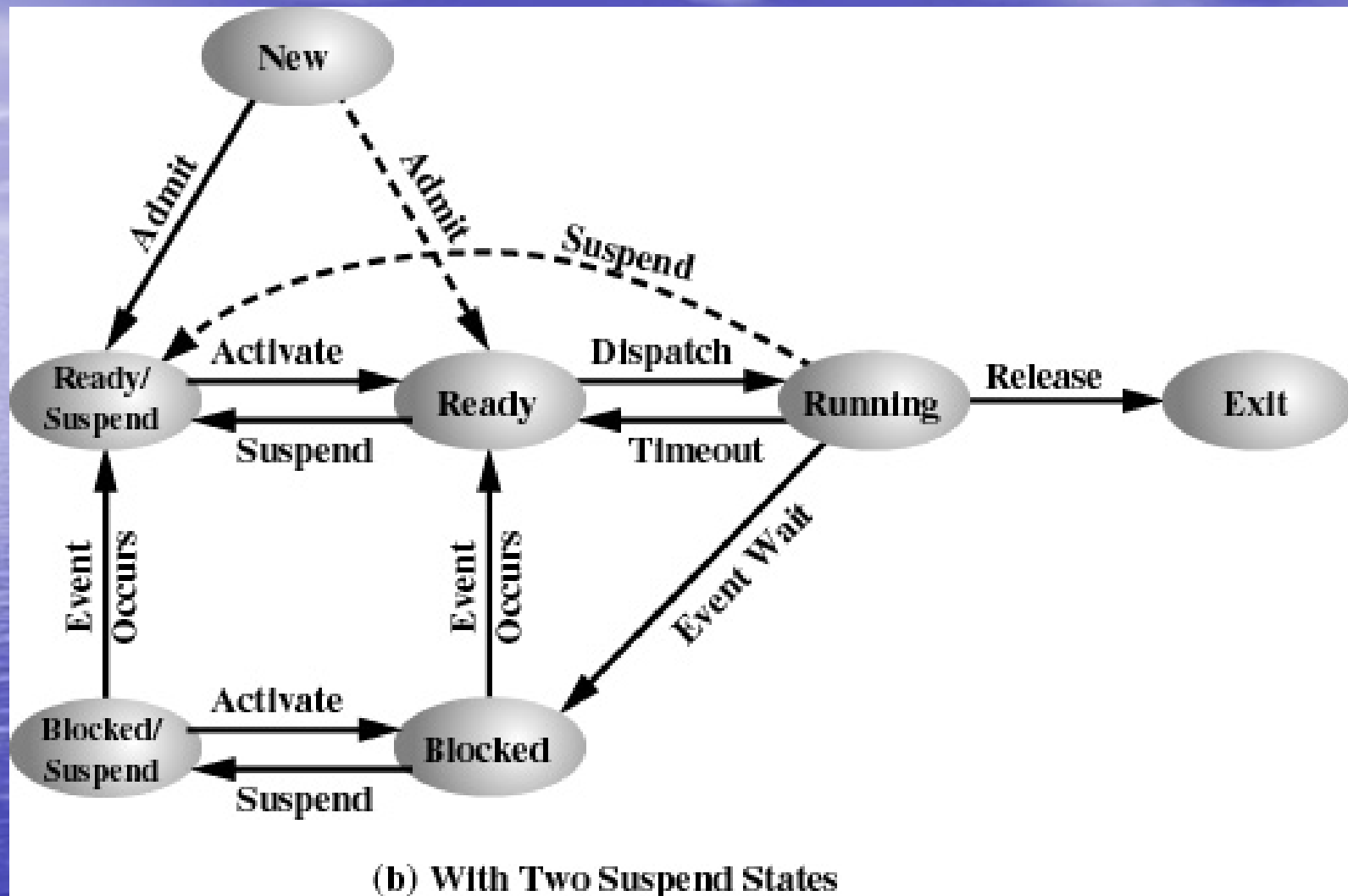
Ø提高处理机效率：就绪进程表为空时，要提交新进程，以提高处理机效率；

Ø为运行进程提供足够内存：资源紧张时，暂停某些进程，如：CPU繁忙（或实时任务执行），内存紧张

Ø用于调试：在调试时，挂起被调试进程（从而对其地址空间进行读写）



单挂起进程模型



双挂起进程模型

进程状态

- 就绪状态：进程在内存且可立即进入运行状态；
- 阻塞状态：进程在内存并等待某事件的出现；
- 阻塞挂起状态：进程在外存并等待某事件的出现；
- 就绪挂起状态：进程在外存，但只要进入内存，即可运行；

进程的创建

	产生新进程	不产生新进程
复制现有进程的上下文	<code>fork</code> (新进程的系统上下文会有不同)	
加载程序	<code>spawn</code> (等待子进程完成,)	<code>exec</code> (加载新程序并覆盖自身)

- 继承：子进程可以从父进程中继承用户标识符、环境变量、打开文件、文件系统的当前目录、控制终端、已经连接的共享存储区、信号处理例程入口表等
- 不被继承：进程标识符，父进程标识符
- `spawn` 创建并执行一个新进程；新进程与父进程的关系可有多种：覆盖、并发、父进程阻塞、后台等。

进程的阻塞和唤醒

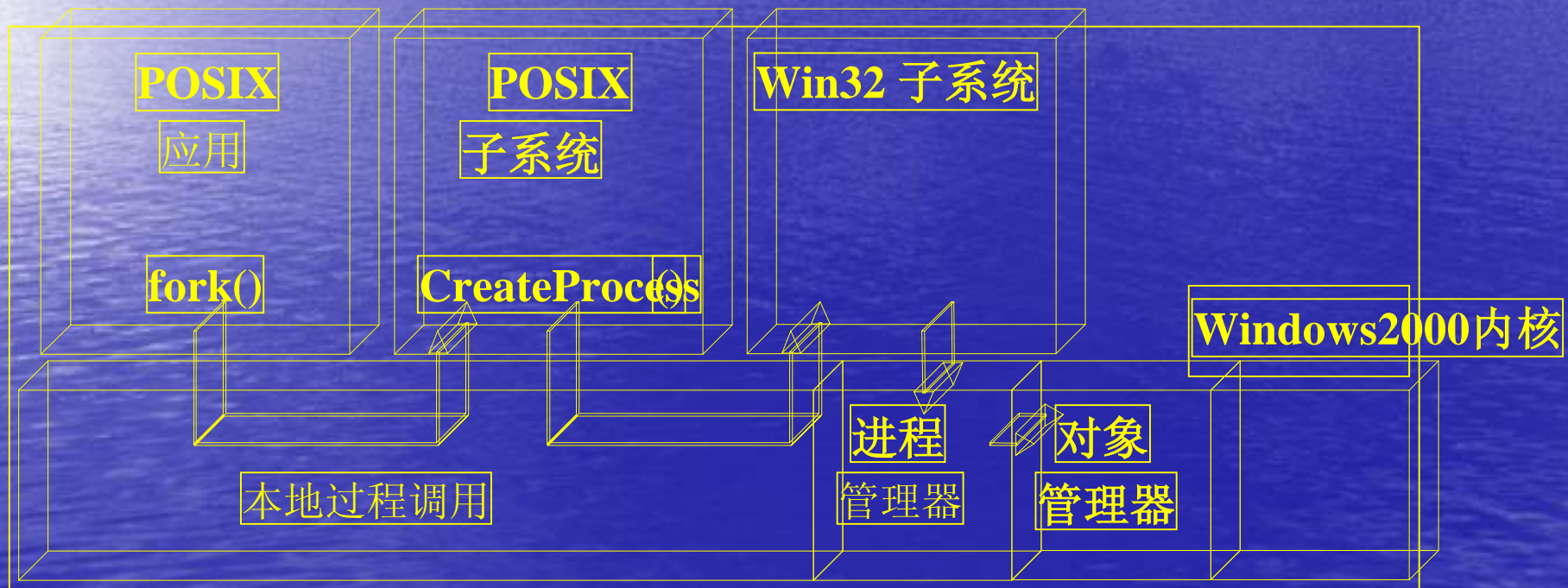
- 1) `sleep`将在指定的时间内挂起本进程。其调用格式为：“`unsigned sleep()`”，返回值为实际的挂起时间。
- 2) `pause`挂起本进程以等待信号，接收到信号后恢复执行。当接收到终止进程信号时，该调用不再返回。其调用格式为“`int pause(void)`”。
- 3) `wait`挂起本进程以等待子进程的结束，子进程结束时返回。当父进程创建多个子进程且已有子进程退出时，父进程中`wait`函数在第一个子进程结束时返回。
- 4) `kill`可发送信号`sig`到某个或一组进程`pid`。其调用格式为：“`int kill(pid_t pid, int sig)`” 信号的定义在文件 `signal. h`中。

Windows 进程管理

- Windows的进程和线程作为对象，以句柄(handle)来引用。相应地有控制对象的服务。

windows的进程管理

- 对Windows核心而言，进程之间没有任何关系（包括父子关系）。那么，如何表达UNIX进程之间的父子关系（以及其他关系）由POSIX子系统来建立和维护



线程的概念

Ø 线程(Thread)是一个动态的对象，它是处理器调度的基本单位。

Ø 线程的优点：减小并发执行的时间和空间开销，在系统中建立更多的线程来提高并发程度。

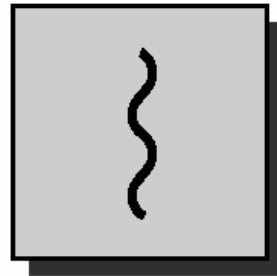
- 创建速度快（在已有进程内）
- 终止所用时间少
- 切换时间少（保存和恢复工作量小）
- 通信效率高（在同一进程内，无需调用内核，可利用共享的存储空间）

线程包括下列基本组件

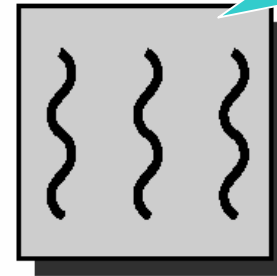
- 表示处理器状态的一组易失寄存器的内容。
- 两个由线程使用的堆栈，一个在核心态时使用，另一个在用户态时使用。
- 供子系统、运行时库和DLL使用的专用存储区。
- 称作“线程ID”的唯一标识符（在内部也被称作“客户ID”，进程ID和线程ID产生自不同的命名空间，因此不会重叠）。

Java 运行环境

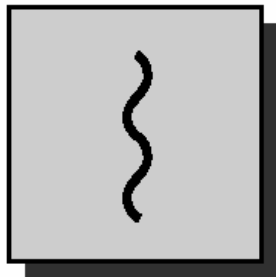
MS-DOS



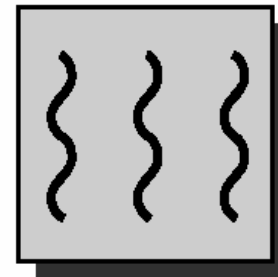
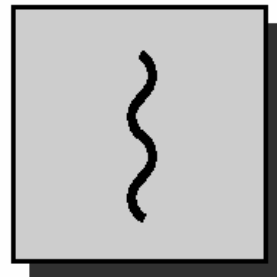
one process
one thread



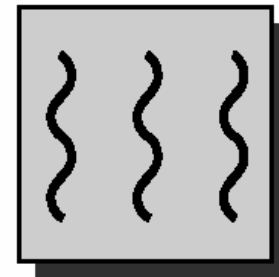
one process
multiple threads



multiple processes
one thread per process



multiple processes
multiple threads per process



UNIX

W2K, Solaris, Linux, Mach, OS/2

进程与线程的关系

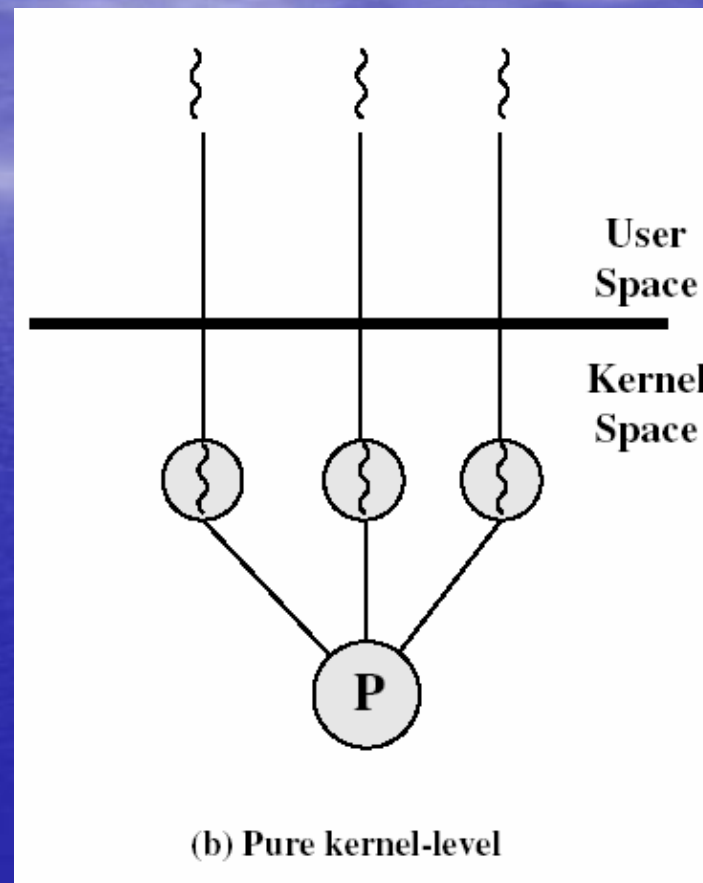
线程的实现

- 用户级线程 (User-level Threads, ULT)
- 内核级线程 (Kernel-level Threads, KLT)
- 组合的方法 (Combined Approaches)

内核线程(kernel-level thread)

依赖于OS核心，由内核的内部需求进行创建和撤销，用来执行一个指定的函数。Windows 2000/xp和Winnt支持内核线程；

- 内核维护进程和线程的上下文信息；
- 线程切换由内核完成；
- 一个线程发起系统调用而阻塞，不会影响其他线程的运行。
- 时间片分配给线程，所以多线程的进程获得更多CPU时间。

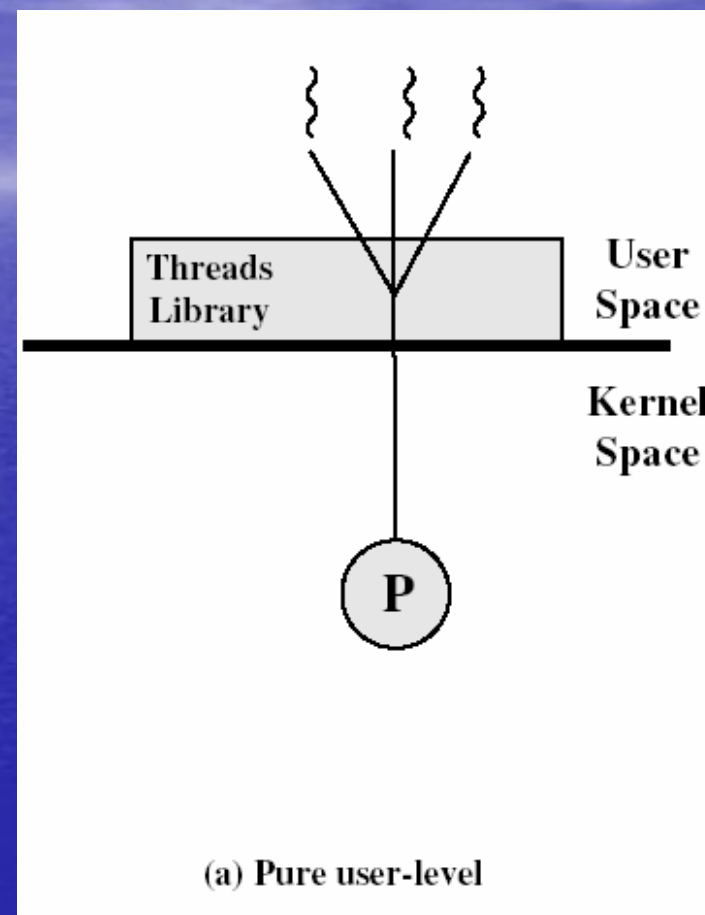


用户线程(user-level thread)

不依赖于OS核心，应用进程利用线程库提供创建、同步、调度和管理线程的函数来控制用户线程。

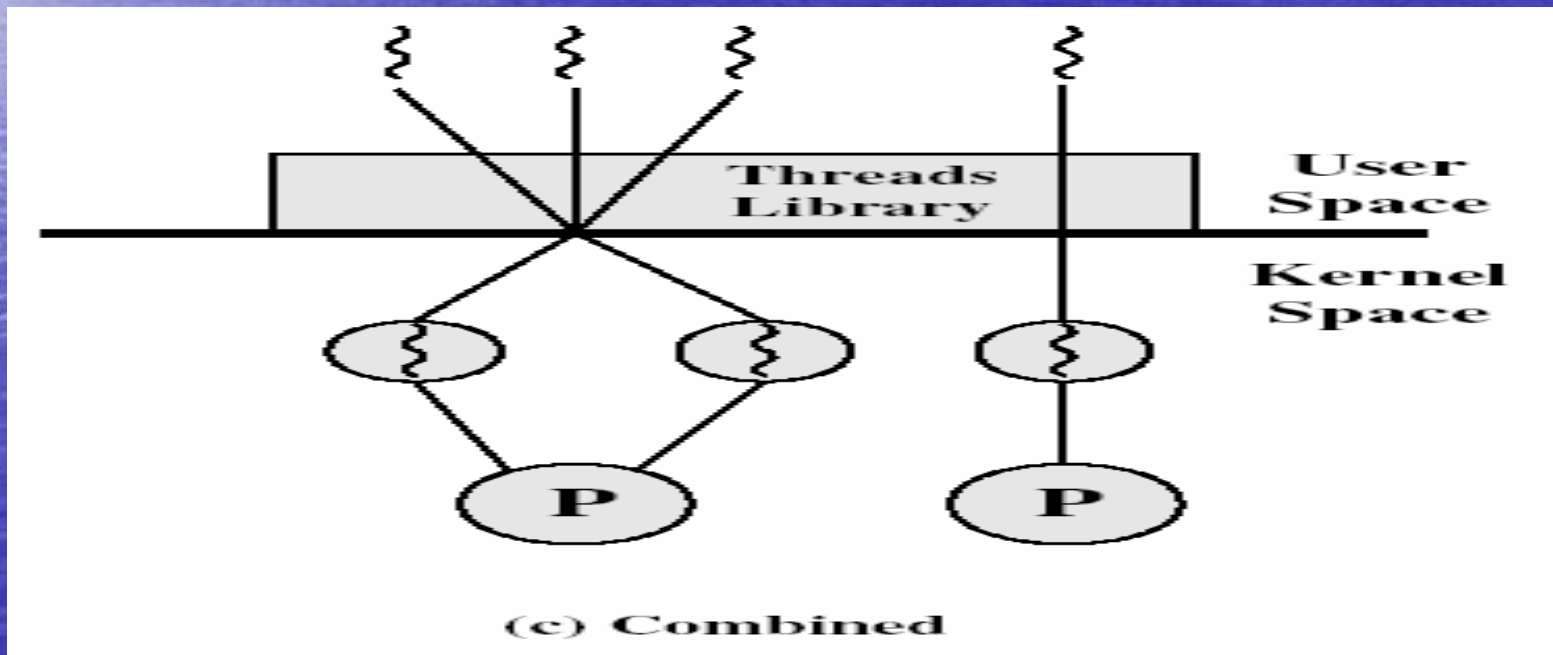
如：数据库系统informix，图形处理Aldus PageMaker。

- 用户线程的维护由应用进程完成；
- 内核不了解用户线程的存在；
- 用户线程切换不需要内核特权；
- 用户线程调度算法可针对应用优化；



轻权进程(LightWeight Process)

它是内核支持的用户线程。一个进程可有一个或多个轻权进程，每个轻权进程由一个单独的内核线程来支持。



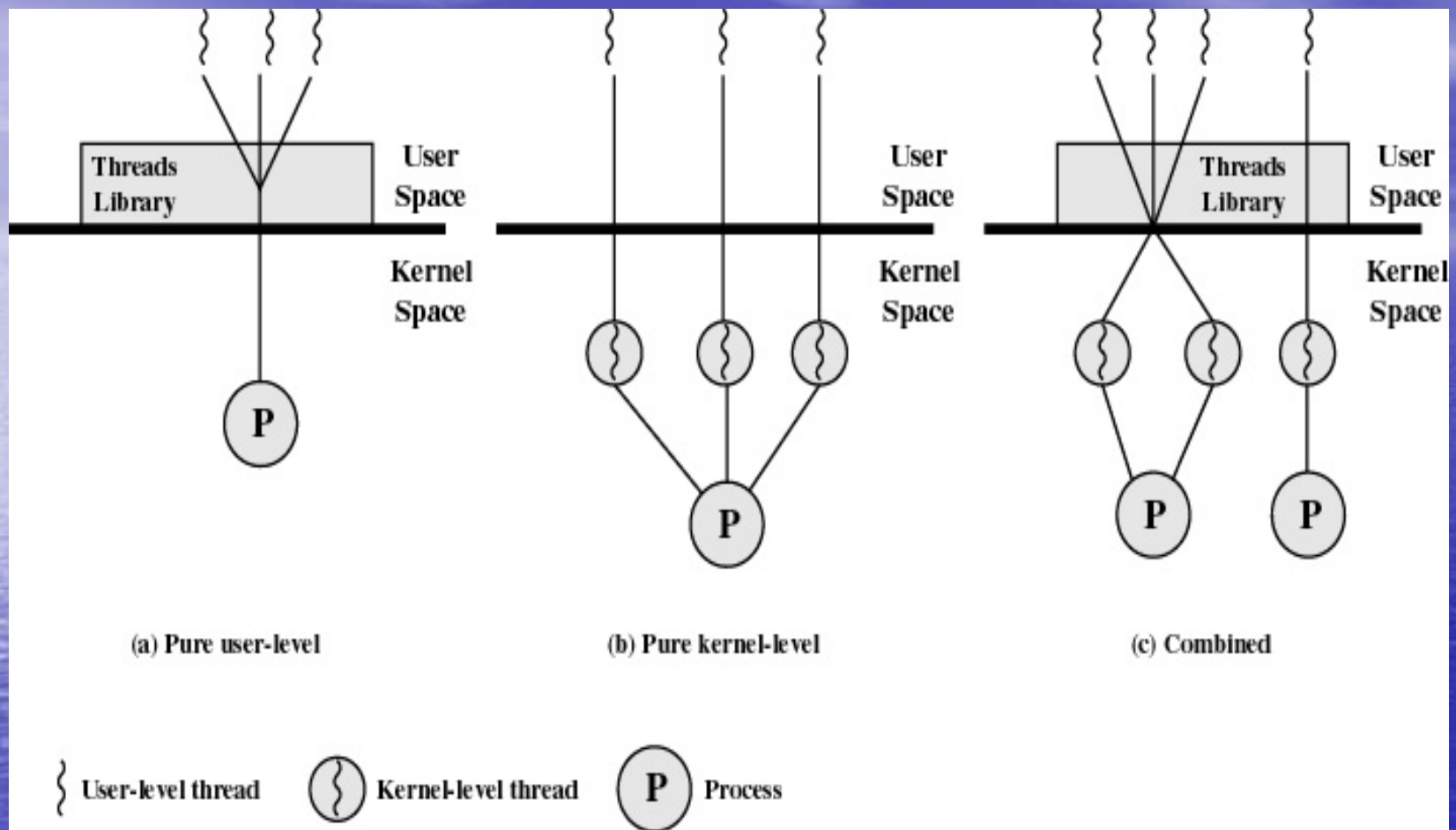
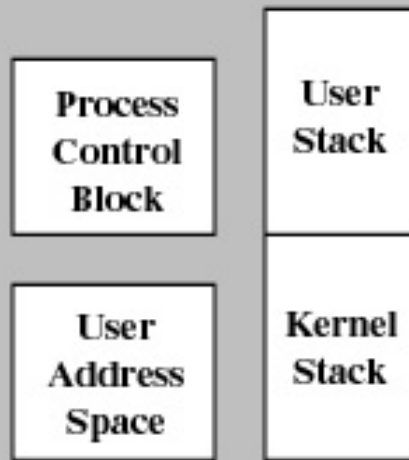


