答：首先得知道什么是**进程**什么是**线程**？

我的理解是**进程**是指在系统中正在运行的一个应用程序；程序一旦运行就是**进程**，或者更专业化来说：**进程**是指程序执行时的一个实例。

**线程**是**进程**的一个实体。

**进程**——资源分配的最小单位，**线程**——程序执行的最小单位。

**线程进程**的区别体现在几个方面：

第一：因为**进程**拥有独立的堆栈空间和数据段，所以每当启动一个新的**进程**必须分配给它独立的地址空间，建立众**多**的数据表来维护它的代码段、堆栈段和数据段，这对于**多进程**来说十分“奢侈”，系统开销比较大，而**线程**不一样，**线程**拥有独立的堆栈空间，但是共享数据段，它们彼此之间使用相同的地址空间，共享大部分数据，比**进程**更节俭，开销比较小，切换速度也比**进程**快，效率高，但是正由于**进程**之间独立的特点，使得**进程**安全性比较高，也因为**进程**有独立的地址空间，一个**进程**崩溃后，在保护模式下不会对其它**进程**产生影响，而**线程**只是一个**进程**中的不同执行路径。一个**线程**死掉就等于整个**进程**死掉。

第二：体现在通信机制上面，正因为**进程**之间互不干扰，相互独立，**进程**的通信机制相对很复杂，譬如管道，信号，消息队列，共享内存，套接字等通信机制，而**线程**由于共享数据段所以通信机制很方便。。

3.属于同一个**进程**的所有**线程**共享该**进程**的所有资源，包括文件描述符。而不同过的**进程**相互独立。

4.**线程**又称为轻量级**进程**，**进程**有**进程**控制块，**线程**有**线程**控制块；

5.**线程**必定也只能属于一个**进程**，而**进程**可以拥有**多**个**线程**而且至少拥有一个**线程**；

第四：体现在程序结构上，举一个简明易懂的列子：当我们使用**进程**的时候，我们不自主的使用if else嵌套来判断pid，使得程序结构繁琐，但是当我们使用**线程**的时候，基本上可以甩掉它，当然程序内部执行功能单元需要使用的时候还是要使用，所以**线程**对程序结构的改善有很大帮助。

**进程**与**线程**的选择取决以下几点：

1、需要频繁创建销毁的优先使用**线程**；因为对**进程**来说创建和销毁一个**进程**代价是很大的。

2、**线程**的切换速度快，所以在需要大量计算，切换频繁时用**线程**，还有耗时的操作使用**线程**可提高应用程序的响应

3、因为对CPU系统的效率使用上**线程**更占优，所以可能要发展到**多**机分布的用**进程**，**多**核分布用**线程**；

4、并行操作时使用**线程**，如C/S[架构](http://lib.csdn.net/base/architecture" \o "大型网站架构知识库" \t "_blank)的服务器端并发**线程**响应用户的请求；

5、需要更稳定安全时，适合选择**进程**；需要速度时，选择**线程**更好。

进程：计算机上所有可运行的软件，也包括操作系统，被组织成若干顺序进程

（PC， register， 地址空间， 虚拟CPU）

四种主要事件导致进程的创建：

（1）系统初始化

（2）执行了正在运行的进程所调用的进程创建系统调用

（3）用户请求创建一个新进程

（4）一个批处理作业的初始化

前台进程：与用户交互优先级别较高

后台进程：依赖于终端，不与用户交互（等待特定请求发生）

前台进程—》后台进程（fg）

后台进程—》前台进程（ctrl + z）

守护进程：LINUX后台进程也叫守护进程（Daemon），是运行在后台的一种特殊进程。它独立于控制终端并且周期性地执行某种任务或等待处理某些发生的事件。一般用作系统服务，可以用crontab提交，编辑或者删除相应得作业。守护的意思就是不受终端控制。Linux的大多数服务器就是用守护进程实现的。比如Internet服务器inetd，Web服务器httpd等。同时，守护进程完成许多系统任务。比如，作业规划进程crond，打印进程lpd等