Figure 4.6 User-Level and Kernel-Level Threads

进程和线程的比较

- ①地址空间资源：不同进程的地址空间是相互独立的，而同一进程的各线程共享同一地址空间。一个进程中的线程在另一个进程中是不可见的。
- ②通信关系：进程间通信必须使用操作系统提供的进程间通信机制，而同一进程中的各线程间可以通过直接读写进程数据段(如全局变量)来进行通信。
- ③调度切换：同一进程中的线程上下文切换比进程上下文切换要快得多。

Single-Threaded Process Model



Multithreaded Process Model

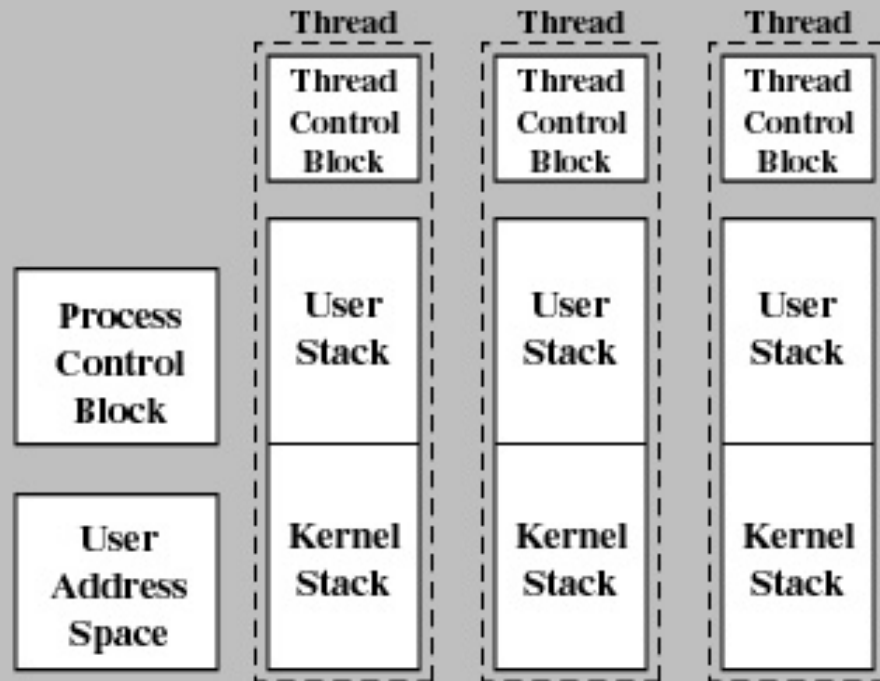
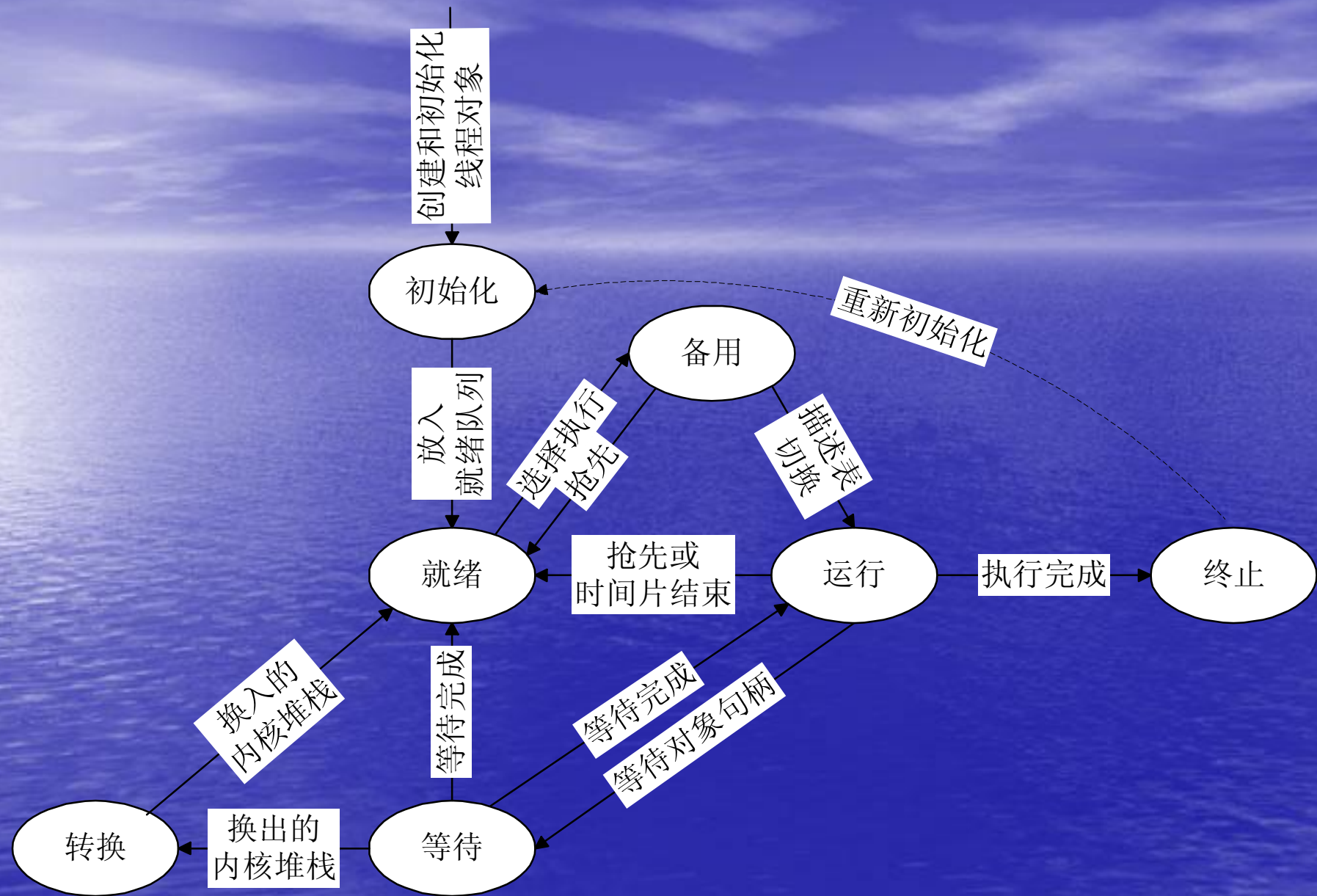


Figure 4.2 Single Threaded and Multithreaded Process Models

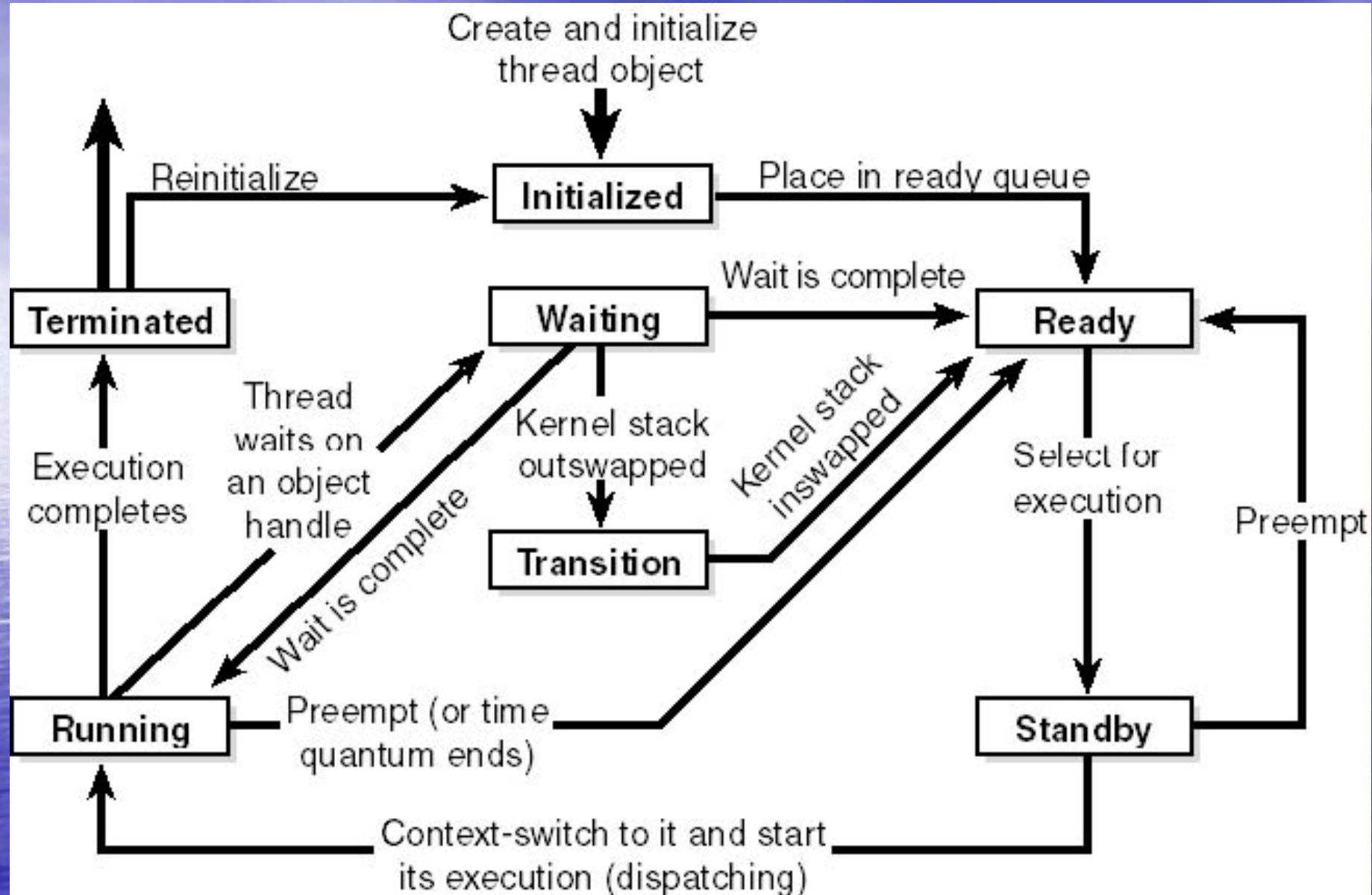
线程切换和进程切换

线程状态

- 就绪状态(Ready): 进程已获得除处理机外的所需资源, 等待执行。
- 备用状态(Standby): 特定处理器的执行对象, 系统中每个处理器上只能有一个处于备用状态的线程。
- 运行状态(Running): 完成描述表切换, 线程进入运行状态, 直到内核抢先、时间片用完、线程终止或进行等待状态。
- 等待状态(Waiting): 线程等待对象句柄, 以同步它的执行。等待结束时, 根据优先级进入运行、就绪状态。
- 转换状态(Transition): 线程在准备执行而其内核堆栈处于外存时, 线程进入转换状态; 当其内核堆栈调回内存, 线程进入就绪状态。
- 终止状态(Terminated): 线程执行完就进入终止状态; 如执行体有一指向线程对象的指针, 可将线程对象重新初始化, 并再次使用。
- 初始化状态(Initialized): 线程创建过程中的线程状态;



Windows2000/xp 的线程状态



Windows 2000线程状态

进程互斥和同步

互斥算法

信号量

经典进程同步问题

管程

Windows进程互斥和同步

互斥算法

临界资源

临界区的访问过程

同步机制应遵循的准则

进程互斥的软件方法

进程互斥的硬件方法

临界资源

多道程序环境—>进程之间存在资源共享，进程的运行时间受影响硬件或软件（如外设、共享代码段、共享数据结构），多个进程在对其进行访问时（关键是进行写入或修改），必须互斥地进行——有些共享资源可以同时访问，如只读数据

- 进程间资源访问冲突
 - 共享变量的修改冲突
 - 操作顺序冲突
- 进程间的制约关系
 - 间接制约：进行竞争——独占分配到的部分或全部共享资源，“互斥”
 - 直接制约：进行协作——等待来自其他进程的信息，“同步”

进程的交互关系：可以按照相互感知的程度来分类

相互感知的程度	交互关系	一个进程对其他进程的影响	潜在的控制问题
相互不感知 完全不了解其它进程的存在	竞争 (competition)	进程的操作对其他进程的结果无影响	互斥，死锁(可释放的资源)，饥饿
间接感知 双方都与第三方交互，如共享资源	通过共享进行协作	进程的结果依赖于其他进程获得的信息	互斥，死锁（可释放的资源），饥饿，数据一致性
直接感知 双方直接交互，如通信	通过通信进行协作	进程的结果依赖于从其他进程获得的信息	死锁，饥饿

互斥，指多个进程不能同时使用同一个资源；
 死锁，指多个进程互不相让，都得不到足够的资源；
 饥饿，指一个进程一直得不到资源（其他进程可能轮流占用资源）

临界区的访问过程

进入区

临界区

推出区

剩余区

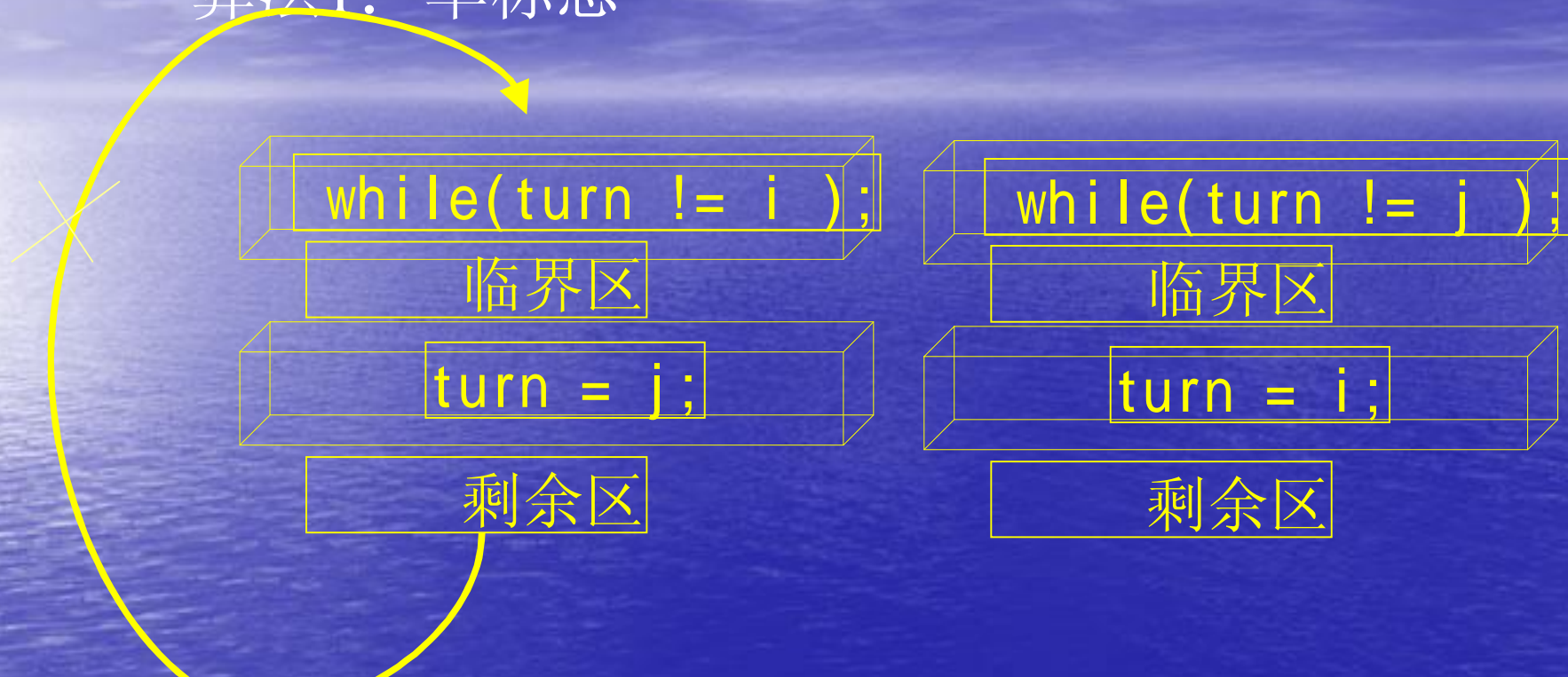
- 临界区(critical section): 进程中访问临界资源的一段代码。
- 进入区(entry section): 在进入临界区之前, 检查可否进入临界区的一段代码。如果可以进入临界区, 通常设置相应"正在访问临界区"标志
- 退出区(exit section): 用于将"正在访问临界区"标志清除。
- 剩余区(remainder section): 代码中的其余部分。

同步机制应遵循的准则

- 空闲则入：其他进程均不处于临界区；
- 忙则等待：已有进程处于其临界区；
- 有限等待：等待进入临界区的进程不能“死等”；
- 让权等待：不能进入临界区的进程，应释放CPU（如转换到阻塞状态）

进程互斥的软件方法

算法1：单标志



- 缺点：强制轮流进入临界区，没有考虑进程的实际需要。容易造成资源利用不充分：在 P_i 出让临界区之后， P_j 使用临界区之前， P_i 不可能再次使用临界区；

算法2：双标志、先检查

while (flag[1]); <a>

flag[0]=TRUE;

临界区

flag[0] = FALSE;

剩余区

while (flag[0]) <a>

flag[1] =TRUE;

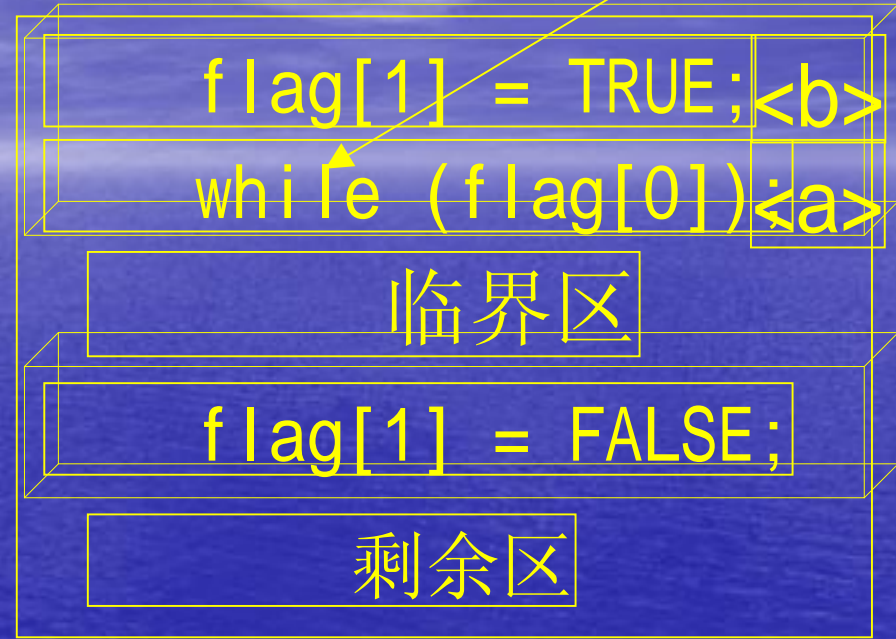
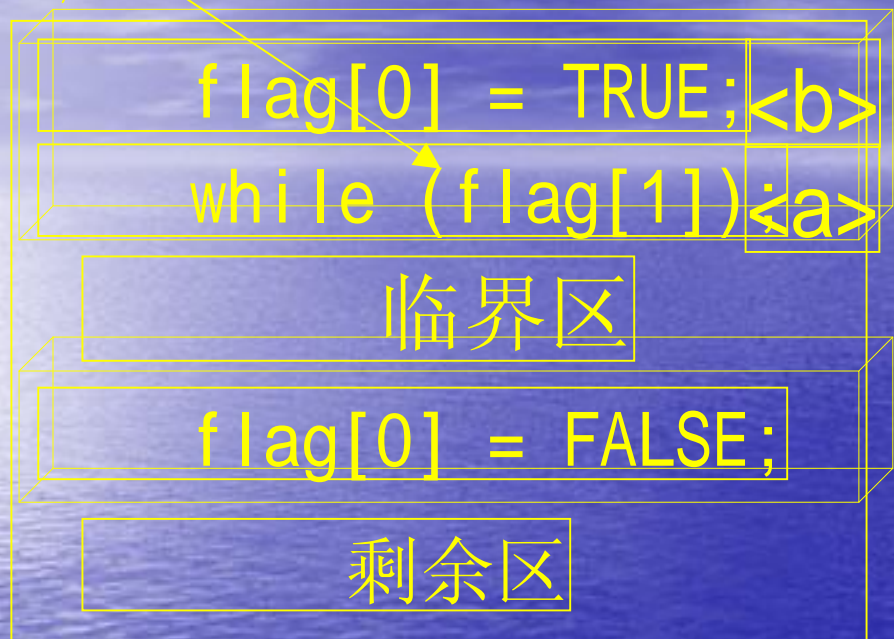
临界区

flag[1] = FALSE;