进程终结引起条件：

1）自愿退出

2）出错退出

3）严重错误

4）被其他进程kill

进程不能继续运行，被阻塞；

还可能的原因：进程挂起由于自己的原因（在输入用户命令行之前，无法完成命令）

系统技术上的原因（没有足够CPU，所以不能时每个进程都有它私用的处理器）

进程的状态：

1）运行态 （进程占用CPU）

2）就绪态 （可运行，但因为其他进程在运行而暂时停止）

3）阻塞态 （除非某种外部事件发生，否则进程不能运行）

进程的实现：

为了实现进程 模型，操作系统维护着一张**进程表**， 一个进程占用一个进程表项（PCB）。

进程表：PC，堆栈指针， 内存分配状况， 所打开文件的状态，账号和调度信息

struct task\_struct

{

// 下面是任务（进程）数据结构，或称为进程描述符。

// long state 任务的运行状态（-1不可运行，0可运行(就绪)，>0已停止）。

// long counter 任务运行时间计数(递减)（滴答数），运行时间片。

// long priority 优先数。任务开始运行时counter=priority，越大运行越长。

// long signal 信号位图，每个比特位代表一种信号，信号值=位偏移值+1。

// struct sigaction sigaction[32] 信号执行属性结构，对应信号将要执行的操作和标志信息。

// long blocked 进程信号屏蔽码（对应信号位图）。

// -------------------

// int exit\_code 任务执行停止的退出码，其父进程会取。

// unsigned long start\_code 代码段地址。

// unsigned long end\_code 代码长度（字节数）。

// unsigned long end\_data 代码长度 + 数据长度（字节数）。

// unsigned long brk 总长度（字节数）。

// unsigned long start\_stack 堆栈段地址。

// long pid 进程标识号(进程号)。

// long pgrp 进程组号。

// long session 会话号。

// long leader 会话首领。

// int groups[NGROUPS] 进程所属组号。一个进程可属于多个组。

// task\_struct \*p\_pptr 指向父进程的指针。

// task\_struct \*p\_cptr 指向最新子进程的指针。

// task\_struct \*p\_ysptr 指向比自己后创建的相邻进程的指针。

// task\_struct \*p\_osptr 指向比自己早创建的相邻进程的指针。

// unsigned short uid 用户标识号（用户id）。

// unsigned short euid 有效用户id。

// unsigned short suid 保存的用户id。

// unsigned short gid 组标识号（组id）。

// unsigned short egid 有效组id。

// unsigned short sgid 保存的组id。

// long timeout 内核定时超时值。

// long alarm 报警定时值（滴答数）。

// long utime 用户态运行时间（滴答数）。

// long stime 系统态运行时间（滴答数）。

// long cutime 子进程用户态运行时间。

// long cstime 子进程系统态运行时间。

// long start\_time 进程开始运行时刻。

// struct rlimit rlim[RLIM\_NLIMITS] 进程资源使用统计数组。

// unsigned int flags; 各进程的标志，在下面第149行开始定义（还未使用）。

// unsigned short used\_math 标志：是否使用了协处理器。

// ------------------------

// int tty 进程使用tty终端的子设备号。-1表示没有使用。

// unsigned short umask 文件创建属性屏蔽位。

// struct m\_inode \* pwd 当前工作目录i节点结构指针。

// struct m\_inode \* root 根目录i节点结构指针。

// struct m\_inode \* executable 执行文件i节点结构指针。

// struct m\_inode \* library 被加载库文件i节点结构指针。

// unsigned long close\_on\_exec 执行时关闭文件句柄位图标志。（参见include/fcntl.h）