剩余区

- 优点：不用交替进入，可连续使用；
- 缺点：Pi和Pj可能同时进入临界区。按下面序列执行时，会同时进入："Pi<a> Pj<a> Pi Pj"。即在检查对方flag之后和切换自己flag之前有一段时间，结果都检查通过。这里的问题出在检查和修改操作不能连续进行。

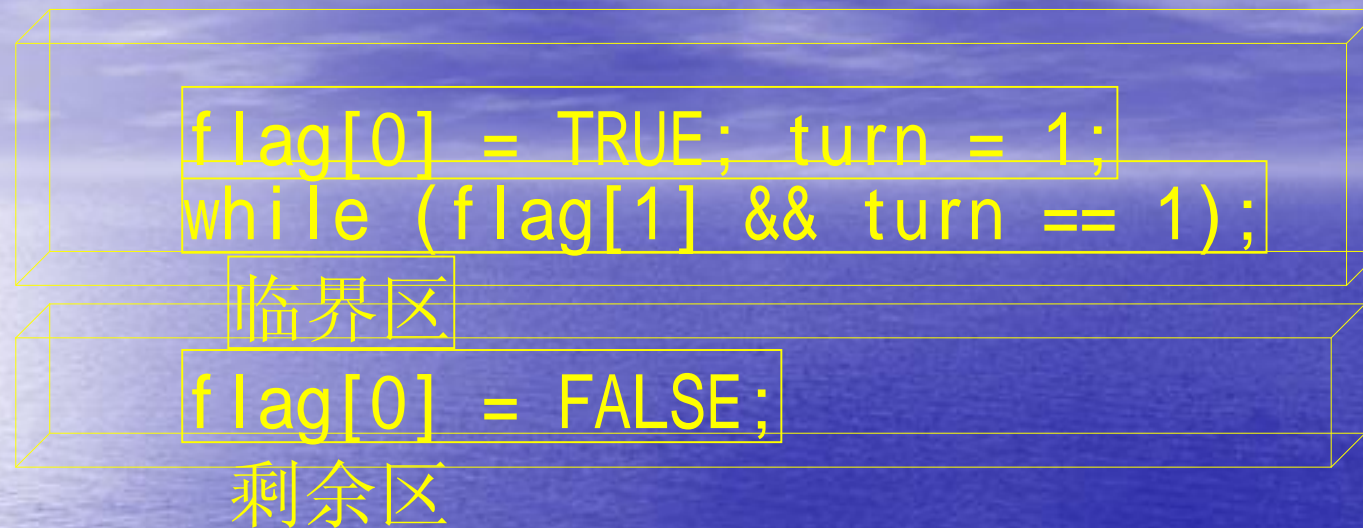
算法3：双标志、后检查



- 类似于算法2，与互斥算法2的区别在于先修改后检查。可防止两个进程同时进入临界区。

- 缺点: P_i 和 P_j 可能都进入不了临界区。按下面序列执行时, 会都进不了临界区:
" $P_i < a > P_j < a > P_i < b > P_j < b >$ ". 即在切换自己flag之后和检查对方flag之前有一段时间, 结果都切换flag, 都检查不通过。

算法4先修改、后检查、后修改者等待



- `turn=1`;描述可进入的进程（同时修改标志时）
- 在进入区先修改后检查，并检查并发修改的先后：
 - 检查对方`flag`，如果不在临界区则自己进入——空闲则入
 - 否则再检查`turn`：保存的是较晚的一次赋值，则较晚的进程等待，较早的进程进入——先到先入，后到等待

进程互斥的硬件方法

- 完全利用软件方法，有很大局限性（如不适于多进程），现在已很少采用。
- 可以利用某些硬件指令——其读写操作由一条指令完成，因而保证读操作与写操作不被打断；

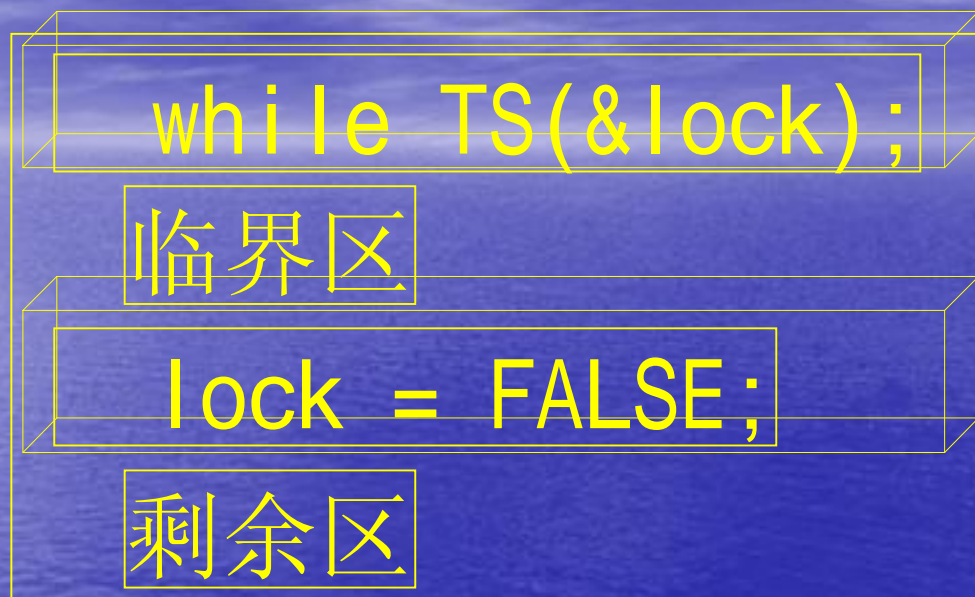
Test-and-Set指令

该指令读出标志后设置为TRUE

```
boolean TS(boolean *lock) {  
    boolean old;  
    old = *lock;    *lock = TRUE;  
    return old;  
}
```

lock表示资源的两种状态：TRUE表示正被占用，FALSE表示空闲

互斥算法（TS指令）



- 利用TS实现进程互斥：每个临界资源设置一个公共布尔变量`lock`，初值为`FALSE`
- 在进入区利用TS进行检查：有进程在临界区时，重复检查；直到其它进程退出时，检查通过；

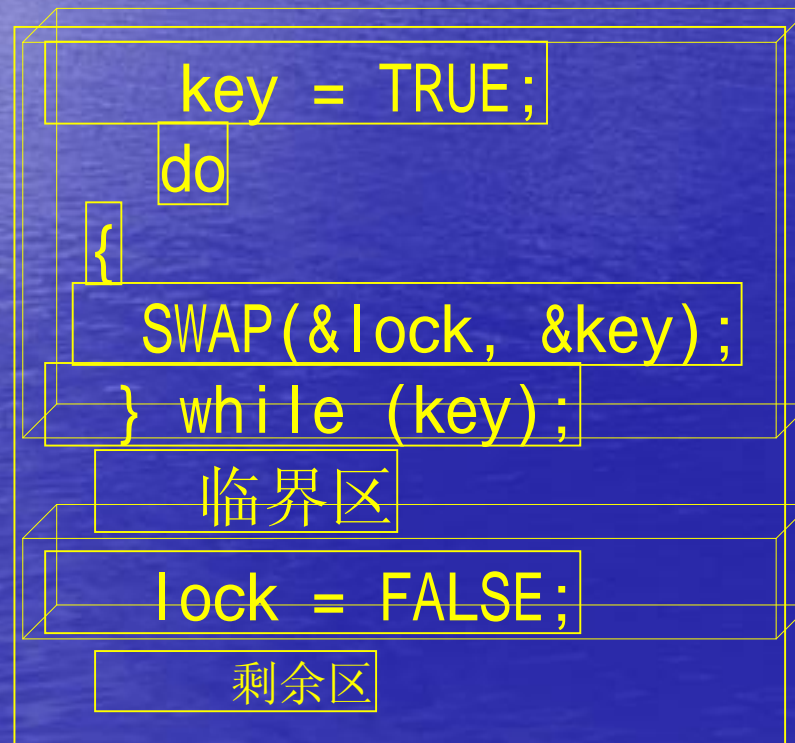
Swap指令（或Exchange指令）

交换两个字（字节）的内容

```
void SWAP(int *a, int *b) {  
    int temp;  
    temp = *a;  *a = *b;  *b =  
temp;  
}
```

互斥算法（Swap指令）

- 利用Swap实现进程互斥：每个临界资源设置一个公共布尔变量lock，初值为FALSE。每个进程设置一个私有布尔变量key



Ø 硬件方法的优点

- Ø 适用于任意数目的进程，在单处理器或多处理器上
- Ø 简单，容易验证其正确性
- Ø 可以支持进程内存在多个临界区，只需为每个临界区设立一个布尔变量

Ø 硬件方法的缺点

- Ø 等待要耗费CPU时间，不能实现"让权等待"
- Ø 可能"饥饿"：从等待进程中随机选择一个进入临界区，有的进程可能一直选不上
- Ø 可能死锁

信号量

前面的互斥算法都存在问题，它们是平等进程间的一种协商机制，需要一个地位高于进程的管理者来解决公共资源的使用问题。

OS可从进程管理者的角度来处理互斥的问题，信号量就是OS提供的管理公共资源的有效手段。

信号量代表可用资源实体的数量。

信号量和P、V原语

信号量集

信号量和P、V原语

- 信号量是荷兰学者DUkstra于1965年提出的一种卓有成效的进程同步机制。
- 每个信号量s除一个整数值s. count(计数)外，还有一个进程等待队列s. queue，其中是阻塞在该信号量的各个进程的标识
- P、V原语的执行，不受进程调度和执行的打断，从而很好地解决了原语操作的整体性的问题。
- 信号量的初始化可指定一个非负整数值，表示空闲资源总数。
- 若信号量为非负整数值，该值表示当前的空闲资源数；若为负值，其绝对值表示当前等待临界区的进程数。

P原语wait(s)

```
--s.count;           //表示申请一个资源;  
if (s.count <0)       //表示没有空闲资源;  
{  
    调用进程进入等待队列s.queue;  
    阻塞调用进程;  
}
```


V原语signal(s)

V原语通常唤醒进程等待队列中的头一个进程

```
++s.count;           //表示释放一个资源;  
if (s.count <= 0) //表示有进程处于阻塞  
    状态;  
{  
    从等待队列s.queue中取出一个进程P;  
    进程P进入就绪队列;  
}
```

利用信号量实现互斥

```
P(mutex);
```

```
critical section
```

```
V(mutex);
```

```
remainder section
```

- 为临界资源设置一个互斥信号量`mutex`，其初值为1；在每个进程中将临界区代码置于`P(mutex)`和`V(mutex)`原语之间
- 必须成对使用`P`和`V`原语：遗漏`P`原语则不能保证互斥访问，遗漏`V`原语则不能在使用临界资源之后将其释放（给其他等待的进程）

经典同步互斥问题

- 1 生产者-消费者问题
- 2 读者-写着问题
- 3 哲学家问题

信号量集

信号量集用于同时需要多个资源时的信号量操作；

AND型信号量集

AND型信号量集用于同时需要多种资源且每种占用一个时的信号量操作；

- 一段处理代码需要同时获取两个或多个临界资源——可能死锁：各进程分别获得部分临界资源，然后等待其余的临界资源，“各不相让”
- 基本思想：在一个原语中，将一段代码同时需要的多个临界资源，要么全部分配给它，要么一个都不分配。称为 **Swait(Simultaneous Wait)**。在 **Swait** 时，各个信号量的次序并不重要，虽然会影响进程归入哪个阻塞队列，但是由于是对资源全部分配或不分配，所以总有进程获得全部资源并在推进之后释放资源，因此不会死锁。

一般“信号量集”

一般信号量集用于同时需要多种资源、每种占用的数目不同、且可分配的资源还存在一个临界值时的处理；

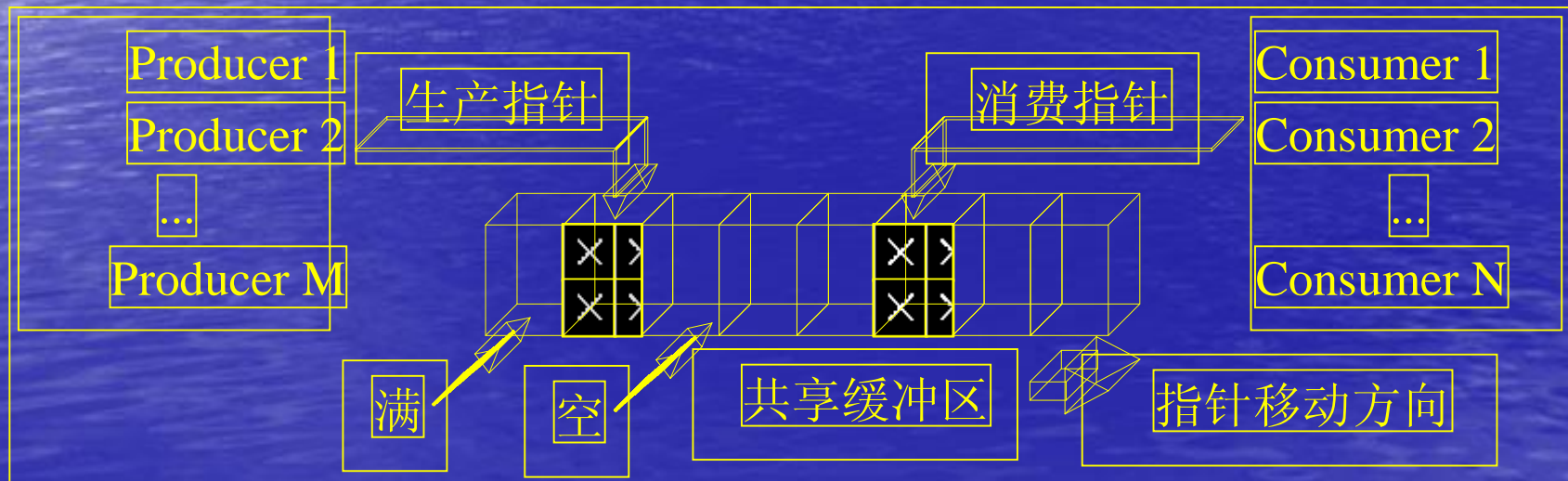
- 一次需要N个某类临界资源时，就要进行N次wait操作——低效又可能死锁
- 基本思想：在AND型信号量集的基础上进行扩充：进程对信号量 S_i 的测试值为 t_i （用于信号量的判断，即 $S_i \geq t_i$ ，表示资源数量低于 t_i 时，便不予分配），占用值为 d_i （用于信号量的增减，即 $S_i = S_i - d_i$ 和 $S_i = S_i + d_i$ ）
- $\text{Swait}(S_1, t_1, d_1; \dots; S_n, t_n, d_n);$
- $\text{Ssignal}(S_1, d_1; \dots; S_n, d_n);$

- 一般"信号量集"的几种特定情况：
 - $\text{Swait}(S, d, d)$ 表示每次申请d个资源，当少于d个时，便不分配；
 - $\text{Swait}(S, 1, 1)$ 表示互斥信号量；
 - $\text{Swait}(S, 1, 0)$ 作为一个可控开关
 - 当 $S \geq 1$ 时，允许多个进程进入临界区；
 - 当 $S = 0$ 时，禁止任何进程进入临界区；
- 一般"信号量集"未必成对使用 Swait 和 Signal ：如：一起申请，但不一起释放；

经典进程同步问题

1. 生产者-消费者问题(the producer-consumer problem)

问题描述：若干进程通过有限的共享缓冲区交换数据。其中，"生产者"进程不断写入，而"消费者"进程不断读出；共享缓冲区共有N个；任何时刻只能有一个进程可对共享缓冲区进行操作。



- 采用信号量机制：
 - full是"满"数目，初值为0，empty是"空"数目，初值为N。实际上，full和empty是同一个含义： $full + empty == N$
 - mutex用于访问缓冲区时的互斥，初值是1
- 每个进程中各个P操作的次序是重要的：先检查资源数目，再检查是否互斥——否则可能死锁(为什么?)
- 采用AND信号量集：Swait(empty, mutex), Ssignal(full, mutex), ...

Producer

P(empty);
P(mutex); //进入区
one unit --> buffer;
V(mutex);
V(full); //退出区

Consumer

P(full);
P(mutex); //进入区
one unit <-- buffer;
V(mutex);
V(empty); //退出区

2. 读者—写者问题(the readers-writers problem)

- 问题描述：对共享资源的读写操作，任一时刻“写者”最多只允许一个，而“读者”则允许多个
有两组并发进程：

读者和写者,共享一组数据区

要求：

允许多个读者同时执行读操作

不允许读者、写者同时操作

不允许多个写者同时操作

- 采用信号量机制：
 - Wmutex表示"允许写", 初值是1。
 - 公共变量Rcount表示"正在读"的进程数, 初值是0;
 - Rmutex表示对Rcount的互斥操作, 初值是1。

读者:

```
while (true) {  
    P(mutex);  
    readcount ++;  
    if (readcount==1)  
        P (w);  
    V(mutex);  
    读  
    P(mutex);  
    readcount --;  
    if (readcount==0)  
        V(w);  
    V(mutex);  
};
```

写者:

```
while (true) {  
  
    P(w);  
    写  
    V(w);  
  
};
```

- 采用一般"信号量集"机制：问题增加一个限制条件：同时读的"读者"最多R个
 - Wmutex表示"允许写"，初值是1
 - Rcount表示"允许读者数目"，初值为R

读者：

```
Swait(wmutex,1,1;  
      rcount,R,0);
```

写；

```
Ssignal(wmutex,1);
```

写者：

```
Swait(rcount,1,1;  
      wmutex,1,0);
```

写；

```
Ssignal(rcount,1);
```


写者优先的算法

```
写者:  begin
        wait(mut1);
wcount:=wcount+1;
if wcount=1 then wait(rmutex);
    signal(mut1);
    wait(wmutex);
    写数据集;
    wait(mut1);
    wcount:=wcount-1;
    if wcount=0 then signal(rmutex);
    signal(wmutex);
    signal(mut1);
    end
```

```
读者:  begin
        wait(mut1);

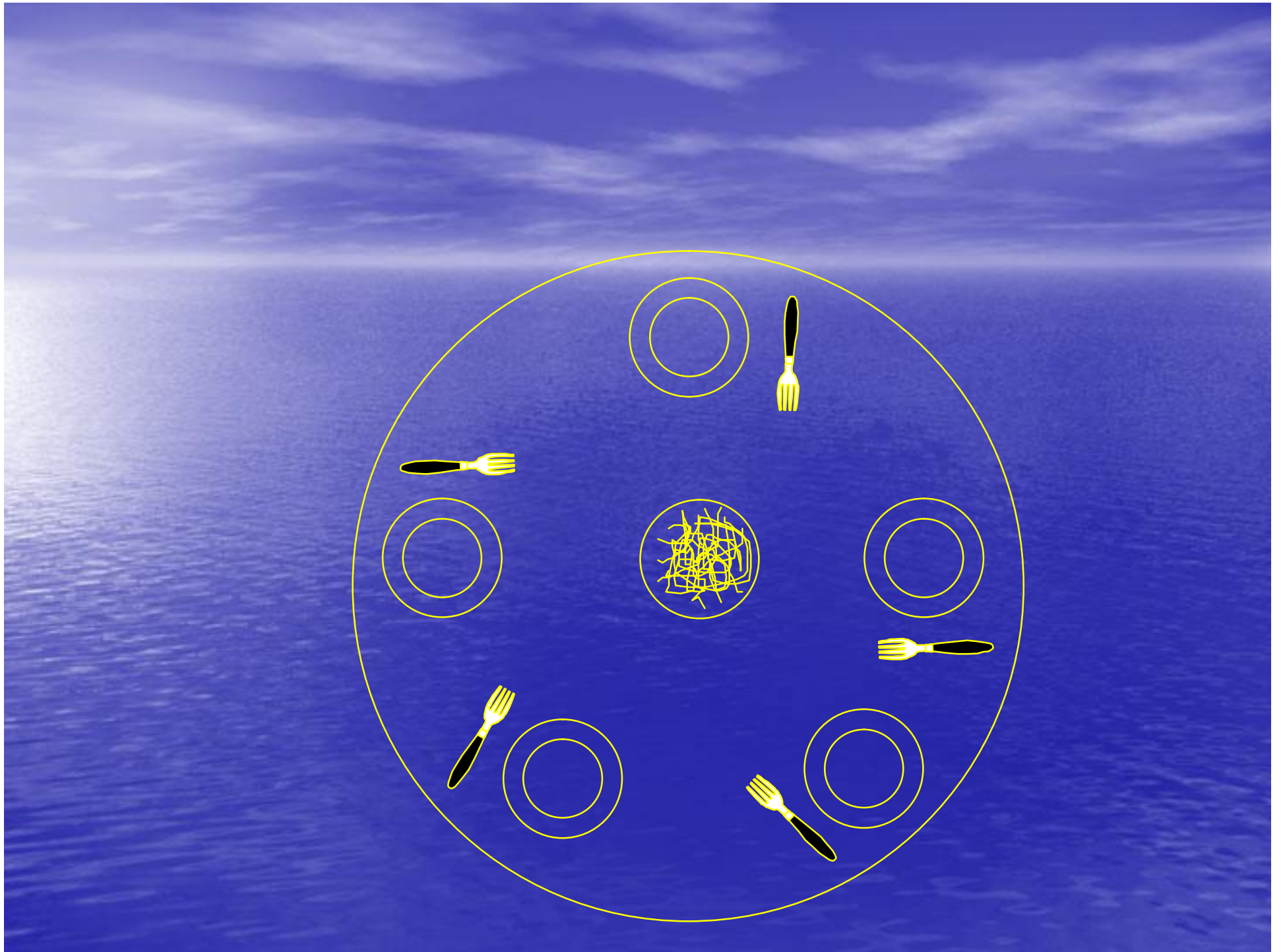
        signal(mut1);
        wait(mut2);
        rcount=rcount+1;
if rcount=1 then wait(rmutex)
    signal(mut2);
    读数据集
        wait(mut2);
        rcount:=rcount-1;
    if rcount=0 then signal(rmutex);
    singal(mut2);
    end
```


哲学家进餐问题

有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，每两人之间放一只筷子

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筷子，并且每个人只能直接从自己的左边或右边去取筷子



管程

Ø 用信号量可实现进程间的同步，但由于信号量的控制分布在整个程序中，易读性差、不利于修改和维护、正确性难以保证

Ø 管程方便地阻塞和唤醒进程。管程可以函数库的形式实现。

Ø 其基本思想是把信号量及其操作原语封装在一个对象内部。即所有操作集中在一个模块中。

管程的主要特性

- (一) 模块化，一个管程是一个基本程序单位，可以单独编译
- (二) 抽象数据类型，管程是一种特殊的数据类型，其中不仅有数据，而且有对数据进行操作的代码
- (三) 信息掩蔽，管程是半透明的，管程中的外部过程（函数）实现了某些功能，在其外部则是不可见的
- (四) 规定管程互斥进入

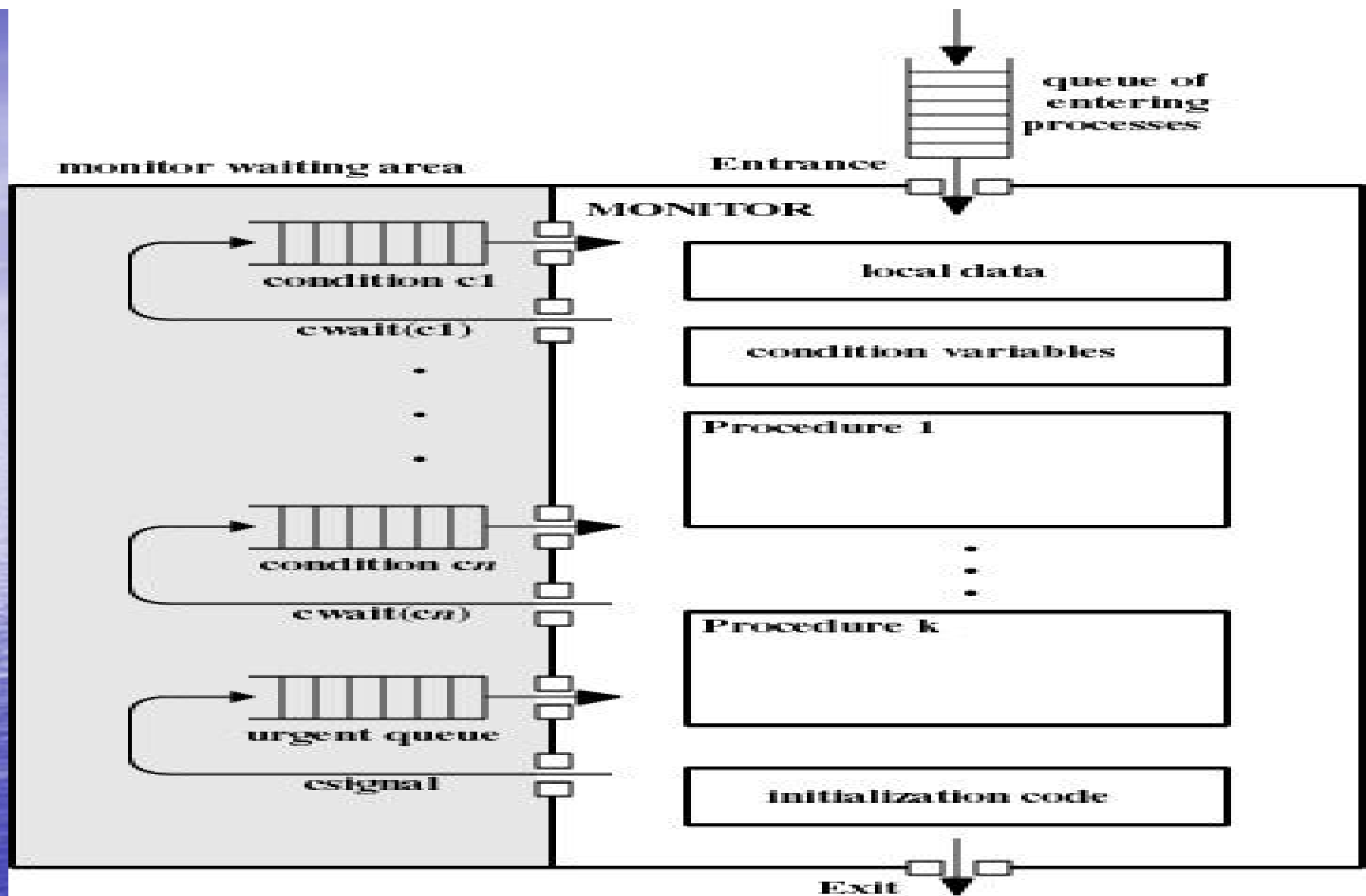


Figure 5.21 Structure of a Monitor

- 入口等待队列：因为管程是互斥进入的，所以当有一个进程试图进入一个已被占用的管程时它应当在管程的入口处等待，因而在管程的入口处应当有一个进程等待队列，称作入口等待队列。
- 紧急等待队列：在管程内部，由于执行唤醒操作，可能会出现多个等待进程（已被唤醒，但由于管程的互斥进入而等待），因而还需要有一个进程等待队列，这个等待队列被称为紧急等待队列。它的优先级应当高于入口等待队列的优先级。

Windows 进程同步与互斥

- **CreateMutex**
- 函数功能：创建有名或无名的互斥对象
- 函数原型：
`HANDLE CreateMutex(LPSECURITY_ATTRIBUTES lpMutexAttributes, BOOL bInitialOwner, LPCTSTR lpName);`
- 返回值：若成功则返回互斥对象句柄，若失败则返回NULL。
- **OpenMutex**
- **WaitForSingleObject**
- 函数功能：当下列情况之一发生时函数返回：(1)指定对象处于信号态 (2) 超时
- 函数原型：
`DWORD WaitForSingleObject(HANDLE hHandle, DWORD dwMilliseconds);`
- 返回值：
调用成功则返回引起函数返回的事件，否则返回WAIT_FAILED。
- **ReleaseMutex**
- 函数功能：放弃指定互斥对象的所有权
- 函数原型：
`BOOL ReleaseMutex(HANDLE hMutex);`
- 返回值：
成功则返回非零值，否则返回零。
- 例如：

```
HANDLE h_Mutex;  
h_Mutex = CreateMutex(NULL, FALSE, "mutex_for_readcount");  
.....  
HANDLE h_Mutex;  
h_Mutex = OpenMutex(MUTEX_ALL_ACCESS, FALSE, "mutex_for_readcount");  
...  
DWORD wait_for_mutex;  
wait_for_mutex = WaitForSingleObject(h_Mutex, -1);  
ReleaseMutex(h_Mutex);
```

- *f* Sleep
- 函数功能：对于指定的时间间隔挂起当前的执行线程
- 函数原型：
 - VOID Sleep(DWORD dwMilliseconds);
- 返回值：无
- „ WaitForMultipleObjects
- 函数功能：
 - 当满足下列条件之一时返回：（1）任意一个或全部指定对象处于信号态；（2）超时。
- 函数原型：
 - DWORD WaitForMultipleObject(DWORD ncount, CONST HANDLE *lpHandles, BOOL fWaitAll, DWORD dwMilliseconds);
- 返回值：
 - 调用成功则返回引起函数返回的事件，否则返回WAIT_FAILED。
- 例如：
 - DWORD wait_for_all;
 - DWORD n_thread; // 线程数目
 - HANDLE h_Thread[MAX_THREAD_NUM];
 -
 - wait_for_all = WaitForMultipleObject(n_thread, h_Thread, TRUE, -1);


- **CreateThread**
- 函数功能：创建一个在调用进程的地址空间中执行的线程
- 函数原型：
 - `HANDLE CreateThread(LPSECURITY_ATTRIBUTES lpThreadAttributes,`
 - `DWORD dwStackSize, LPTHREAD_START_ROUTINE lpStartAddress,`
 - `LPVOID lpParameter, DWORD dwCreationFlags, LPDWORD lpThreadId);`
- 返回值：
 - 成功则返回新线程的句柄，否则返回NULL
- 例如：
 - `HANDLE h_Thread;`
 - `DWORD thread_ID;`
 - `ThreadInfo thread_info; // 假设ThreadInfo是用户预定义`
- 结构
 - `.....`
 - `h_Thread = Create(NULL, 0,`
 - `(LPTHREAD_START_ROUTINE)(RP_WriterThread), &thread_info, 0,`
 - `&thread_ID);`
- **ExitThread**
- 函数功能：结束一个线程
- 函数原型：
 - `VOID ExitThread(DWORD dwExitCode);`
- 返回值：无

- **InitializeCriticalSection**
- 函数功能：初始化临界区对象（临界区使用之前必须初始化）
- 函数原型：
 - `VOID InitializeCriticalSection(LPCRITICAL_SECTION lpCriticalSection);`
- 返回值：无
- **EnterCriticalSection**
- 函数功能：等待指定临界区对象的所有权。当调用线程被赋予所有权时，该函数返回。
- 函数原型：
 - `VOID EnterCriticalSection(LPCRITICAL_SECTION lpCriticalSection);`
- 返回值：无
- **LeaveCriticalSection**
- 函数功能：释放指定临界区对象的所有权。
- 函数原型：
 - `VOID LeaveCriticalSection(LPCRITICAL_SECTION lpCriticalSection);`
- 返回值：
 - 调用成功则返回引起函数返回的事件，否则返回WAIT_FAILED。
- 例如：
 - `CRITICAL_SECTION RP_Write; // 定义一个临界区对象`
 - `InitializeCriticalSection(&RP_Write);`
 - `.....`
 - `EnterCriticalSection(&RP_Write);`
 - `.....`
 - `LeaveCriticalSection(&RP_Write);`

- • **CreateSemaphore**
- 函数功能：创建一个有名或无名的信号量对象
- 函数原型：
HANDLE CreateSemaphore(LPSECURITY_ATTRIBUTES lpAttributes,
LONG lInitialCount, LONG lMaximumCount, LPCTSTR lpName);
- 返回值：
若成功则返回信号量对象句柄，失败则返回NULL。
- , **ReleaseSemaphore**
- 函数功能：将指定信号量对象的计数增加一个指定的数量。
- 函数原型：
BOOL ReleaseSemaphore(HANDLE hSemaphore, LONG lReleaseCount,
LPLONG lpPreviousCount);
- 返回值：
成功则返回TRUE，否则返回FALSE。

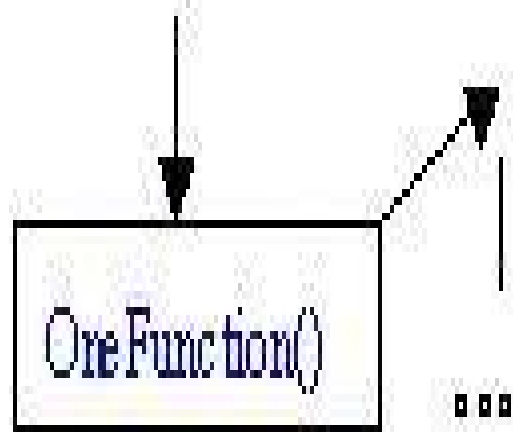
下面问题：m取值为1还是0？

- ```
#include<iostream.h>
#include<windows.h>
int m=0;
DWORD WINAPI proc(LPVOID pParam){
 m=1;
 return 1;
}
void main(){
 CreateThread(NULL,0,&proc,NULL,0,NULL);
 cout<<m;
}
```


- **WaitForSingleObject(handle,INFINITE);**

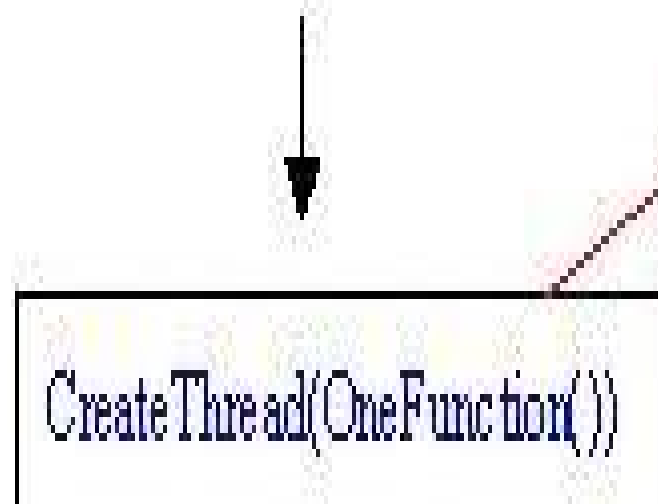


主程序



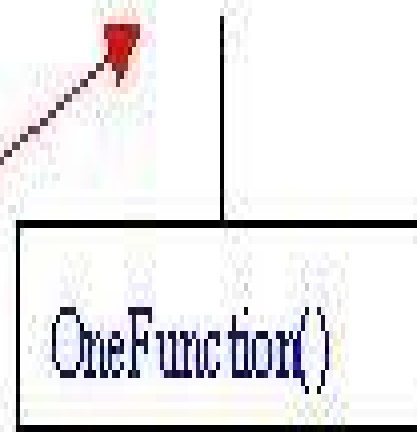
等候函数执行完毕后，  
继续向下执行

主程序



不必等候函数执行完毕，  
继续向下执行

产生分枝程序，执行函数



完毕

# 题目:吃苹果

- 父亲、儿子、女儿三人和一个盘子，当盘子空时，父亲往盘中随机放苹果或香蕉，儿子只从盘中拿苹果，女儿只从盘中拿香蕉。



```
• DWORD WINAPI Son(LPVOID n)
• {
• int i = 1;
• HANDLE Apple;
• Apple = ::OpenSemaphore(MUTEX_ALL_ACCESS,false,"apple");
• while (1)
• {
• ::WaitForSingleObject(Apple,INFINITE);//等苹果
• cout<<"Son eats "<<i<<" apples"<<endl;
• LeaveCriticalSection(&mmutex);
• i++;
• }
• ::CloseHandle(Apple);
• return 0;
• }
```

```
• DWORD WINAPI Daughter(LPVOID n)
• {
• int i = 1;
• HANDLE Banana;
• Banana = ::OpenSemaphore(MUTEX_ALL_ACCESS,false,"banana");
• while (1)
• {
• ::WaitForSingleObject(Banana,INFINITE);//等香蕉
• cout<<"Daughter eats "<<i<<" bananas"<<endl;
• LeaveCriticalSection(&mmutex);
• i++;
• }
• ::CloseHandle(Banana);
• return 0;
• }
```



```
• DWORD WINAPI Father(LPVOID n)
• {
• UINT fruit;
• EnterCriticalSection(&mmutex);
• fruit = rand()%2;
• if (fruit == 0)
• {
• //盘中放入苹果
• cout<<k+1<<" father produce an apple"<<endl;
• ::ReleaseSemaphore(Apple,1,NULL);
• }
• else { //盘中放入香蕉
• cout<<k+1<<" father produce a banana"<<endl;
• ::ReleaseSemaphore(Banana,1,NULL);
• }
• k=k+1;
• return 0;
• }
```

```
• int main()
• {
• k=0;
• Apple = ::CreateSemaphore(NULL,0,1,"apple");
• Banana = ::CreateSemaphore(NULL,0,1,"banana");
• InitializeCriticalSection(&mmutex);
•
• srand((unsigned)time(NULL));
• for (j= 0 ; j < 20; j++)
• {
• ::CreateThread(NULL,0,Father,NULL,0,0);};
• ::CreateThread(NULL,0,Son,NULL,0,0);
• ::CreateThread(NULL,0,Daughter,NULL,0,0);
• return 0;
• }
```



# 进程间通信(IPC)

Windows2000/xp信号

Windows2000/Xp基于文件映射的共享存储区

Windows2000/XP的管道

Windows2000/Xp邮件槽

套接字

# 进程间通信的类型

- 低级通信：只能传递状态和整数值（控制信息），包括进程互斥和同步所采用的信号量和管程机制。优点的速度快。缺点是：
  - 传送信息量小：效率低，每次通信传递的信息量固定，若传递较多信息则需要多次通信。
  - 编程复杂：用户直接实现通信的细节，编程复杂，容易出错。
- 高级通信：能够传送任意数量的数据，包括三类：共享存储区、管道、消息。



# 直接通信和间接通信

- 直接通信：信息直接传递给接收方，如管道。
  - 在发送时，指定接收方的地址或标识，也可以指定多个接收方或广播式地址；
  - 在接收时，允许接收来自任意发送方的消息，并在读出消息的同时获取发送方的地址。
- 间接通信：借助于收发双方进程之外的共享数据结构作为通信中转，如消息队列。通常收方和发方的数目可以是任意的。



# Windows2000/XP的信号

|                     |                                                                                                                                          |
|---------------------|------------------------------------------------------------------------------------------------------------------------------------------|
| CTRL_C_EVENT        | A CTRL+C signal was received                                                                                                             |
| CTRL_BREAK_EVENT    | A CTRL+BREAK signal was received                                                                                                         |
| CTRL_CLOSE_EVENT    | A signal that the system sends to all processes attached to a console when the user closes the console                                   |
| CTRL_LOGOFF_EVENT   | A signal that the system sends to console processes when a user is logging off. This signal does not indicate which user is logging off. |
| CTRL_SHUTDOWN_EVENT | A signal that the system sends to console processes when the system is shutting down.                                                    |

|         |                            |
|---------|----------------------------|
| SIGABRT | Abnormal termination       |
| SIGFPE  | Floating-point error       |
| SIGILL  | Illegal instruction        |
| SIGINT  | CTRL+C signal (对 Win32 无效) |
| SIGSEGV | Illegal storage access     |
| SIGTERM | Termination request        |

# Windows2000/XP

## 基于文件映射的共享存储区

相当于内存，可以任意读写和使用任意数据结构（当然，对指针要注意），需要进程互斥和同步的辅助来确保数据一致性

- 采用文件映射机制：可以将整个文件映射为进程虚拟地址空间的一部分来加以访问。



# Windows2000/XP的管道

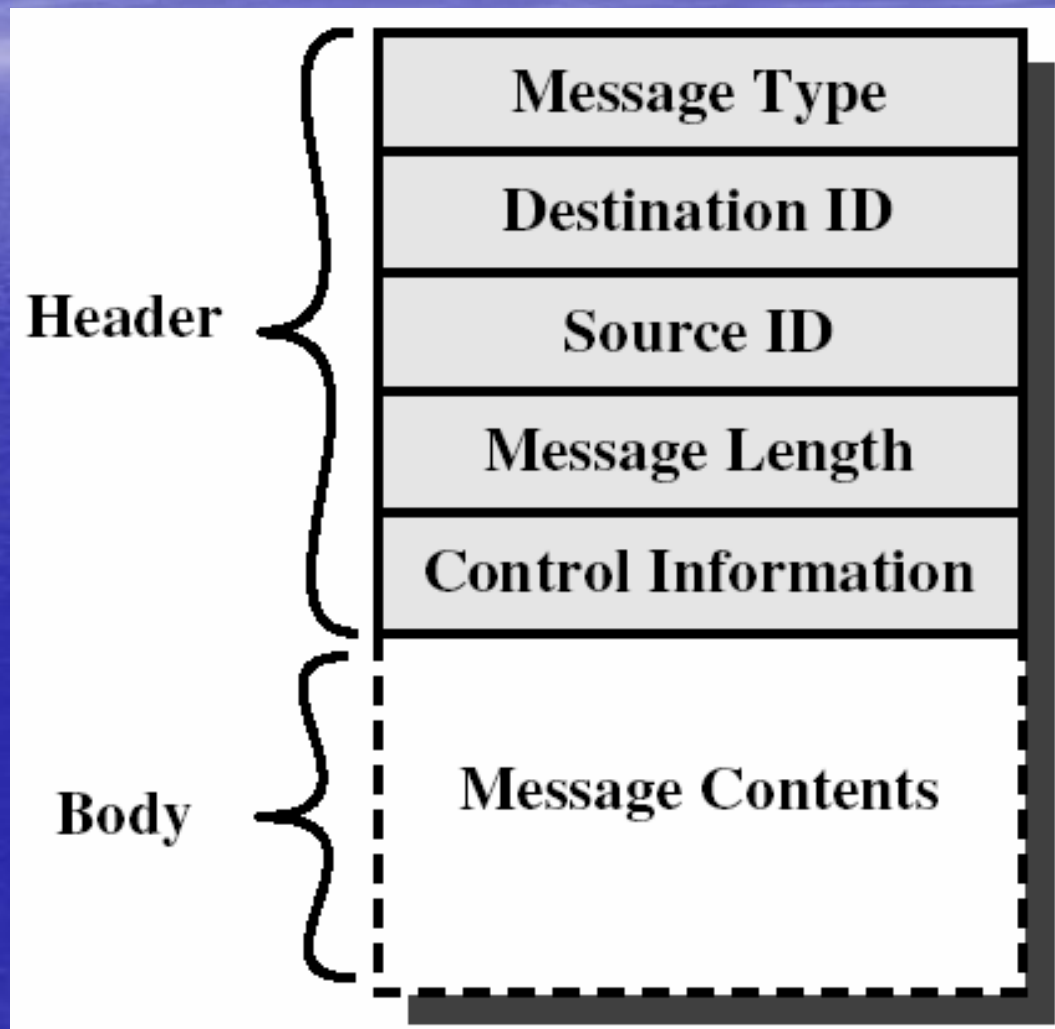
无名管道：类似于UNIX管道，CreatePipe可创建无名管道，得到两个读写句柄；利用ReadFile和WriteFile可进行无名管道的读写；

```
BOOL CreatePipe(PHANDLE hReadPipe,
 // address of variable for read handle
 PHANDLE hWritePipe,
 // address of variable for write handle
 LPSECURITY_ATTRIBUTES lpPipeAttributes,
 // pointer to security attributes
 DWORD nSize // number of bytes reserved for
pipe
);
```



# 消息

与窗口系统中的“消息”不同。通常是不定长数据块。消息的发送不需要接收方准备好，随时可发送。



# Windows2000/Xp邮件槽

- 邮件槽(mailslot): 驻留在OS核心, 不定长数据块(报文), 不可靠传递通常采用client-server模式:

套接字

Windows中的规范称为"Winsock"

# 死锁问题

- 死锁是多个进程因竞争资源且推进顺序不合理而造成的一种僵局，若无外力作用，这些进程将永远不能再向前推进

死锁的起因：竞争资源



一般地，可以把死锁描述为：有并发进程 $P_1, P_2, \dots, P_n$ ，它们共享资源 $R_1, R_2, \dots, R_m$ （ $n > 0, m > 0, n \geq m$ ）。其中，每个 $P_i$ （ $1 \leq i \leq n$ ）拥有资源 $R_j$ （ $1 \leq j \leq m$ ），直到不再有剩余资源。同时，各 $P_i$ 又在不释放 $R_j$ 的前提下要求得到 $R_k$ （ $k \neq j, 1 \leq k \leq m$ ），从而造成资源的互相占有和互相等待。在没有外力驱动的情况下，该组并发进程停止往前推进，陷入永久等待状态。

## 2. 死锁的起因

死锁的起因是并发进程的资源竞争。产生死锁的根本原因在于系统提供的资源个数少于并发进程所要求的该类资源数。可以采用适当的资源分配算法，以达到消除死锁的目的。为此，先看看产生死锁的必要条件。