// struct file \* filp[NR\_OPEN] 文件结构指针表，最多32项。表项号即是文件描述符的值。

// struct desc\_struct ldt[3] 局部描述符表。0-空，1-代码段cs，2-数据和堆栈段ds&ss。

// struct tss\_struct tss 进程的任务状态段信息结构。

}

进程表中字段：

进程管理： 寄存器 pc 程序状态字 堆栈指针 进程状态 优先级 调度参数

PID 父进程 信号 进程开始时间 使用的CPU时间 子进程时间

下次报警时间

存储管理： 正文段指针 数据段指针 堆栈段指针

文件管理：根目录 工作目录 文件描述符（FD） 用户ID 组ID

中断向量：中断服务程序的入口地址，在计算机中中断向量的地址存放一条跳转到中断服务程序的跳转指令

中断地址是存储中断向量的内存单元。

中断向量表：用来存放中断向量（共256个），它的地址范围是0~3FFH

中断号：

微机通过中断号对应中断向量值，再通过中断向量的值跳转到中断服务程序。

与每一个I/O类关联的是一个称为**中断向量（interrupt vector）** 的位置（靠近内存底部的固定区域）

内存分布图（32位4GB）

0xffff ffff

内核空间

0xcfff ffff

栈（往下生长）

共享区域（动态链接库， 共享内存）

堆（往上生长）

全局区

BSS段

数据段

代码段

0x0000 000

**线程**

每一个进程都有一个地址空间和一个控制线程。

使用线程的原因：

（1）应用中同时发生许多不同的活动，其中某些活动会被阻塞，通过将这些应用程序分解成并行运行的顺序线程

（2）线程比进程的更加轻量级

（3）处理大量的I/O操作与计算，多线程能够提高应用程序的执行速度

web常用模型：

模型 特性

多线程 并行性，阻塞系统调用

单线程进程 无并行性， 阻塞系统调用

有限状态机 并行性，非阻塞系统调用，中断

经典多线程模型

进程模型基于两种独立概念：资源分组处理和执行

进程是将资源集中在一起，而线程是CPU上被调度的实体

线程中的内容 每个线程中的内容

地址空间 PC

全局变量 register

打开文件 堆栈

子进程 状态

即将发生的报警

信号与信号处理程序

账号信息

线程的状态：运行，阻塞，就绪，终止

线程的堆栈：存放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

thread\_create(func) : 创建一个线程，返回 一个线程标识符就是新线程的名字

thread\_exit() : 线程退出

thread\_yield（） ： 允许线程放弃CPU而让其他线程运行，由于无法使用时钟中断强制中断线程让出CPU

多线程与fork（）

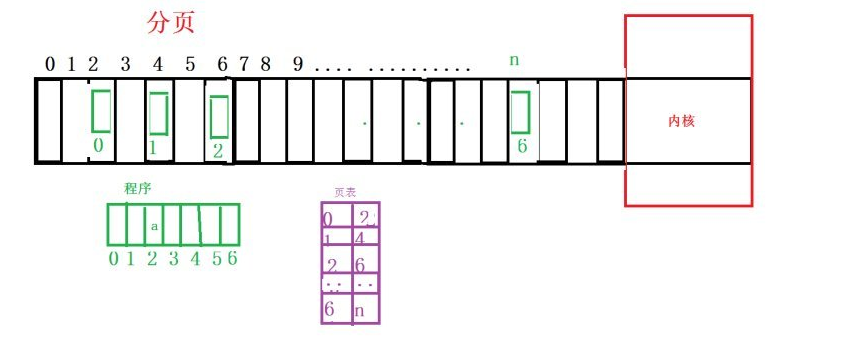
：但却各自拥有不同的栈段、数据段以及堆栈拷贝。子进程的栈、数据以及栈段开始时是父进程内存相应各部分的完全拷贝，因此它们互不影响.

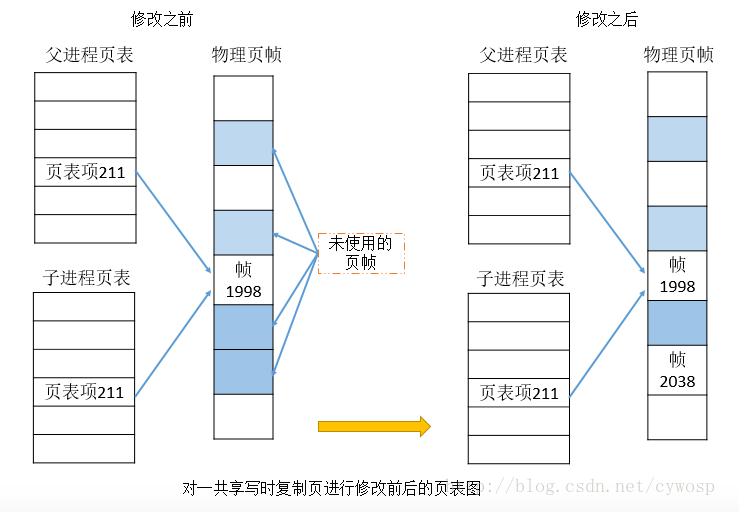
以下是写实拷贝技术

分页加载机制就是linux把内存划分成很多的小块，进程中的数据按照某种对应关系（页表）放入内存中

**fork（）实际拷贝**

如果子进程只是对父进程的数据进行读取操作，那么子进程用的就是父进程的数据。如果子进程需要对某数据进行修改，那么在修改前，子进程才会拷贝出需要修改的这份数据，对这份备份进行修改。这就满足了父子进程的数据相互独立，互不影响的要求。这么做的初衷也是为了节省内存。





1. 虽然只将发起fork()调用的线程复制到子进程中，但全局变量的状态以及所有的pthreads对象（如互斥量、条件变量等）都会在子进程中得以保留，这就造成一个危险的局面

fork()函数被调用之后，子进程就相当于处于signal handler之中，此时就不能调用线程安全的函数（用锁机制实现安全的函数），除非函数是可重入的，而只能调用异步信号安全（async-signal-safe）的函数。fork()之后，子进程不能调用：

处理操作：

推荐在多线程程序中调用fork()的唯一情况是：其后立即调用exec()函数执行另一个程序，彻底隔断子进程与父进程的关系。由新的进程覆盖掉原有的内存，使得子进程中的所有pthreads对象消失。

#include <pthread.h>

// Upon successful completion, pthread\_atfork() shall return a value of zero; otherwise, an error number shall be returned to indicate the error.

// @prepare 新进程产生之前被调用

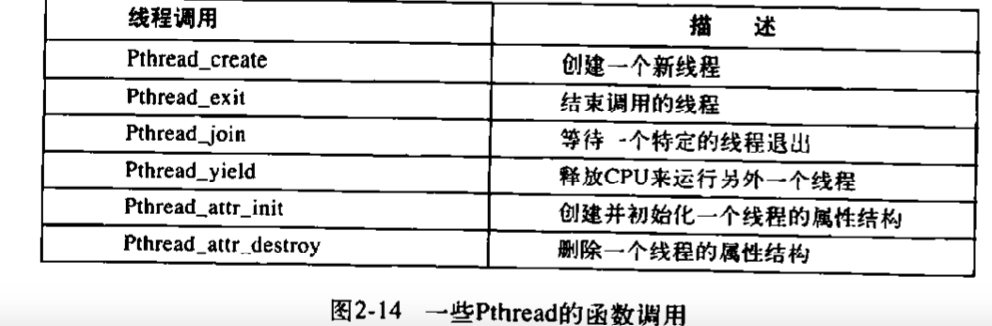
// @parent  新进程产生之后在父进程被调用

// @child    新进程产生之后，在子进程被调用

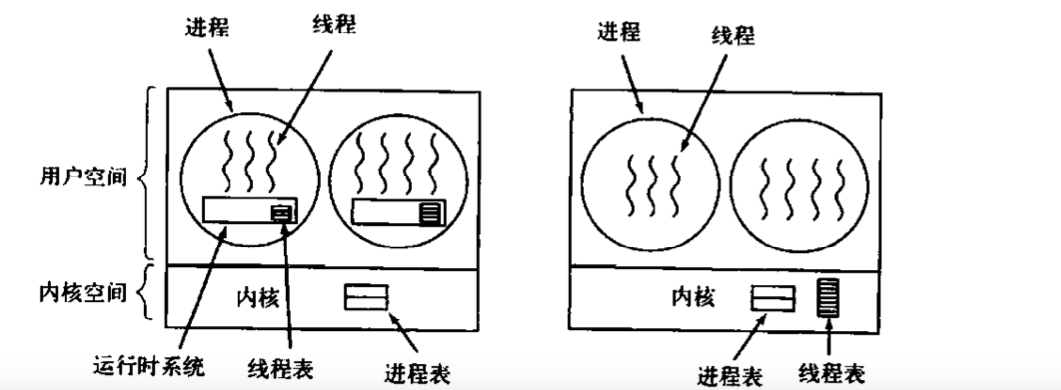
int pthread\_atfork (void (\*prepare) (void), void (\*parent) (void), void (\*child) (void));

该函数的作用就是往进程中注册三个函数，以便在不同的阶段调用，有了这三个参数，我们就可以在对应的函数中加入对应的处理功能

**POSIX线程**



Pthread\_attr\_init:设置代码的优先级

实现线程包的两种方法：在用户空间和在内核

用户空间的优点：线程的切换相对于陷入内核调用快了一个数量级，线程之间的切换和调度程序都只是本地过程，因此比进行内核调用的效率高；不需要陷阱，不需要上下文切换，不需要内存高速缓存就行刷新；运行每个进程都有自己定制的调度算法

问题：如何实现阻塞系统调用

内核实现线程：

通过系统调用对线程表的更新，完成线程的创建和撤销工作

内核线程保存了每个线程的寄存器，状态和其他信息。

内核还维护传统线程表，用来跟踪进程的状态

与用户线程相比：

内核：当一个线程阻塞时，内核会调度其他就绪线程

用户：一直运行自己进程的线程直到内核剥夺它的 CPU为止

缺点：系统调用的开销大，线程操作如果多，会带来很大的开销

question：为一个多线程的进程创建新进程时？

Question2： 如果多个线程注册同一个信号？

如果一个多线程的进程得到了信号，它是会在它诸多子线程里面选一个来执行

混合（用户级和内核级）实现：

内核级线程被多个用户级线程多路复用

### 一对一模型

#### 特点：

有了内核线程，每个用户线程被映射或绑定到一个内核线程。用户线程在其生命期内都会绑定到该内核线程。一旦用户线程终止，两个线程都将离开系统。这被称作"一对一"线程映射。（反过来，一个内核线程不一定就会对应一个用户线程）。一般一直使用API或者是系统调用创建的线程均为一对一线程。例如，linux使用clone创建的线程，以及win下使用CreateThread创建的线程。