产生死锁必须同时满足下列四个必要条件：

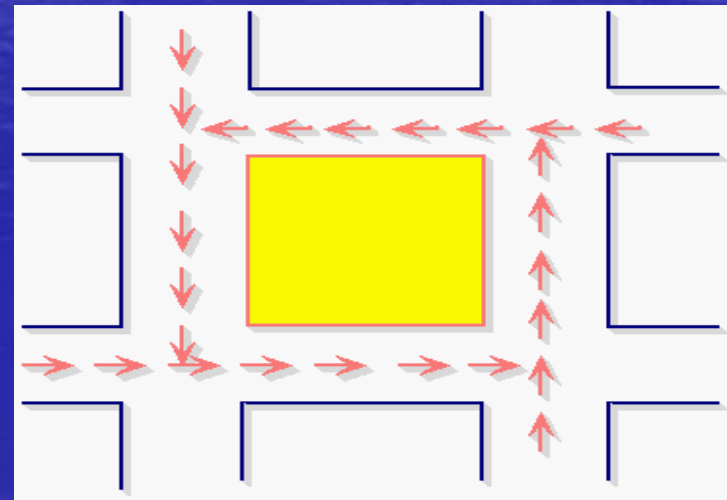
- (1)互斥条件。
- (2)不剥夺条件。
- (3) 部分分配。
- (4) 环路条件。

显然，只要使上述4个必要条件中的某一个不满足，则死锁就可以排除。



# 死锁的现象

- 1 互斥条件：每条道路只能通过一辆车
- 2 保持并请求（部分分配）：占用一段道路并等待前方道路的释放
- 3 不剥夺：单行线不能超越抢道
- 4 环路条件：如图



# 死锁的预防

- 根据产生死锁的四个必要条件，只要使其中之一不能成立，死锁就不会出现。
- (1)破坏互斥条件：
- (2)破坏占有等待条件：
- (3)破坏非剥夺条件：
- (4)破坏循环等待条件：



# 破坏互斥条件

- 方法
  - 允许多个进程同时使用资源
- 适用条件
  - 资源的固有特性允许多个进程同时使用（如文件允许多个进程同时读）
  - 借助特殊技术允许多个进程同时使用（如打印机借助Spooling技术）
- 缺点
  - 不适用于绝大多数资源

# 破坏占有及等待条件

- 方法

- 禁止已拥有资源的进程再申请其他资源，如要求所有进程在开始时一次性地申请在整个运行过程所需的全部资源，或申请资源时要先释放其占有资源再一次性申请所需全部资源

- 优点

- 简单、易于实现、安全

- 缺点

- 进程延迟运行
- 资源严重浪费



# 破坏不可剥夺条件

- 方法

- 一个已经占有了某些资源的进程, 当它再提出新的资源请求而不能立即得到满足时, 必须释放它已经占有的所有资源, 待以后需要时再重新申请
- OS可以剥夺一个进程占有的资源, 分配给其他进程

- 适用条件

- 资源的状态可以很容易地保存和恢复(如CPU)

- 缺点

- 实现复杂、代价大, 反复申请/释放资源, 系统开销大, 降低系统吞吐量

# 破坏环路等待条件

- 方法

- 要求每个进程任何时刻只能占有一个资源，如果要申请第二个则必须先释放第一个（不现实）
- 对所有资源按类型进行线性排队，进程申请资源必须严格按资源序号递增的顺序
- 缺点
- 很难找到令每个人都满意的编号次序，类型序号的安排只能考虑一般作业的情况，限制用户简单、自主地编程，易造成资源浪费



# 死锁的避免

- 死锁预防是设法至少破坏产生死锁的必要条件之一，而死锁的避免是在系统运行过程中小心地避免死锁的最终发生。最著名的死锁避免算法是Dijkstra提出的银行家算法。
- 安全状态:所谓安全状态是指如果存在一个由系统中所有进程构成的安全序列 $\langle P_1, P_2, \dots, P_n \rangle$ ，而一个进程序列 $\langle P_1, P_2, \dots, P_n \rangle$ 是安全的，如果对于每一个进程 $P_i (1 \leq i \leq n)$ ，它以后尚需要的资源数量不超过系统中当前剩余资源数量与所有进程 $P_j (j < i)$ 当前占有资源量之和。安全状态是没有死锁的状态，非安全状态称为不安全状态，但不一定是死锁状况。

# 银行家算法问题的描述

- 例：某银行家共有资金100万元和10个借贷工厂
  - 每个工厂需要20(万元)才能开始生产
  - 10个工厂分别贷款为10(万元)
  - 考虑：
    - 银行家可能收回全部贷款吗？
    - 如果银行家给前5个工厂每个20 (万元)可以满足吗？



# 习题

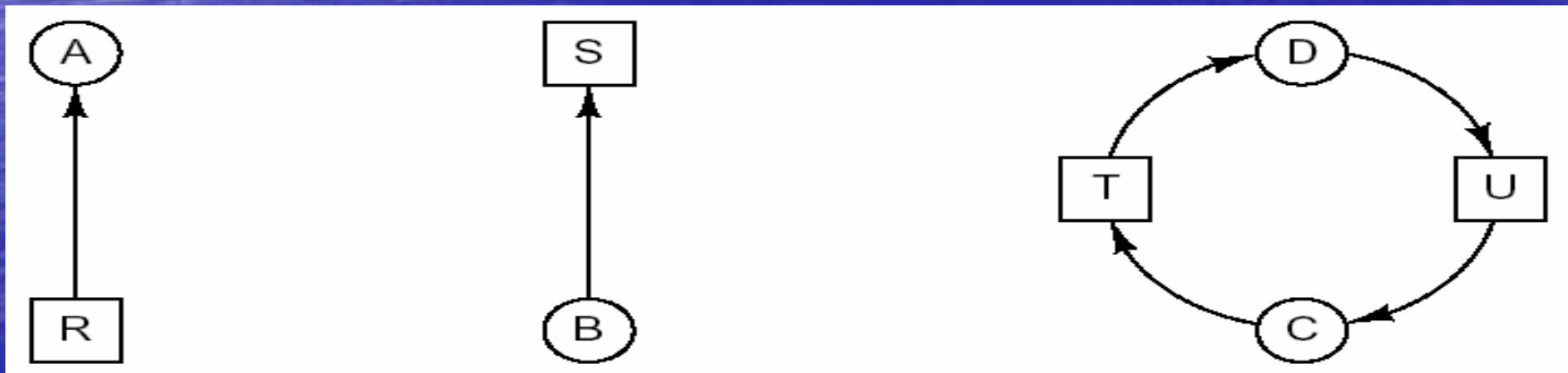
| 01 | MAX |   |   | ALLO |   |   | NEED |   |   | AVA |   |   |
|----|-----|---|---|------|---|---|------|---|---|-----|---|---|
|    | A   | B | C | A    | B | C | A    | B | C | A   | B | C |
| P0 | 7   | 5 | 3 | 0    | 1 | 0 | 7    | 4 | 3 | 3   | 3 | 2 |
| P1 | 3   | 2 | 2 | 2    | 0 | 0 | 1    | 2 | 2 |     |   |   |
| P2 | 9   | 0 | 2 | 3    | 0 | 2 | 6    | 0 | 0 |     |   |   |
| P3 | 2   | 2 | 2 | 2    | 1 | 1 | 0    | 1 | 1 |     |   |   |
| P4 | 4   | 3 | 3 | 0    | 0 | 2 | 4    | 3 | 1 |     |   |   |

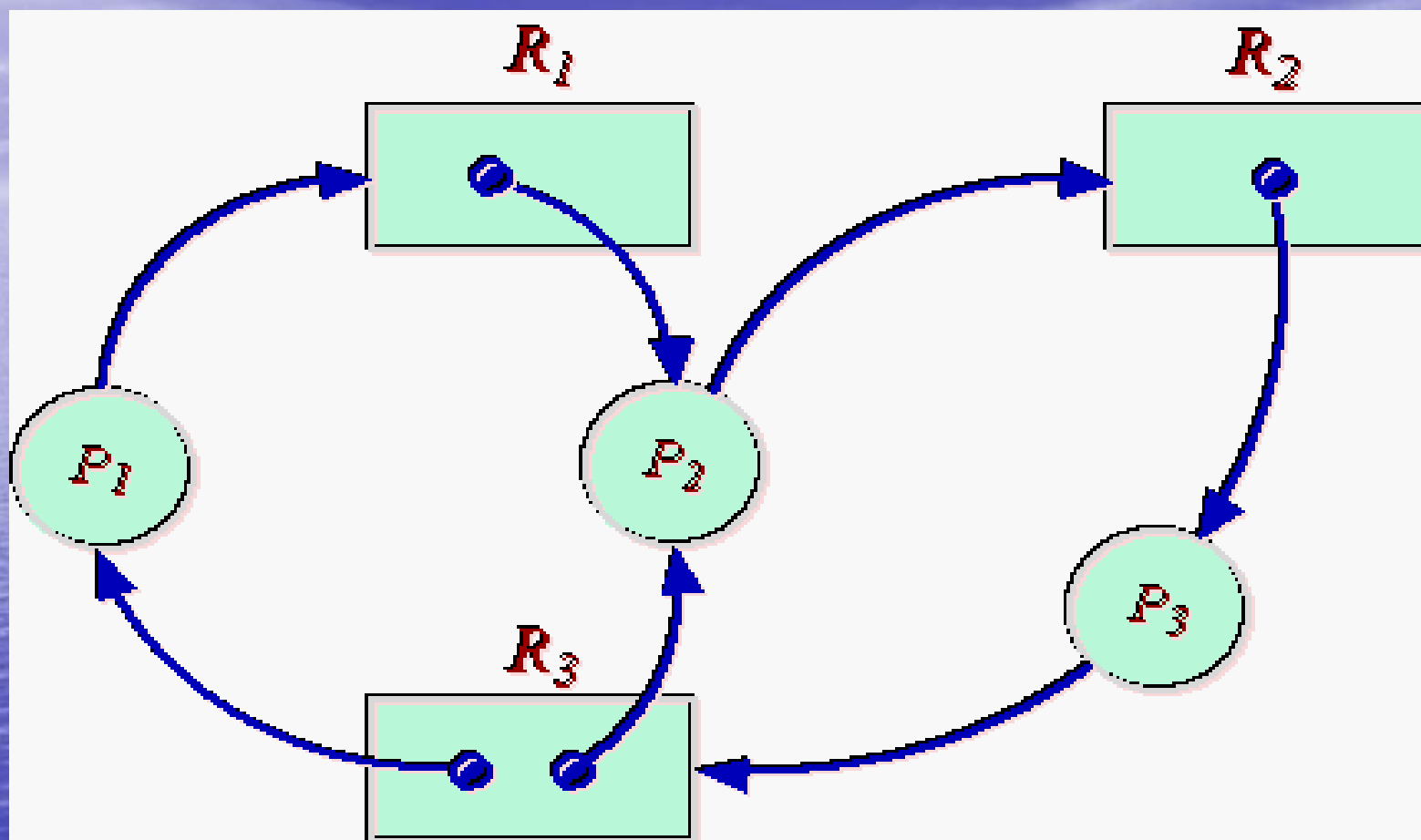
# 死锁的检测与解除

- 解决死锁的另一条途径是死锁检测，通常采用化简资源分配图的方法，其过程如下：
- (1)在图中找一个进程顶点 $P_1$ ,  $P_1$ 的请求边均能立即满足。
- (2)若找到这样的 $P_1$ ，则将与其相连的边全部删去，转至(1)，否则化简结束。
- 如果化简后所有的进程顶点都成了孤立点，就称之为完全化简，且无死锁现象，反之则称之为不完全化简，且非孤立点的进程处于死锁状态。一旦发现死锁应立即排除，以确保系统正常运行，常采用资源剥夺法和撤消进程法。



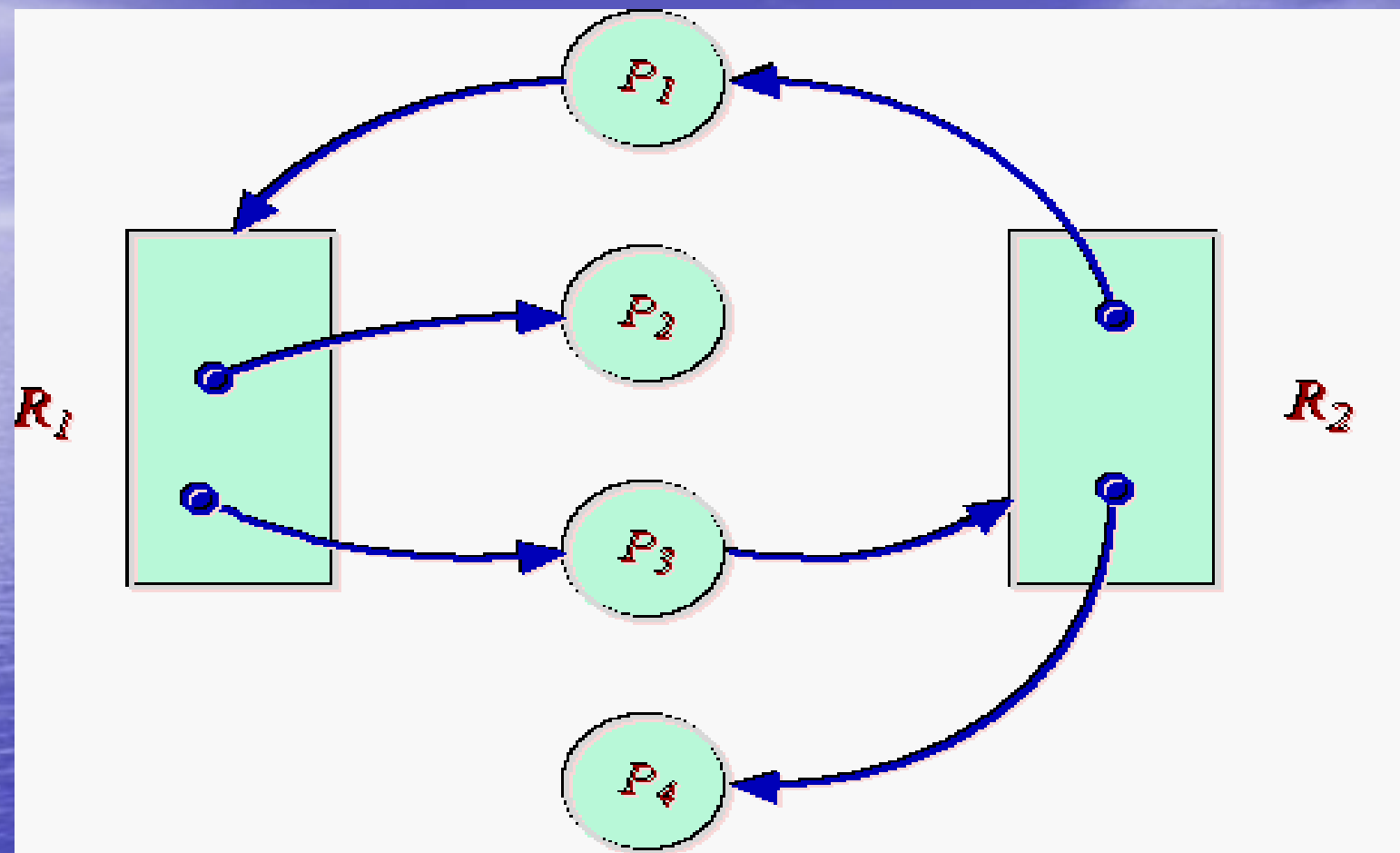
- 资源类（资源的不同类型）：方框
- 资源实例（存在于每个资源类中）：黑圆点
- 进程：圆圈中加进程名
- 分配边：资源实例指向进程的一条有向边
- 申请边：进程指向资源类的一条有向边





有环有死锁





有环无死锁

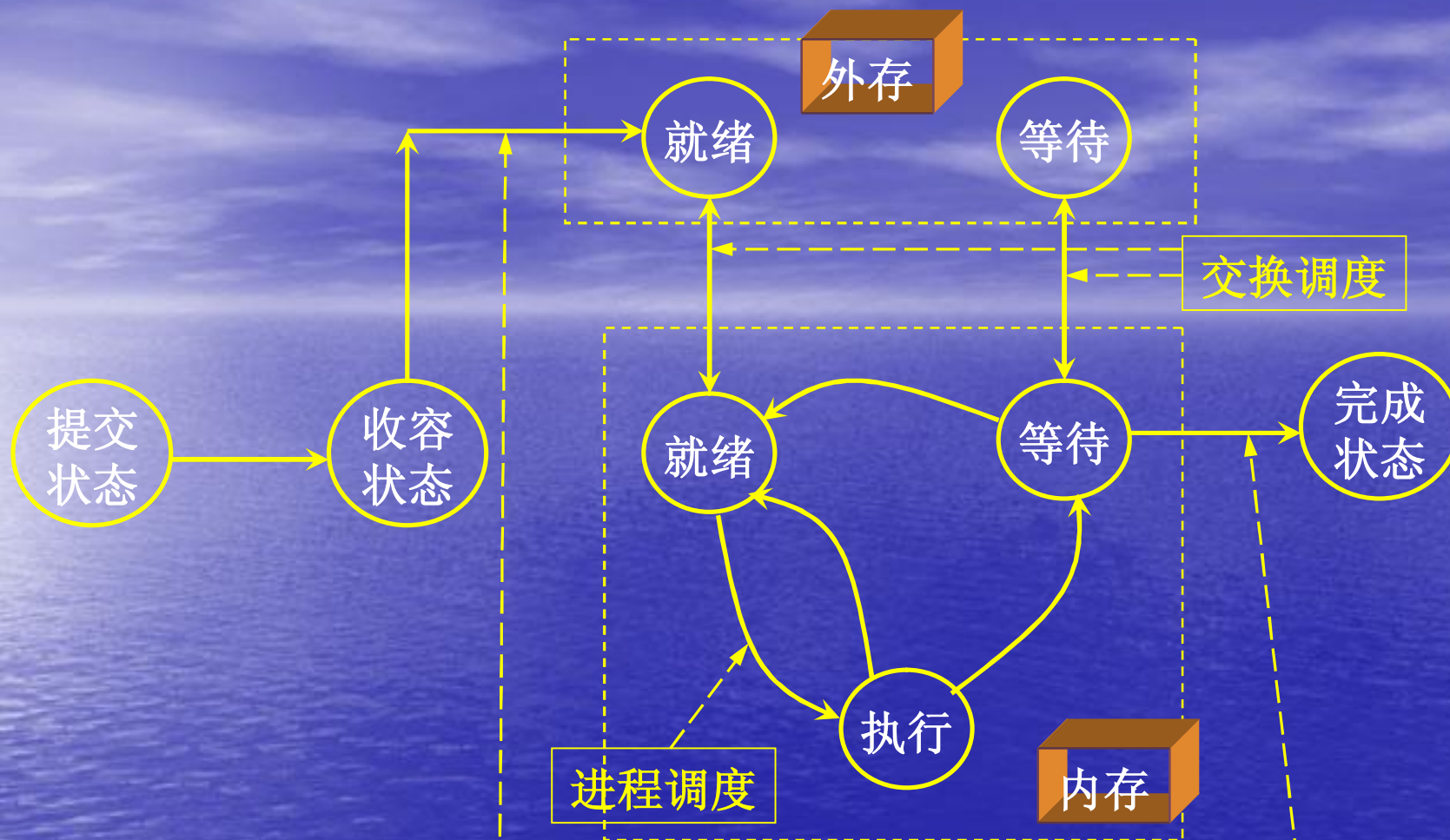
| 方法 | 资源分配策略                      | 各种可能模式                             | 主要优点                                                         | 主要缺点                                           |
|----|-----------------------------|------------------------------------|--------------------------------------------------------------|------------------------------------------------|
| 预防 | 保守的；宁可资源闲置                  | 一次请求所有资源<br><br>资源剥夺<br><br>资源按序申请 | 适用于作突发式处理的进程；不必剥夺<br>适用于状态可以保存和恢复的资源<br>可以在编译时（而不必在运行时）就进行检查 | 效率低；进程初始化时间延长剥夺次数过多；<br>多次对资源重新启动<br>不便灵活申请新资源 |
| 避免 | 是“预防”和“检测”的折衷（在运行时判断是否可能死锁） | 寻找可能的安全的运行顺序                       | 不必进行剥夺                                                       | 必须知道将来的资源需求；进程可能会长时间阻塞                         |
| 检测 | 宽松的；只要允许，就分配资源              | 定期检查死锁是否已经发生                       | 不延长进程初始化时间；允许对死锁进行现场处理                                       | 通过剥夺解除死锁，造成损失                                  |



# 处理器调度概述

## 按照调度的层次

- 作业：又称为“宏观调度”、“高级调度”。
- 内外存交换：又称为“中级调度”。
- 进程或线程：又称为“微观调度”、“低级调度”。



高级调度：作业调度  
中级调度：交换调度  
低级调度：进程调度

作业调度

## 处理机调度的层次



# 调度的性能准则

- 调度目标:

一般来说, 调度目标主要是以下四点

- (1)公平合理: 对所有作业应该是公平合理的;
- (2)高利用率: 应使设备有高的利用率,
- (3)吞吐量大: 每天执行尽可能多的作业;
- (4)响应迅速: 有快的响应时间。

# 调度性能准则

任一调度算法要想同时满足上述目标是不可能的：

- 1) 如要想吞吐量大，调度算法就应选择那些估计执行时间短的作业。这对那些估计执行时间长的作业不公平，并且可能使它们的得不到调度执行或响应时间很长。
- 2) 如果考虑的因素过多，调度算法就会变得非常复杂。其结果是系统开销增加，资源利用率下降。



周转时间:

作业*i*的周转时间 $T_i$ 为

$$T_i = T_{ei} - T_{si}$$

其中 $T_{ei}$ 为作业*i*的完成时间;  $T_{si}$ 为作业的提交时间。

$N$ 个作业的平均周转时间 $T = (T_1 + T_2 + \dots + T_N) / N$

带权周转时间:

周转时间 $T_i$ 可分解为两部分, 即

$$T_i = T_{wi} + T_{ri}$$

这里,  $T_{wi}$ 主要指作业*i*由后备状态到执行状态的等待时间

$T_{ri}$ 指作业*i*的执行时间。

带权周转时间是作业周转时间与作业执行时间的比:

$$W_i = T_i / T_{ri}$$

$N$ 个作业的平均带权周转时间 $W = (W_1 + W_2 + \dots +$

$W_N) / N$

# 调度算法

通常将作业或进程归入各种就绪或阻塞队列。有的算法适用于作业调度，有的算法适用于进程调度，有的两者都适应。

先来先服务

短作业优先

时间片轮转算法

多级队列算法

优先级算法

多级反馈队列算法



- 常用作业调度算法

- 1 先来先服务 (First come first serve, FCFS) 方式:

执行时间很短的作业是在那些长作业的后面到达系统的话, 则必须等待很长时间

- 2 短作业优先 (Shortest Job first, SJF) 方式

选择那些估计需要执行时间最短的作业投入执行, 为它们创建进程和分配资源。有可能使得那些长作业永远得不到调度执行

- 3 响应比高者优先 ( Highest Response-ratio Next , HRN) 方式

- 响应比 $R = (W + T) / T = 1 + W / T$   
T: 为估计需要的执行时间  
W: 在后备状态队列中的等待时间  
T+W: 响应时间
- 作业调度时, 系统计算每个作业的响应比, 选择R最大者投入执行。
  - 长作业有机会获得调度执行(随着它等待时间的增加,  $W / T$ 也就随着增加)。
  - HRN的吞吐量小于SJF, 由于长作业也有机会投入运行, 在同一时间内处理的作业数显然要少于SJF法。
  - 系统开销增加: 每次调度前要计算响应比。
  - HRN是对FCFS方式和SJF方式的一种综合平衡。



# 时间片轮转(RR)算法

本算法主要用于微观调度，说明怎样并发运行，即切换的方式；设计目标是提高资源利用率。

其基本思路是通过时间片轮转，提高进程并发性和响应时间特性，从而提高资源利用率；

# 时间片轮转算法

- Ø 将系统中所有的就绪进程按照FCFS原则，排成一个队列。
- Ø 每次调度时将CPU分派给队首进程，让其执行一个时间片。时间片的长度从几个ms到几百ms。
- Ø 在一个时间片结束时，发生时钟中断。
- Ø 调度程序据此暂停当前进程的执行，将其送到就绪队列的末尾，并通过上下文切换执行当前的队首进程。
- Ø 进程可以未使用完一个时间片，就出让CPU（如阻塞）。



## 2. 时间片长度的确定

### Ø 时间片长度变化的影响

Ø 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。

Ø 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，响应时间长。

### Ø 对响应时间的要求：

Ø  $T(\text{响应时间}) = N(\text{进程数目}) * q(\text{时间片})$

### Ø 时间片长度的影响因素：

Ø 就绪进程的数目：数目越多，时间片越小（当响应时间一定时）

Ø 系统的处理能力：应当使用户输入通常在一个时间片内能处理完，否则使响应时间，平均周转时间和平均带权周转时间延长。

# 多级队列算法

本算法引入多个就绪队列，通过各队列的区别对待，达到一个综合的调度目标；

- 根据作业或进程的性质或类型的不同，将就绪队列再分为若干个子队列。
- 每个作业固定归入一个队列。
- 各队列的不同处理：不同队列可有不同的优先级、时间片长度、调度策略等。如：系统进程、用户交互进程、批处理进程等。



# 优先级算法

本算法是多级队列算法的改进，平衡各进程对响应时间的要求。适用于作业调度和进程调度，可分成抢先式和非抢先式；

静态优先级

动态优先级

线性优先级调度算法（SRR）

# 静态优先级

创建进程时就确定，直到进程终止前都不改变。

Ø依据：

Ø进程类型（系统进程优先级较高）

Ø对资源的需求（对CPU和内存需求较少的进程，优先级较高）

Ø用户要求（紧迫程度和付费多少）



# 动态优先级

在创建进程时赋予的优先级，在进程运行过程中可以自动改变，以便获得更好的调度性能。如：

- Ø 在就绪队列中，等待时间延长则优先级提高，从而使优先级较低的进程在等待足够的时间后，其优先级提高到可被调度执行；
- Ø 进程每执行一个时间片，就降低其优先级，从而一个进程持续执行时，其优先级降低到出让CPU。

# 线性优先级调度算法

本算法是优先级算法的一个实例，它通过进程优先级的递增来改进长执行时间进程的周转时间特征。

- 就绪进程队列分成两个：
  - 新创建进程队列：按FCFS方式排成；进程优先级按速率 $a$ 增加；
  - 享受服务队列：已得到过时间片服务的进程按FCFS方式排成；进程优先级按速率 $b$ 增加；
- 新创建进程等待时间的确定：当新创建进程优先级与享受服务队列中最后一个进程优先级相同时，转入享受服务队列；



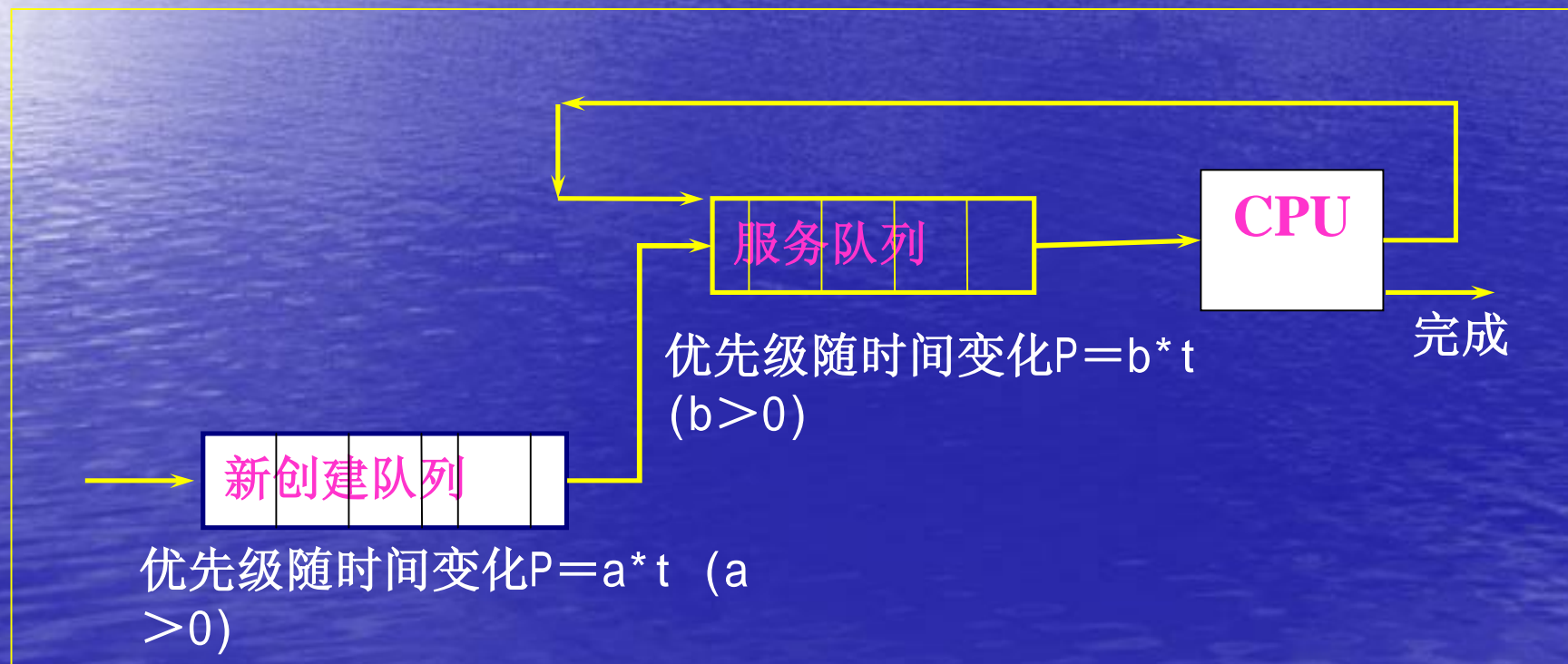
对于这两个不同队列中的进程，设新创建进程进入新创建进程就绪队列时的优先级 $P$ 为 $0$ ，进入就绪队列后以 $P=a*t$  ( $a>0$ )的速率增加。

享受服务队列中进程的优先级 $P$ 以 $P=b*t$  ( $a>b>0$ )的速率增长。

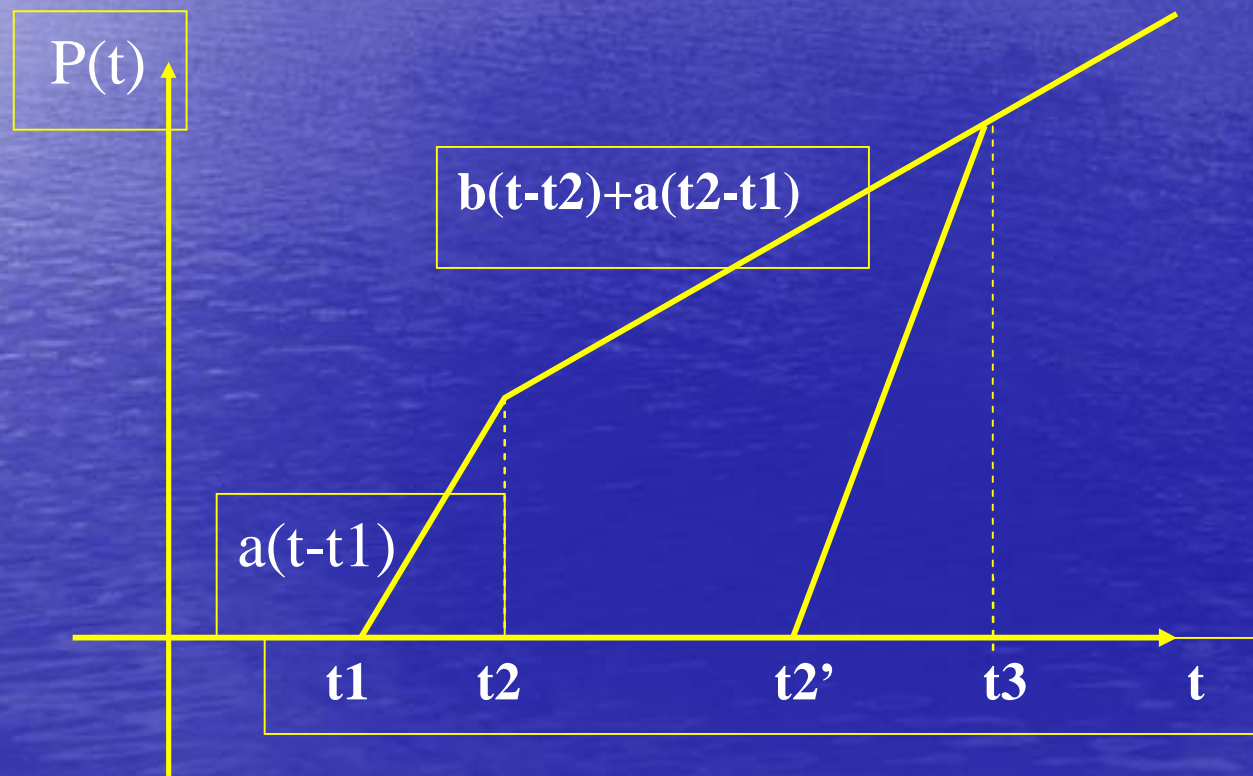
新创建进程进入享受服务队列条件：

当新创建进程就绪队列中的头一个进程的优先级 $P(t-t_1)$ 与享受服务队列中最后一个就绪进程的优先级 $P(t)=b*t$ 相等时

当享受服务进程队列为空时，新创建进程队列的头一个进程也将移入享受服务进程队列。



- 设某一进程在时刻 $t_1$ 时被创建. 在时刻 $t$ 时, 该进程的优先级为 $P(t)=a \cdot (t-t_1)$  ( $t_1 < t < t_2$ )
- 又设该进程在 $t_2$ 时刻转入享受服务队列, 则在时刻 $t$ , 该进程的优先级变为 $P(t)=a \cdot (t_2-t_1)+b \cdot (t-t_2)$  ( $t_2 < t < t_3$ )





# 多级反馈队列算法

- Ø 多级反馈队列算法是时间片轮转算法和优先级算法的综合和发展。优点：
  - Ø 为提高系统吞吐量和缩短平均周转时间而照顾短进程
  - Ø 为获得较好的I/O设备利用率和缩短响应时间而照顾I/O型进程
  - Ø 不必估计进程的执行时间，动态调节

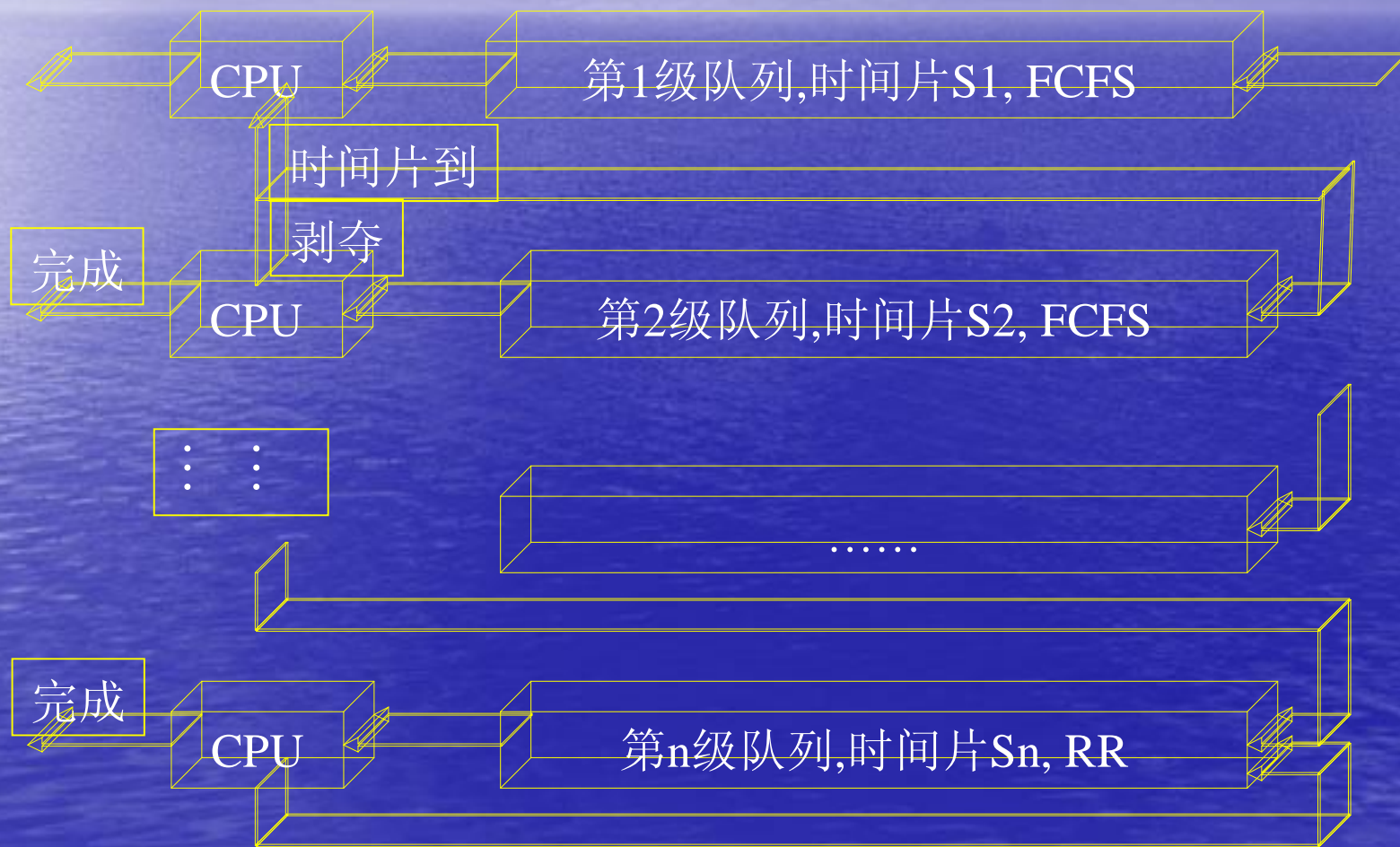
# 多级反馈队列算法

## 1. 基本思想

- ① 系统按优先级别设置 $n$ 个就绪进程队列,第一级队列优先级最高,第 $n$ 级最低
- ② 每个就绪队列对应一个时间片 $S_i (i=1,2,\dots,n)$ , 且  $S_1 < S_2 < \dots < S_n$ , 一般  $S_{i+1} = 2S_i$ ;
- ③ 除第 $n$ 级队列按RR调度外,其余各级队列按FCFS调度;
- ④ 系统每次总是调度级别较高的队列中进程, 仅当该队列为空时,才去调度下一级队列中的进程;
- ⑤ 当现行进程正在执行它的CPU周期时, 如发生时间片中断或有进程进入更高级的就绪队列时, 将引起剥夺;对前一种情况,现行进程将进入下一级队列,对后一种情况,进入本级队列末尾. 当一进程被唤醒时, 它进入原离开的那个队列,即与其当前优先级对应的就绪队列;



- I/O型进程：让其进入最高优先级队列，以及时响应I/O交互。通常执行一个时间片，要求可处理完一次I/O请求的数据，然后转入到阻塞队列。
- 计算型进程：每次都执行完时间片，进入更低级队列。最终采用最大时间片来执行，减少调度次数。
- I/O次数不多，而主要是CPU处理的进程：在I/O完成后，放回优先I/O请求时离开的队列，以免每次都回到最高优先级队列后再逐次下降。
- 为适应一个进程在不同时间段的运行特点，I/O完成时，提高优先级；时间片用完时，降低优先级；





# 调度性能比较

先来先服务方式:  $R_{fc}(k) = 1/(\mu - \lambda)$

时间轮转方式:  $R_{rr}(k) = kq\mu(\mu - \lambda)$

线性优先级方式:  $R_{rr}(k) = 1/(\mu - \lambda) - (1 - kq\mu)(\mu - \lambda')$

① 如果  $kq = 1/\mu$ , 即顾客的服务时间与其平均服务时间相等, 则

$$R_{fc}(k) = R_{rr}(k) = R_{rr}(k)$$

先来先服务调度算法、时间片轮转调度算法、线性优先级调度算法的调度性能一样。

② 如果  $kq < 1/\mu$ , 则

$$R_{rr}(k) < R_{rr}(k) < R_{fc}(k)$$

对于短作业或进程来说, 时间片轮转调度算法最好, 先来先服务调度算法最差。

③ 如果  $kq > 1/\mu$ , 则

$$R_{fc}(k) < R_{rr}(k) < R_{rr}(k)$$

对于长作业或进程来说, 先来先服务调度算法最佳, 时间片轮转调度算法最差。

# Windows 2000的线程调度概述

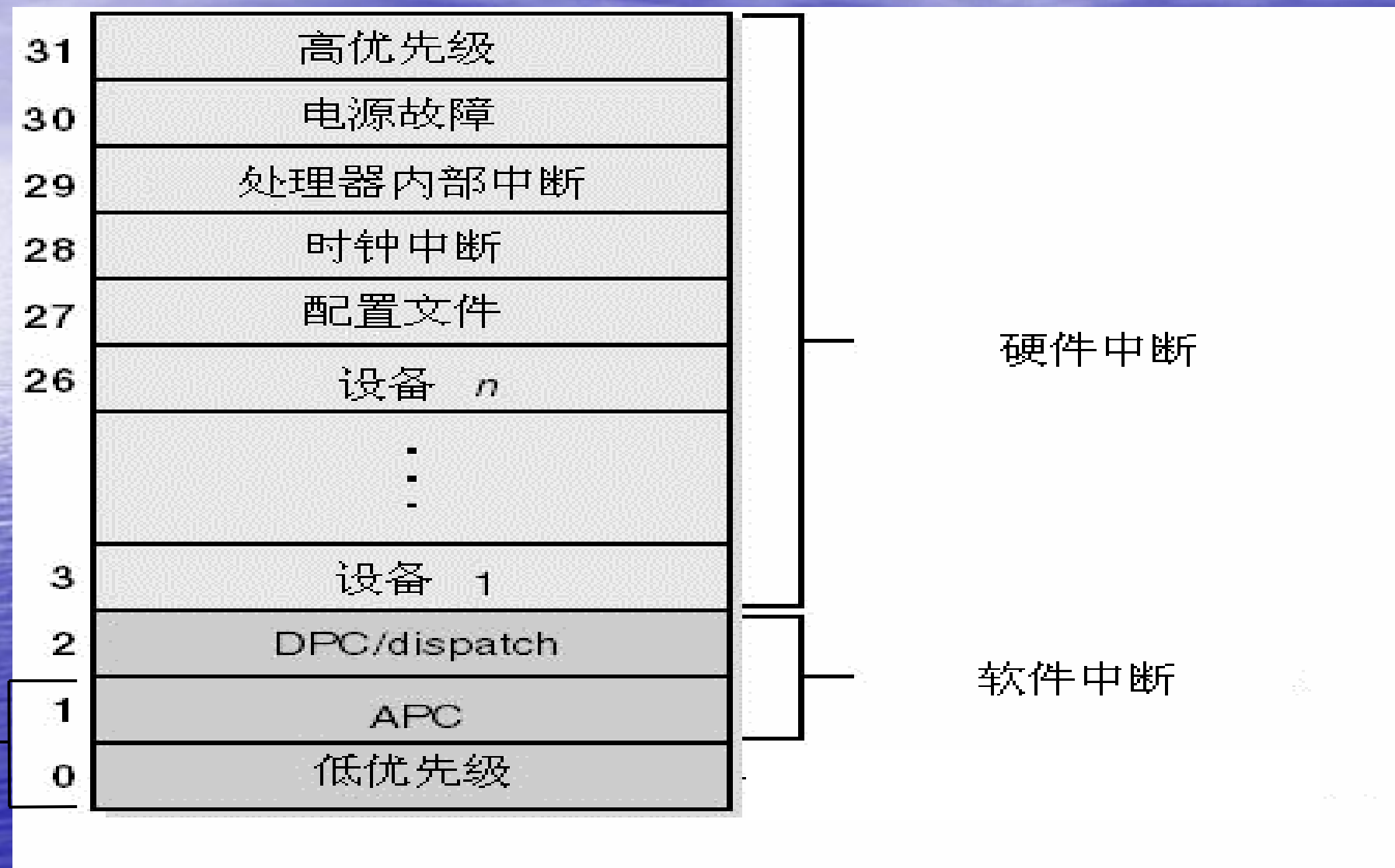
调度单位是线程而不是进程，采用严格的抢先式动态优先级调度，依据优先级和分配时间片来调度。

- 每个优先级的就绪进程排成一个先进先出队列；
- 当一个线程状态变成就绪时，它可能立即运行或排到相应优先级队列的尾部。
- 总运行优先级最高的就绪线程；



- Ø 完全的事件驱动机制，在被抢先前没有保证的运行时间；
  - Ø 没有形式的调度循环；
  - Ø 时间片用完事件；
  - Ø 等待结束事件；
  - Ø 主动挂起事件；
- Ø 在同一优先级的各线程按时间片轮转算法进行调度；
- Ø 在多处理机系统中多个线程并行运行；