#### 弊端:

内核线程数量有限.  
许多操作系统内核线程调用的时候，上下文切换的开销很大。

### 混合线程模型

* 混合线程实现是用户线程和内核线程的交叉，使得库和操作系统都可以管理线程。用户线程由运行时库调度器管理，内核线程由操作系统调度器管理。在这种实现中，进程有着自己的内核线程池。可运行的用户线程由运行时库分派并标记为准备好执行的可用线程。操作系统选择用户线程并将它映射到线程池中的可用内核线程。多个用户线程可以分配给相同的内核线程。

### 多对一模型

#### 特点：

多对一的模型将多个用户线程映射到一个内核线程。  
多对一模型线程的切换速度要快很多（线程之间的切换由用户代码来执行）

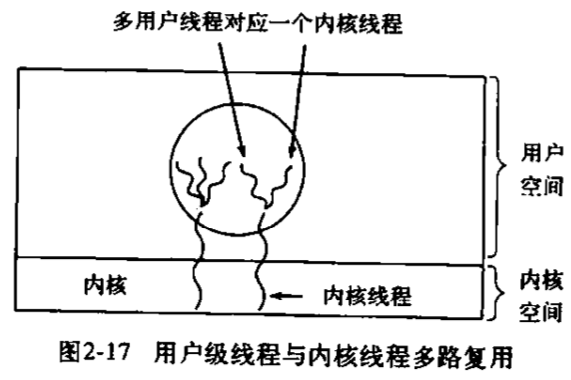
#### 弊端:

如果其中一个线程阻塞，那么所有线程将无法执行。

### 多对多模型

#### 特点：

将上述两种模型的特点进行综合，即将多个用户线程映射到少数但不只一个内核线程中去。  
多对多模型对用户线程的数量没有什么限制，在多处理器系统上也会有一定的性能提升，不过提升的幅度比不上一对一模型。



**调度程序激活：**

目标是模拟内核线程的功能，但是为线程包提供了通常通常在用户空间中才能实现的更好的灵活性和性能。避免用户空间与内核空间不必要的转换；

内核给每个进程安排一定数量的虚拟处理器，并且让（用户空间）运行时系统将线程分配到处理器上。

机制工作的基本思路：内核了解到一个线程被阻塞后，内核通知该进程的运行时系统，并且在堆栈中以参数形式传递有问题的TID和所发生的事件的描述

**上行调用：**

内核通知一个以知的起始地址启动运行时系统

**线程调度过程：**

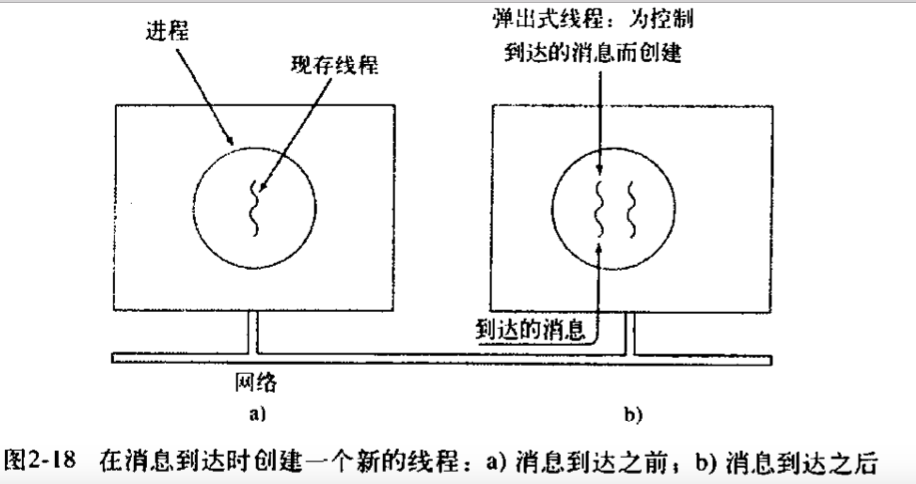
把当前线程标记为阻塞线程，并从线程就绪表中取出另外一个线程，设置寄存器，再启动

**弹出式线程：**

传统服务请求：将线程或者进程阻塞在一个receive系统调用上，等待消息的到来。当消息到达时，该系统调用接受消息，并且打开消息内容并且检查处理

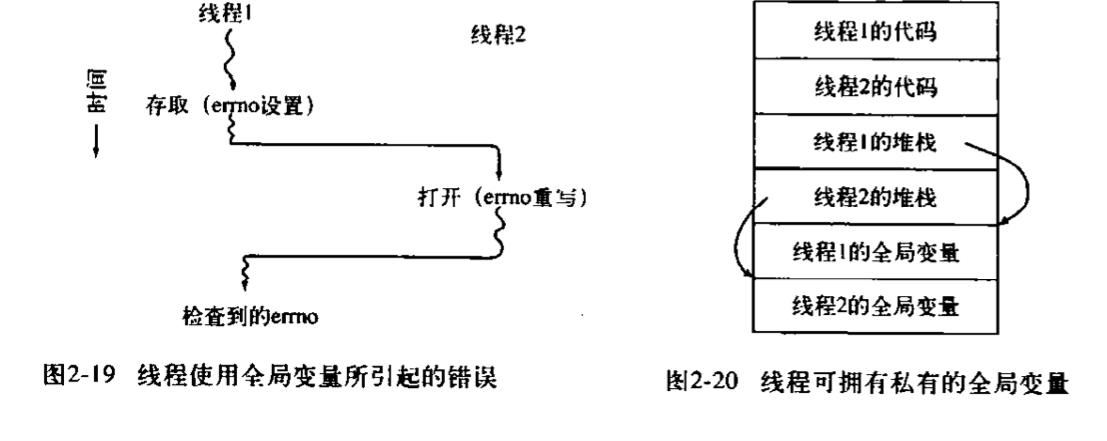
弹出式线程：一个消息的到达导致系统创建一个处理该消息的线程

优点：没有必须存储的寄存器，堆栈诸如此类的内容。每个线程从全新开始，每一个线程彼此之间完成一样，消息到达与处理开始之间的时间非常短



**使单线程代码多线程化：**

UNIX维护的errno变量：当线程或者进程进行系统调用失败时，错误码放在二errno。

解决方法：为线程赋予其私有的全局变量

解决方案（1）为全局变量申请一块内存，并将它转达给线程中的每一个过程作为额外的参数

（2）引入新的库过程，以便创建，设置和读取这些线程范围的全局变量

create\_global(“bufptr”)

set\_global(“bufptr”, &buf)

read\_global(“bufptr”)

**进程间的通信（IPC）**

竞争条件：多个进程读写某些共享数据，而最后的结果取决于进程运行的精确时序

临界区：共享内存进行访问的程序片段

解决保证使用共享数据时的并发进程能够正确和高效的协作方案

1）任何两个进程不同时在同一临界区

2）不应对CPU的速度和数量做任何处理

3）临界区外运行的进程不得阻塞其他进程

4）不得使进程无限期的等待进入临界区

互斥：阻止多个进程同时读写共享数据

**忙等待互斥：**

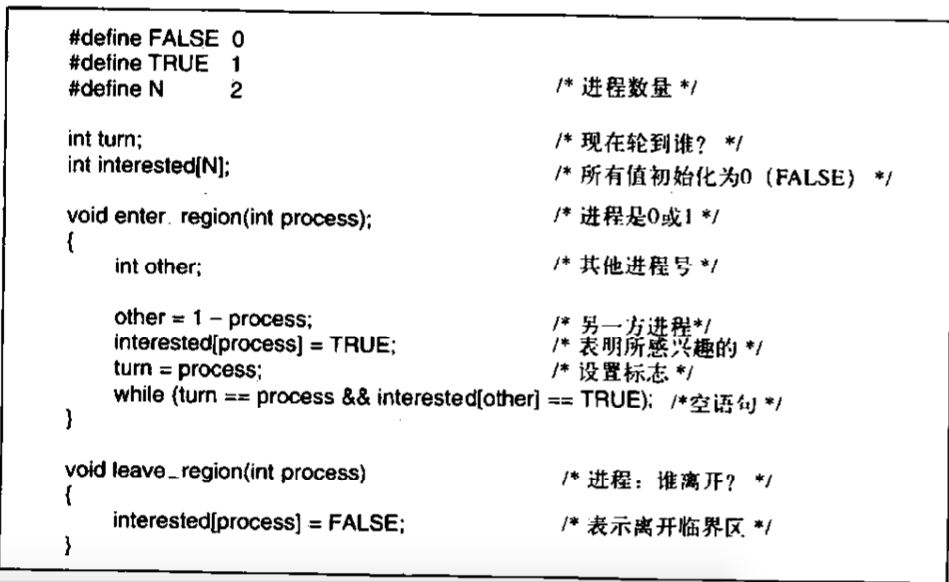
（1）屏蔽中断：每个进入临界区的进程屏蔽中断，离开之后再打开中断。CPU只有在发生时钟中断或者其他中断的时候才可以进行进程切换。屏蔽中断之后进程就可以检测和修改共享内存