# Windows 2000的中断优先级

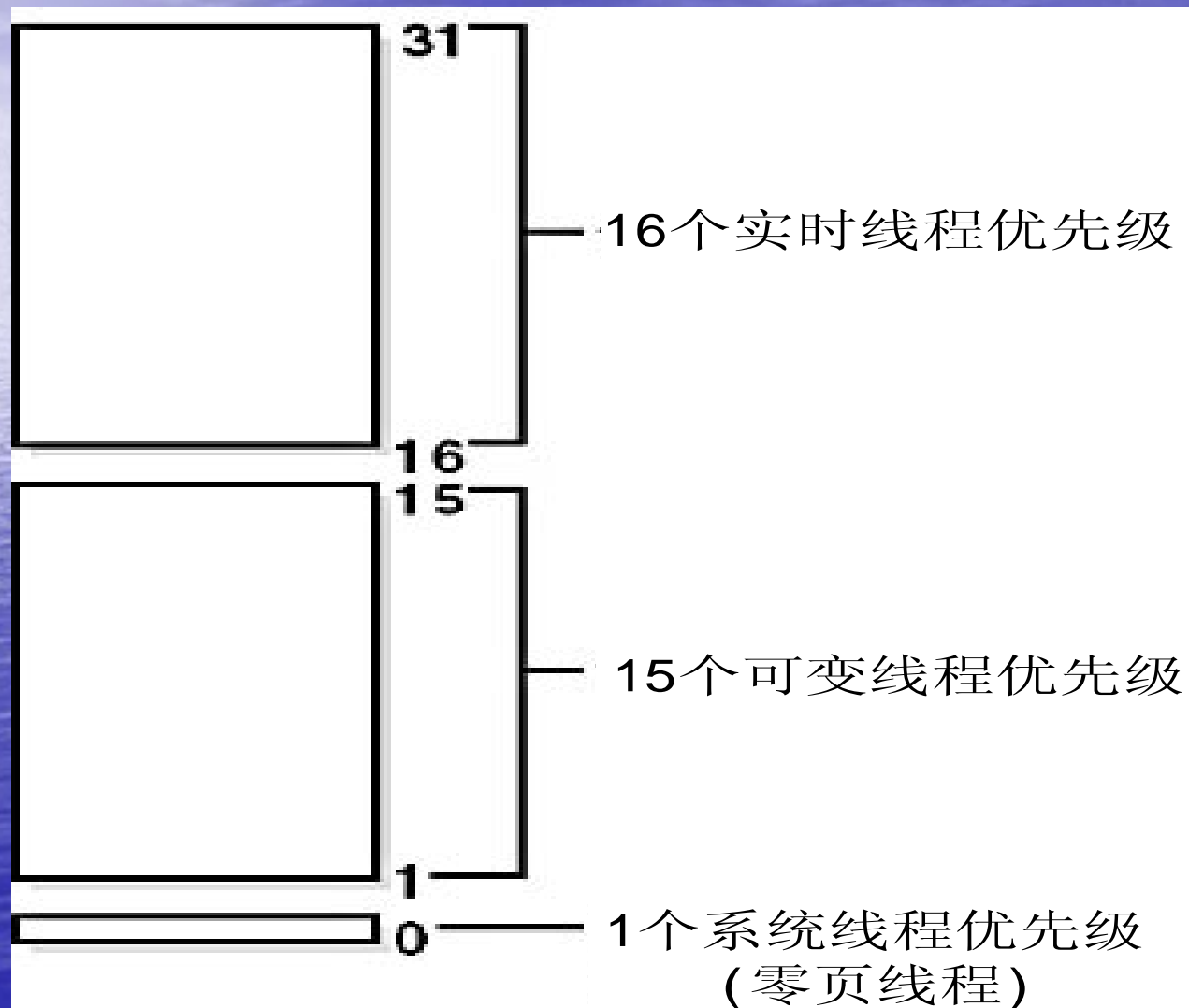




# Windows2000线程的优先级

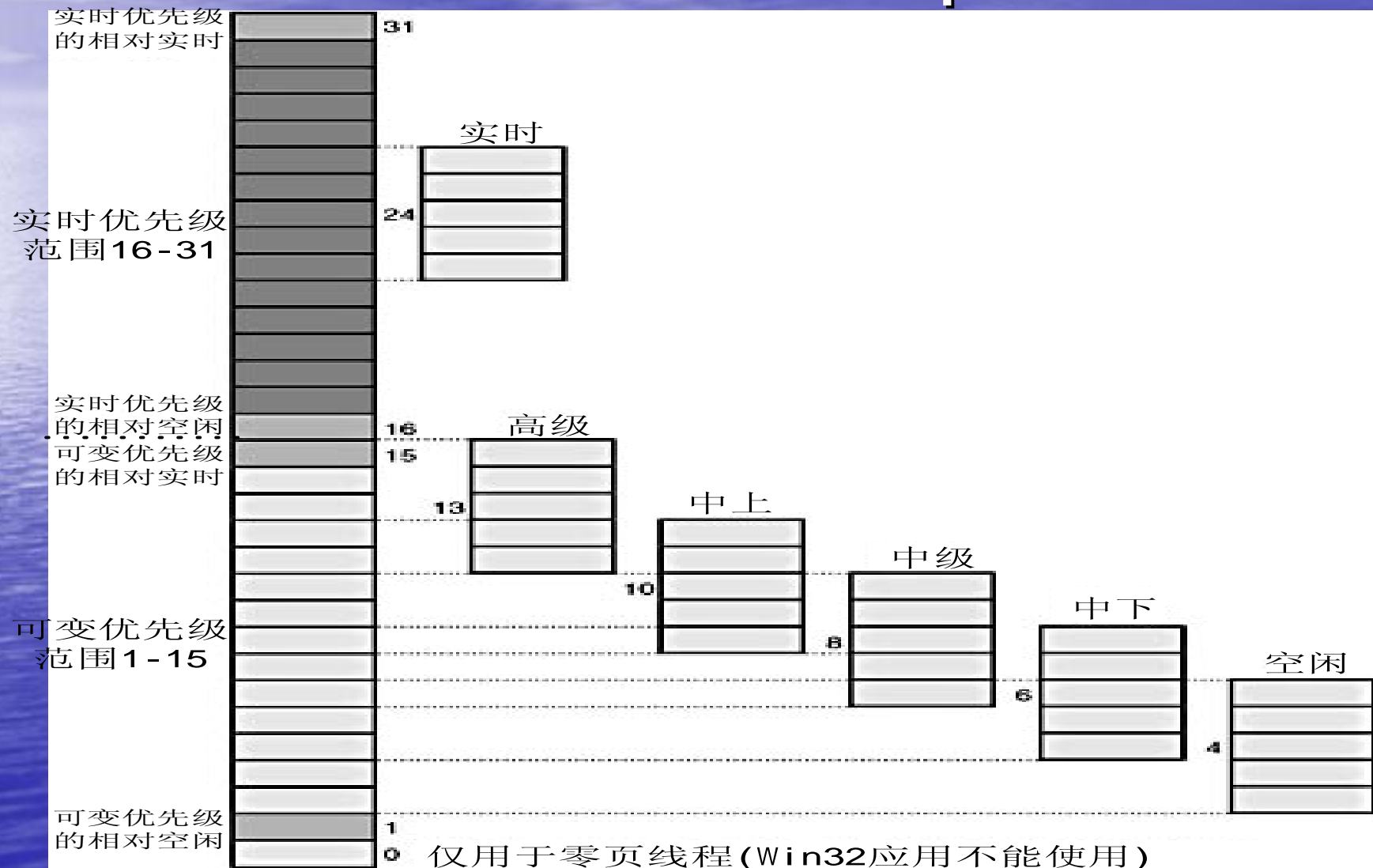
- 实时(real-time): 从16到31, 如设备监控线程。
- 可变优先级(variable-priority): 从1到15 (级别0保留为系统使用)。
  - 线程的基本优先级 = [进程的基本优先级 - 2, 进程的基本优先级 + 2], 由应用程序控制
  - 线程的动态优先级 = [进程的基本优先级 - 2, 31], 由windows2000核心控制

- Windows2000的线程优先级由进程优先级类和线程优先级偏移构成，分别由相关函数控制。

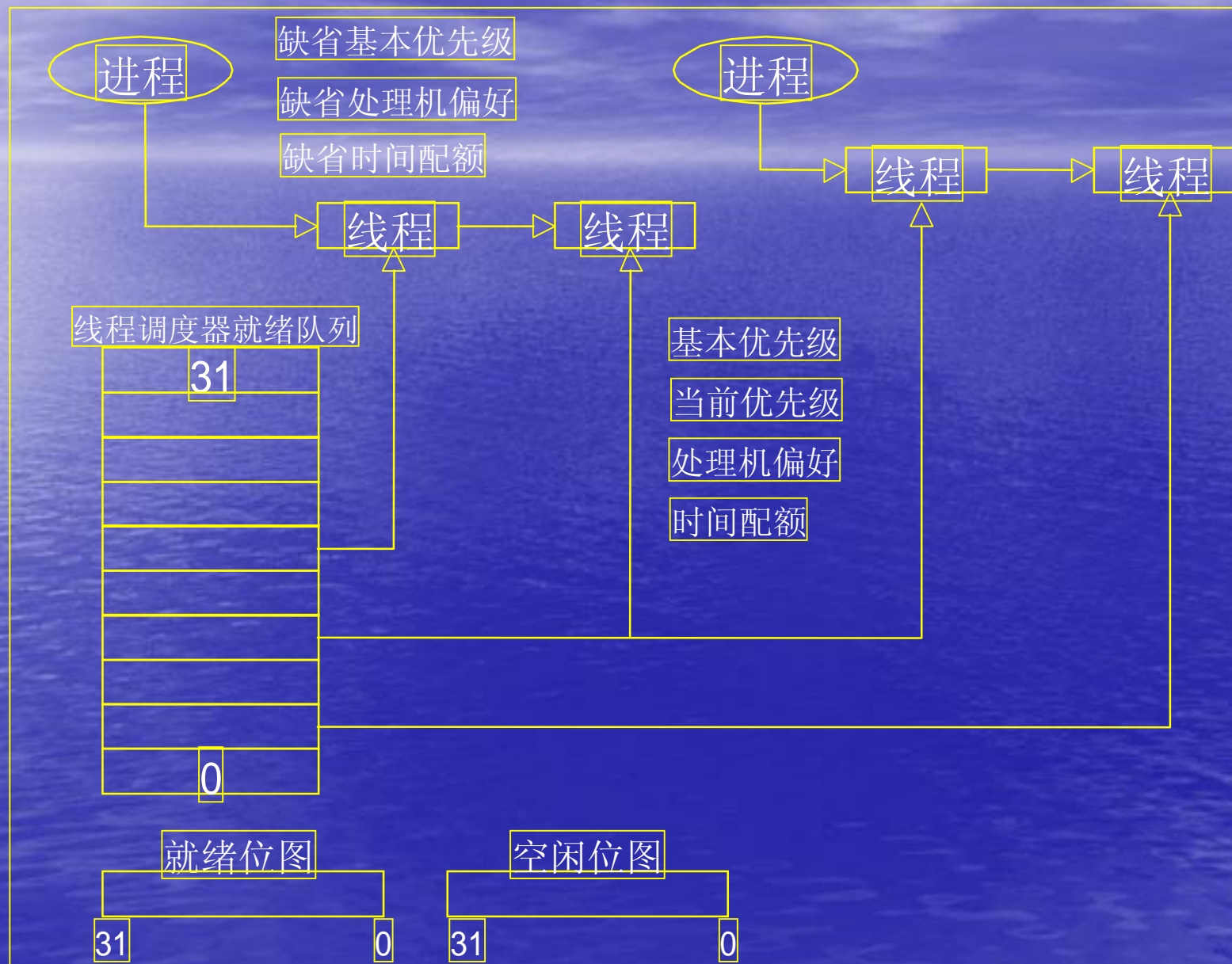




# Win32和Windows2000/xp线程优先级



# 线程调度数据结构





### Ø 就绪位图

Ø 为了提高调度速度，Windows 2000维护了一个称为就绪位图的32位量。就绪位图中的每一位指示一个调度优先级的就绪队列中是否有线程等待运行。B0与调度优先级0相对应，B1与调度优先级1相对应，等待。

### Ø 空闲位图

Ø Windows 2000还维护一个称为空闲位图的32位量。空闲位图中的每一位指示一个处理机是否处于空闲状态。

### Ø 调度器自旋锁

Ø 为了防止调度器代码与线程在访问调度器数据结构时发生冲突，处理机调度仅出现在DPC/调度层次。但在多处理机系统中，修改调度器数据结构需要额外的步骤来得到内核调度器自旋锁，以协调各处理机对调度器数据结构的访问。

# 线程时间配额

- 时间配额是一个线程从进入运行状态到Windows 2000检查是否有其他优先级相同的线程需要开始运行之间的时间总和。一个线程用完了自己的时间配额时，如果没有其它相同优先级线程，Windows 2000将重新给该线程分配一个新的时间配额，并继续运行。
- 每个线程都有一个代表本次运行最大时间长度的时间配额。时间配额不是一个时间长度值，而是一个称为配额单位的整数。



# 时间配额的计算

- Ø 缺省时，在Windows 2000专业版中线程时间配额为6；而在Windows 2000服务器中线程时间配额为36。
  - Ø 在Windows 2000服务器具备较长缺省时间配额，保证客户请求所唤醒的服务器应用有足够的时间在它的时间配额用完前完成客户的请求并回到等待状态。
- Ø 每次时钟中断，时钟中断服务例程从线程的时间配额中减少一个固定值(3)。
  - Ø 如果没有剩余的时间配额，系统将触发时间配额用完处理，选择另外一个线程进入运行状态。
  - Ø 在Windows 2000专业版中，由于每个时钟中断时减少的时间配额为3，一个线程的缺省运行时间为2个时钟中断间隔；在Windows 2000服务器中，一个线程的缺省运行时间为12个时钟中断间隔。
- Ø 如果时钟中断出现时系统正在处在DPC/线程调度层次以上(如系统正在执行一个延迟过程调用或一个中断服务例程)，当前线程的时间配额仍然要减少。甚至在时钟中断间隔期间，当前线程一条指令也没有执行，它的时间配额在时钟中断中也会被减少。

Ø 不同硬件平台的时钟中断间隔是不同的，时钟中断的频率是由硬件抽象层确定的，而不是内核确定的。

Ø 例如，大多数x86单处理机系统的时钟中断间隔为10毫秒，大多数x86多处理机系统的时钟中断间隔为15毫秒。

Ø 在等待完成时允许减少部分时间配额。

Ø 当优先级小于14的线程执行一个等待函数(如WaitForSingleObject或WaitForMultipleObjects)时，它的时间配额被减少1个时间配额单位。当优先级大于等于14的线程在执行完等待函数后，它的时间配额被重置。

Ø 这种部分减少时间配额的做法可解决线程在时钟中断触发前进入等待状态所产生的问题。

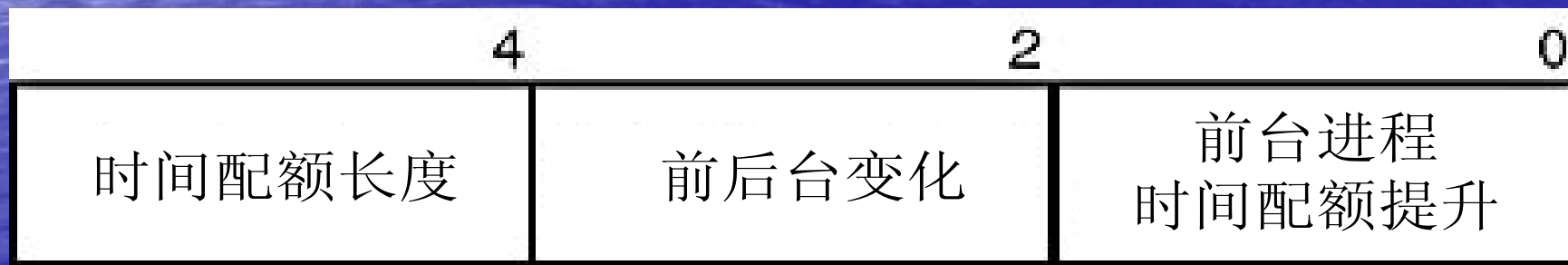
Ø 如果不进行这种部分减少时间配额操作，一个线程可能永远不减少它的时间配额。例如，一个线程运行一段时间后进入等待状态，再运行一段时间后又进入等待状态，但在时钟中断出现时它都不是当前线程，则它的时间配额永远也不会因为运行而减少。



# 时间配额的控制

在系统注册库中的一个注册项

“HKLM\SYSTEM\CurrentControlSet\Control\PriorityControl\Win32PrioritySeparation”，允许用户指定线程时间配额的相对长度(长或短)和前台进程的进程的时间配额是否加长。该注册项为6位，分成3个字段，每个字段占2位。



Ø 时间配额长度字段：

Ø 1表示长时间配额，2表示短时间配额。0或3表示缺省设置(Windows 2000专业版的缺省设置为短时间配额，Windows 2000服务器版的缺省设置为长时间配额)。

Ø 前后台变化字段：

Ø 1表示改变前台进程时间配额，2表示前后台进程的时间配额相同。0或3表示缺省设置(Windows 2000专业版的缺省设置为改变前台进程时间配额，Windows 2000服务器版的缺省设置为前后台进程的时间配额相同)。

Ø 前台进程时间配额字段：

Ø 该字段的取值只能是0、1或2(取3是非法的，被视为2)。该字段是一个时间配额表索引，用于设置前后台进程的时间配额，后台进程的时间配额为第一项，前台进程的时间配额为第二项。该字段的值保存在内核变量。



# 时间配额的设置

|              | 短时间配额 |    |    | 长时间配额 |    |    |
|--------------|-------|----|----|-------|----|----|
| 改变前台进程时间配额   | 6     | 12 | 18 | 12    | 24 | 36 |
| 前后台进程的时间配额相同 | 18    | 18 | 18 | 36    | 36 | 36 |

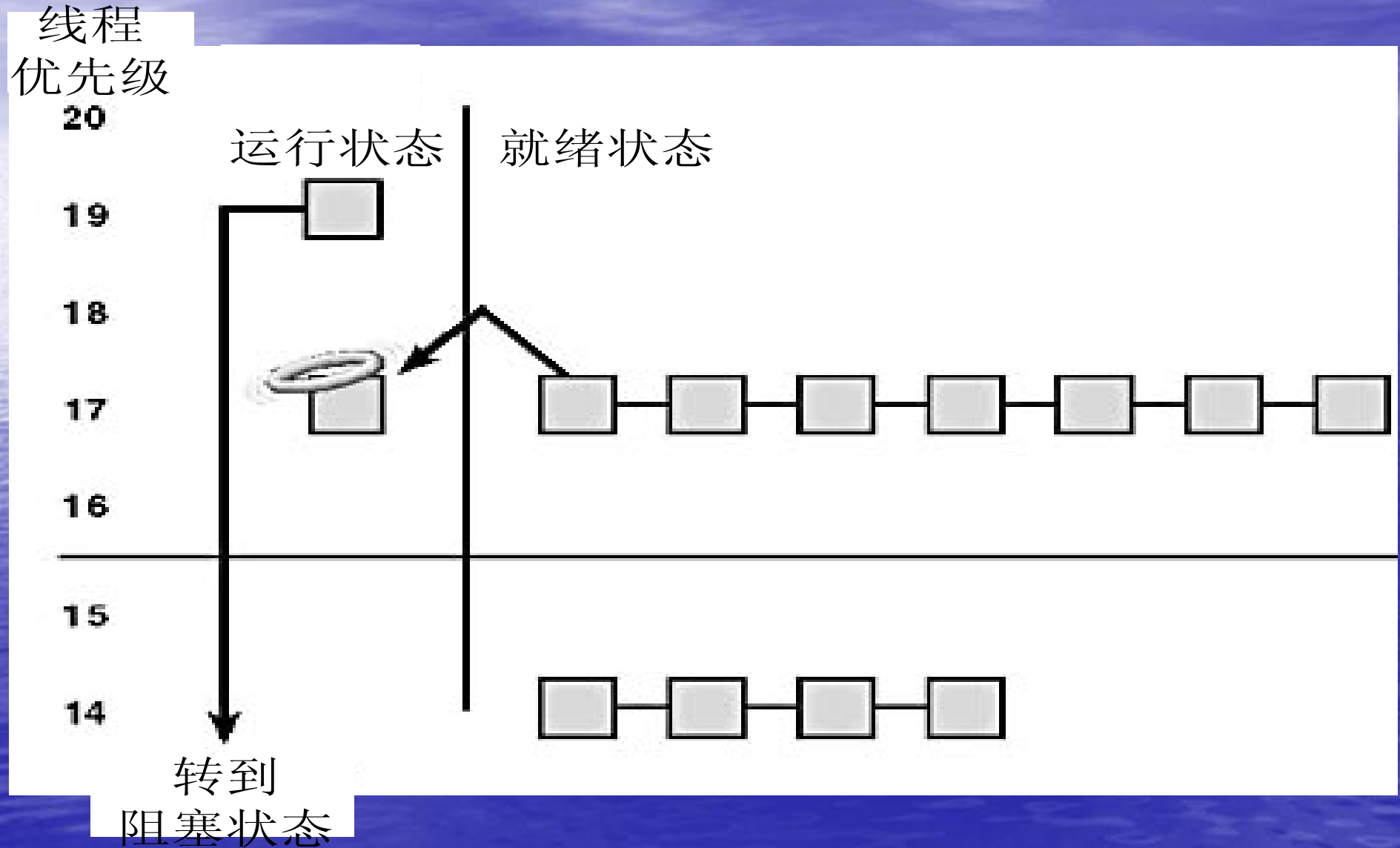
如果当前窗口切换到一个优先级高于空闲优先级类的进程中某线程，Win32子系统将用注册项 Win32PrioritySeparation 的前台进程时间配额字段作为索引，依据一个有3个元素的数组中取值，来设置该进程中所有线程的时间配额。该数组的内容由注册项 Win32PrioritySeparation 的另外2个字段确定。

# 提高前台线程优先级的潜在问题

- Ø 假设用户首先启动了一个运行时间很长的电子表格计算程序，然后切换到一个计算密集型的应用(如一个需要复杂图形显示的游戏)。
- Ø 如果前台的游戏进程提高它的优先级，后台的电子表格将会几乎得不到CPU时间。
- Ø 但增加游戏进程的时间配额，则不会停止电子表格计算的执行，只是给游戏进程的CPU时间多一些。
- Ø 如果用户希望运行一个交互式应用程序时的优先级比其他交互进程的优先级高，可利用任务管理器来修改进程的优先级类型为中上或高级，也可利用命令行在启动应用时使用命令“start /abovenormal”或“start /high”来设置进程优先级类型。