总结：屏蔽中断对于操作系统本身是一项有用的技术，但对用户进程不是一直通用的互斥机制

（2）锁机制：互斥锁， 自旋锁， 读写锁

（3）严格轮换法

（4）perterson算法



**睡眠与唤醒：**

sleep是一个引起进程阻塞的系统调用，即被挂起，直到另外的进程将其唤醒

wakeup调用有一个参数，即唤醒进程。

**生成者与消费者问题（界缓冲区问题）：两个进程共享公共固定大小的buffer，其中一个是生产者，将信息放入buffer，另外一个是消费者，将信息从buffer中取出。（m个生产者， m个消费者）**

**信号量：**

使用一个整型变量来累计唤醒次数，供以后使用。这个变量就是信号量

0（表示没有存下来的唤醒操作）或者为正值（表示一个或者多个唤醒操作）

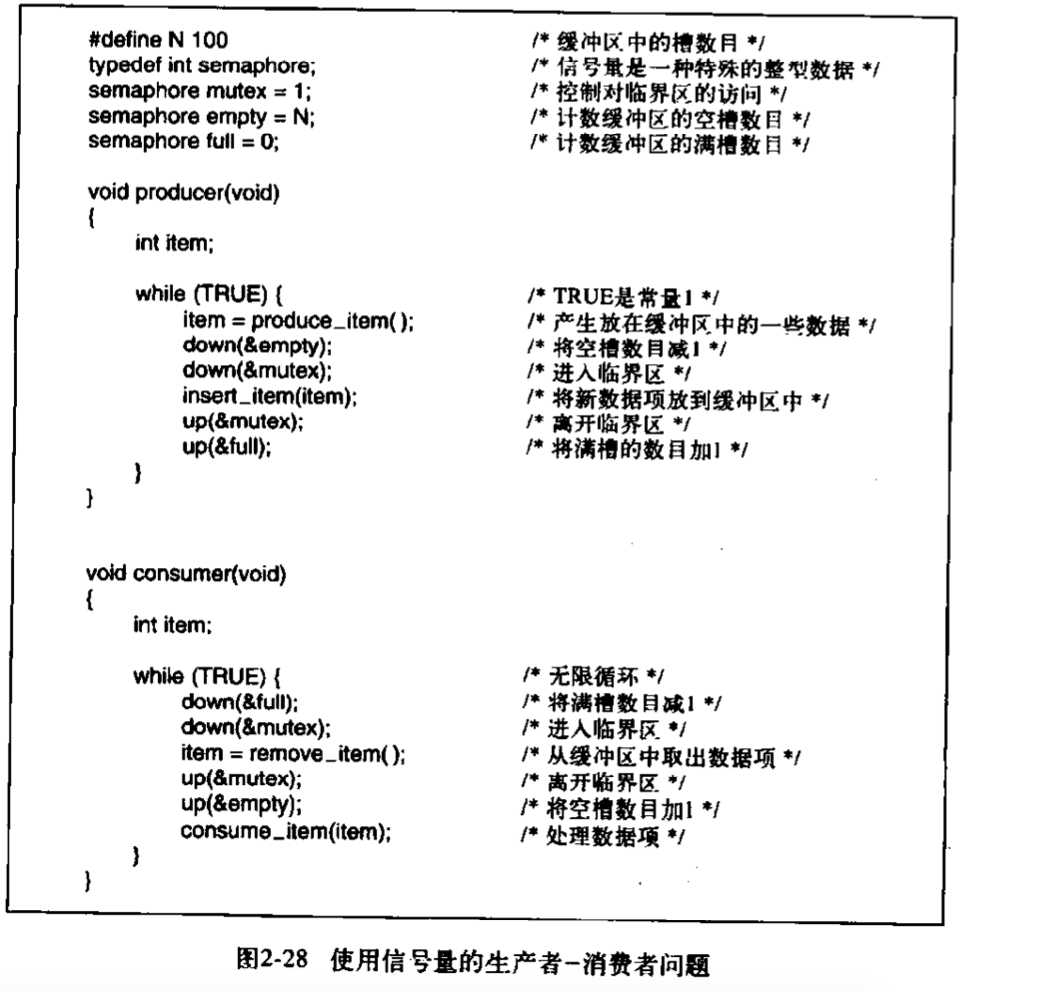
原子操作：指相关联的操作要么都不间断的执行，要么都不执行（即执行某一操作不会被打断）；主要分临界区的锁和执行原子操作的函数