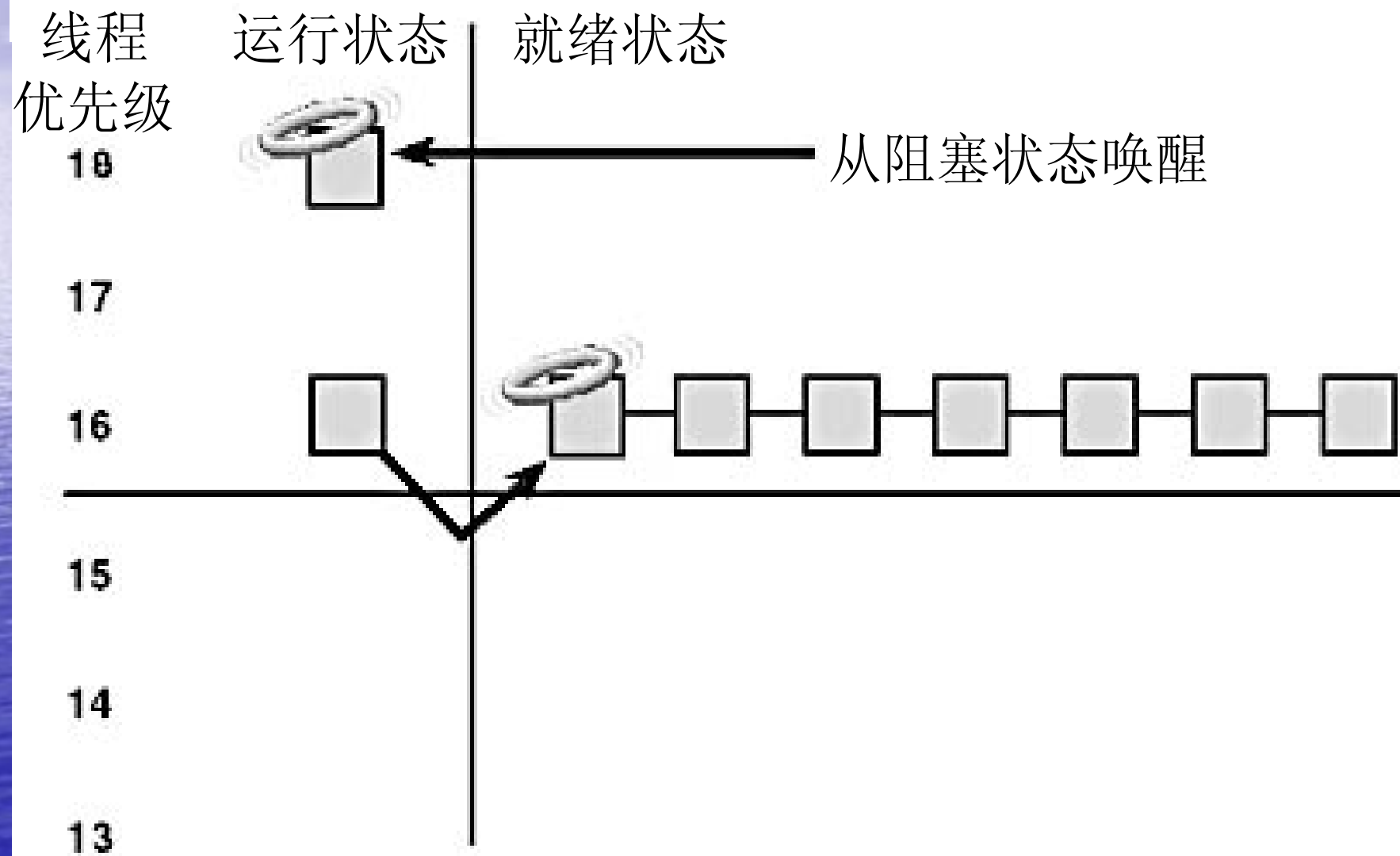
# Windows 2000线程调度-主动切换



- Ø 许多Win32等待函数调用(如WaitForSingleObject或WaitForMultipleObjects等)都使线程等待某个对象，等待的对象可能有事件、互斥信号量、资源信号量、I/O操作、进程、线程、窗口消息等。
- Ø 通常进入等待状态线程的时间配额不会被重置，而是在等待事件出现时，线程的时间配额被减1，相当于1/3个时钟间隔；如果线程的优先级大于等于14，在等待事件出现时，线程的优先级被重置。



# 抢先



Ø 可能在以下两种情况下出现抢先：

Ø 高优先级线程的等待完成，即一个线程等待的事件出现。

Ø 一个线程的优先级被增加或减少。

Ø 用户态下运行的线程可以抢先内核态下运行的线程。

Ø 在判断一个线程是否被抢先时，并不考虑线程处于用户态还是内核态，调度器只是依据线程优先级进行判断。

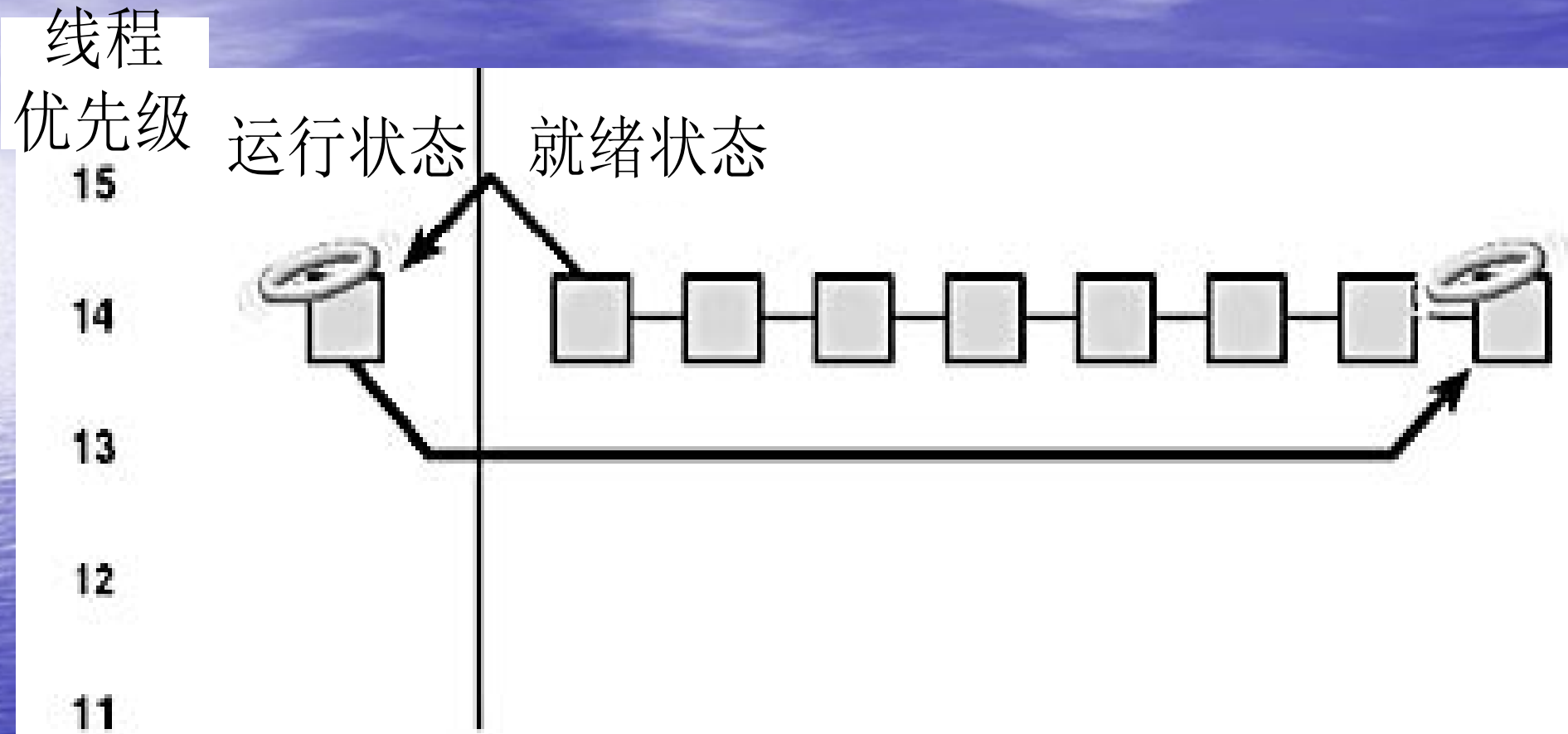
Ø 当线程被抢先时，它被放回相应优先级的就绪队列的队首。

Ø 处于实时优先级的线程在被抢先时，时间配额被重置为一个完整的时间片；

Ø 处于动态优先级的线程在被抢先时，时间配额不变，重新得到处理机使用权后将运行到剩余的时间配额用完。



# 时间片用完



线程完整用完一个规定的时间片值时，重新赋予新时间片值，优先级降一级（不低于基本优先级），放在相应优先级就绪队列的尾部；

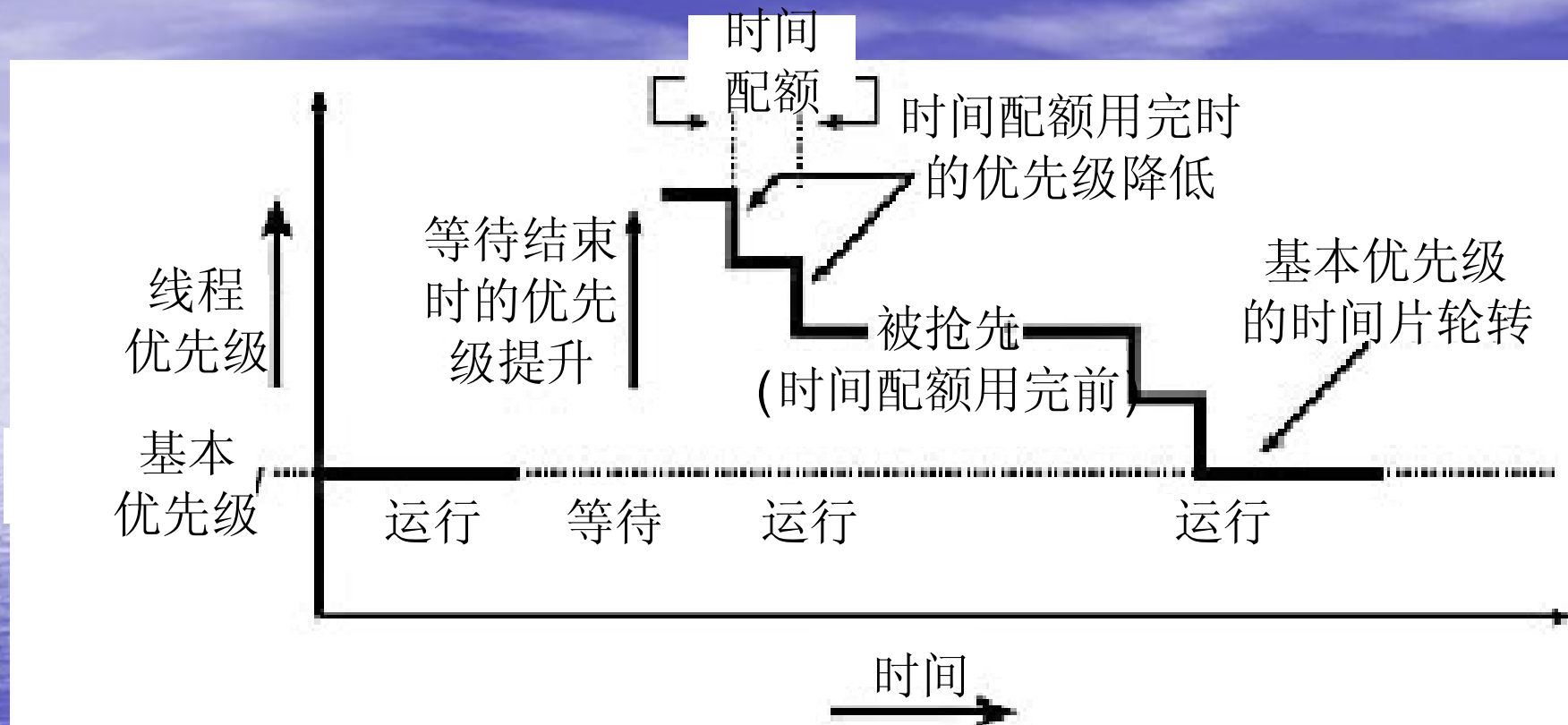
- 如果刚用完时间配额的线程优先级降低了，Windows 2000将寻找一个优先级高于刚用完时间配额线程的新设置值的就绪线程。
- 如果刚用完时间配额的线程的优先级没有降低，并且有其他优先级相同的就绪线程，Windows 2000将选择相同优先级的就绪队列中的下一个线程进入运行状态，刚用完时间配额的线程被排到就绪队列的队尾(即分配一个新的时间配额并把线程状态从运行状态改为就绪状态)。
- 如果没有优先级相同的就绪线程可运行，刚用完时间配额的线程将得到一个新的时间配额并继续运行。



# 结束

- 当线程完成运行时，它的状态从运行状态转到终止状态。线程完成运行的原因可能是通过调用ExitThread而从主函数中返回或通过被其他线程通过调用TerminateThread来终止。如果处于终止状态的线程对象上没有未关闭的句柄，则该线程将被从进程的线程列表中删除，相关数据结构将被释放。

# 优先级调整



线程由于调用等待函数而阻塞时，减少一个时间片，并依据等待事件类型提高优先级；如等待键盘事件比等待磁盘事件的提高幅度大。



Ø 在下列5种情况下，Windows 2000会提升线程的当前优先级：

Ø I/O操作完成

Ø 信号量或事件等待结束

Ø 前台进程中的线程完成一个等待操作

Ø 由于窗口活动而唤醒图形用户接口线程

Ø 线程处于就绪状态超过一定时间，但没能进入运行状态(处理机饥饿)

Ø 线程优先级提升的目的是改进系统吞吐量、响应时间等整体特征，解决线程调度策略中潜在的不公正性。但它也不是完美的，它并不会使所有应用都受益。

Ø Windows 2000永远不会提升实时优先级范围内(16至31)的线程优先级。

## I/O操作完成后的线程优先级提升

- Ø 在完成I/O操作后，保证等待I/O操作的线程能有更多的机会立即开始处理得到的结果。
- Ø 为了避免I/O操作导致对某些线程的不公平偏好，在I/O操作完成后唤醒等待线程时将把该线程的时间配额减1。
- Ø 设备驱动程序在完成I/O请求时通过内核函数 `IoCompleteRequest` 来指定优先级提升的幅度。
- Ø 线程优先级的提升幅度与I/O请求的响应时间要求是一致的，响应时间要求越高，优先级提升幅度越大。



## 线程优先级提升的建议值

| 设备             | 优先级提升值 |
|----------------|--------|
| 磁盘、光驱、并口、视频    | 1      |
| 网络、邮件槽、命名管道、串口 | 2      |
| 键盘、鼠标          | 6      |
| 音频             | 8      |
|                |        |

- ❌ 线程优先级提升是以线程的基本优先级为基点的，不是以线程的当前优先级为基点。
- ❌ 当用完它的一个时间配额后，线程会降低一个优先级，并运行另一个时间配额。这个降低过程会一直进行下去，直到线程的优先级降低至原来的基本优先级。
- ❌ 优先级提升策略仅适用于可变优先级范围(0到15)内的线程。不管线程的优先级提升幅度有多大，提升后的优先级都不会超过15而进入实时优先级。



# 等待事件和信号量后的线程优先级提升

- Ø 当一个等待执行事件对象或信号量对象的线程完成等待后，它的优先级将提升一个优先级。
- Ø 阻塞于事件或信号量的线程得到的处理机时间比处理机繁忙型线程要少，这种提升可减少这种不平衡带来的影响。
- Ø 释放信号量函数调用可导致事件对象或信号量对象等待的结束。
- Ø 在等待结束时，线程的时间配额被减1，并在提升后的优先级上执行完剩余的时间配额；随后降低1个优先级，运行一个新的时间配额，直到优先级降低到初始的基本优先级。

## 前台线程在等待结束后的优先级提升

- Ø 对于前台进程中的线程，一个内核对象上的等待操作完成时，内核函数会提升线程的当前优先级(非基本优先级)，提升幅度为相应取值。
- Ø 在前台应用完成它的等待操作时小幅提升它的优先级，以使它更有可能马上进入运行状态，有效改进前台应用的响应时间特征。
- Ø 用户不能禁止这种优先级提升，甚至是在用户已利用Win32的函数禁止了其他的优先级提升策略时，也是如此。



## 图形用户接口线程被唤醒后的优先级提升

- Ø 拥有窗口的线程在被窗口活动唤醒(如收到窗口消息)时将得到一个幅度为2的额外优先级提升。
- Ø 窗口系统在调用函数SetEvent时实施这种优先级提升，SetEvent函数调用设置一个事件，用于唤醒一个图形用户接口线程。
- Ø 这种优先级提升的原因是改进交互应用的响应时间。



# 对处理机饥饿线程的优先级提升

- Ø 系统线程平衡集管理器会每秒钟检查一次就绪队列，是否存在一直在就绪队列中排队超过300个时钟中断间隔的线程。
- Ø 如果找到这样的线程，平衡集管理器将把该线程的优先级提升到15，并分配给它一个长度为正常值两倍的时间配额；
- Ø 当被提升线程用完它的时间配额后，该线程的优先级立即衰减到它原来的基本优先级。

- Ø 如果在该线程结束前出现其他高优先级的就绪线程，该线程会被放回就绪队列，并在就绪队列中超过另外300个时钟中断间隔后再次被提升优先级。
- Ø 平衡集管理器只扫描16个就绪线程。如果就绪队列中有更多的线程，它将记住暂停时的位置，并在下一次开始时从当前位置开始扫描。
- Ø 平衡集管理器在每次扫描时最多提升10个线程的优先级。如果在一次扫描中已提升了10个线程的优先级，平衡集管理器会停止本次扫描，并在下一次开始时从当前位置开始扫描。
- Ø 这种算法并不能解决所有优先级倒置的问题，但它很有效。



# 对称多处理机系统上Windows 2000的线程调度

当Windows 2000试图调度优先级最高的可执行线程时，有几个因素会影响到处理机的选择。Windows 2000只保证一个优先级最高的线程处于运行状态。

## Ø亲合关系

- Ø描述该线程可在哪些处理机上运行。

- Ø线程的亲合掩码是从进程的亲合掩码继承得到的。

- Ø缺省时，所有进程(即所有线程)的

- Ø亲合掩码为系统中所有可用处理机的集合。应用程序通过调用相应函数来指定亲合掩码；

Ø 首选处理机：线程运行时的偏好处理机；

Ø 线程创建后，Windows 2000系统不会修改线程的首选处理机设置；

Ø 应用程序可通过SetThreadIdealProcessor函数来修改线程的首选处理机。

Ø 第二处理机：线程第二个选择的运行处理机；



# 就绪线程的运行处理机选择

- Ø 当线程进入运行状态时，Windows 2000首先试图调度该线程到一个空闲处理机上运行。如果有多个空闲处理机，线程调度器的调度顺序为：
  - Ø 线程的首选处理机
  - Ø 线程的第二处理机
  - Ø 当前执行处理机(即正在执行调度器代码的处理机)。
  - Ø 如果这些处理机都不是空闲的，Windows 2000将依据处理机标识从高到低扫描系统中的空闲处理机状态，选择找到的第一个空闲处理机。

Ø 如果线程进入就绪状态时，所有处理机都处于繁忙状态，Windows 2000将检查一个处于运行状态或备用状态的线程，判断它是否可抢先。检查的顺序如下：

Ø 线程的首选处理机

Ø 线程的第二处理机

Ø 如果这两个处理机都不在线程的亲合掩码中，Windows 2000将依据活动处理机掩码选择该线程可运行的编号最大的处理机。

Ø Windows 2000并不检查所有处理机上的运行线程和备用线程的优先级，而仅仅检查一个被选中处理机上的运行线程和备用线程的优先级。

Ø 如果在被选中的处理机上没有线程可被抢先，则新线程放入相应优先级的就绪队列，并等待调度执行。



# 为特定的处理机调度线程

- Ø 在多台处理机系统，Windows 2000不能简单地  
从就绪队列中取第一个线程，它要在亲和掩码  
限制下寻找一个满足下列条件之一的线程。
  - Ø 线程的上一次运行是在该处理机上；
  - Ø 线程的首选处理机是该处理机；
  - Ø 处于就绪状态的时间超过2个时间配额；
  - Ø 优先级大于等于24；
- Ø 如果Windows 2000不能找到满足要求的线  
程，它将从就绪队列的队首取第一个线程进入  
运行状态。

## 最高优先级就绪线程可能不处于运行状态

- 有可能出现这种情况，一个比当前正在运行线程优先级更高的线程处于就绪状态，但不能立即抢先当前线程，进入运行状态。

例如：假设0号处理机上正运行着一个可在任何处理机上运行的优先级为8的线程，1号处理机上正运行着一个可在任何处理机上运行的优先级为4的线程；这时一个只能在0号处理机上运行的优先级为6的线程进入就绪状态。

在这种情况下，优先级为6的线程只能等待0号处理机上优先级为8的线程结束。因为Windows 2000不会为了让优先级为6的线程在0号处理机上运行，而把优先级为8的线程从0号处理机移到1号处理机。即0号处理机上的优先级为8的线程不会抢先1号处理机上优先级为4的线程。



# 空闲线程

- Ø 如果在一个处理机上没有可运行的线程，Windows 2000/XP会调度相应处理机对应的空闲线程。
- Ø 由于在多处理机系统中可能两个处理机同时运行空闲线程，所以系统中的每个处理机都有一个对应的空闲线程。
- Ø Windows 2000/XP给空闲线程指定的线程优先级为0，该空闲线程只在没有其他线程要运行时才运行。

# 空闲线程的功能

- Ø 空闲线程的功能就是在一个循环中检测是否有要进行的工作。其基本的控制流程如下：
  - Ø 处理所有待处理的中断请求。
  - Ø 检查是否有待处理的DPC请求。如果有，则清除相应软中断并执行DPC。
  - Ø 检查是否有就绪线程可进入运行状态。如果有，调度相应线程进入运行状态。
  - Ø 调用硬件抽象层的处理机空闲例程，执行相应的电源管理功能。