**@解决生产者与消费者问题**

**full ： 记录充满的缓冲槽数月，初值为0；**

**empty ： 记录空的缓冲槽总数，处值为缓冲区的数目；**

**mutex：用来确保生产者和消费者不会同时访问缓冲区，初值为1；**



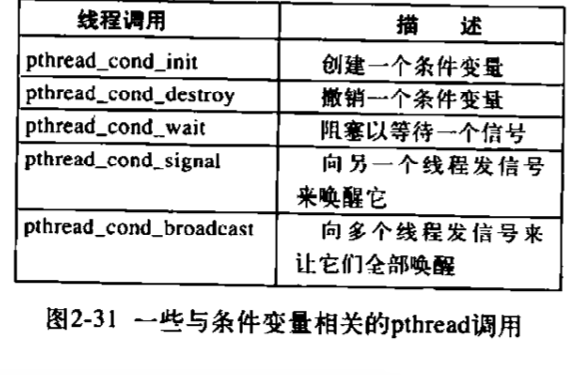
**互斥量：**

不需要信号量的计数能力，适用于管理共享资源或者一小段代码。

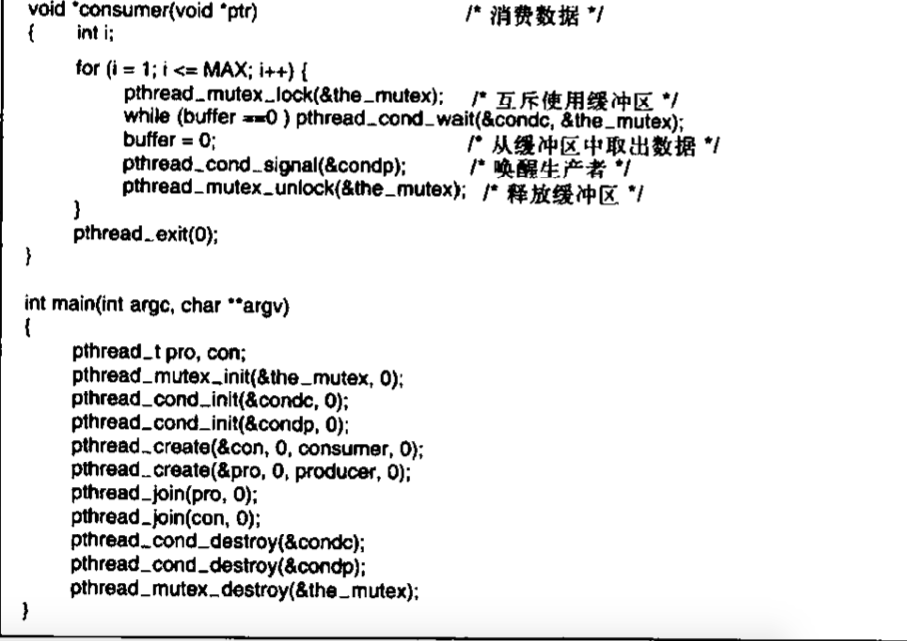
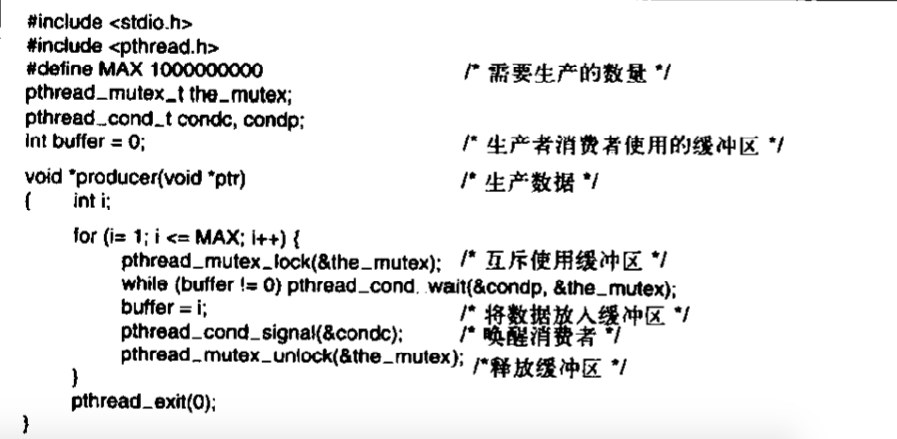
PThread中的互斥

Pthread提供了许多同步线程的函数，基本机制是使用一个可以被锁定和解锁的互斥量保护临界区。



另外一种同步机制：条件变量

**使用条件变量和互斥量一起来解决生产者和消费这问题**



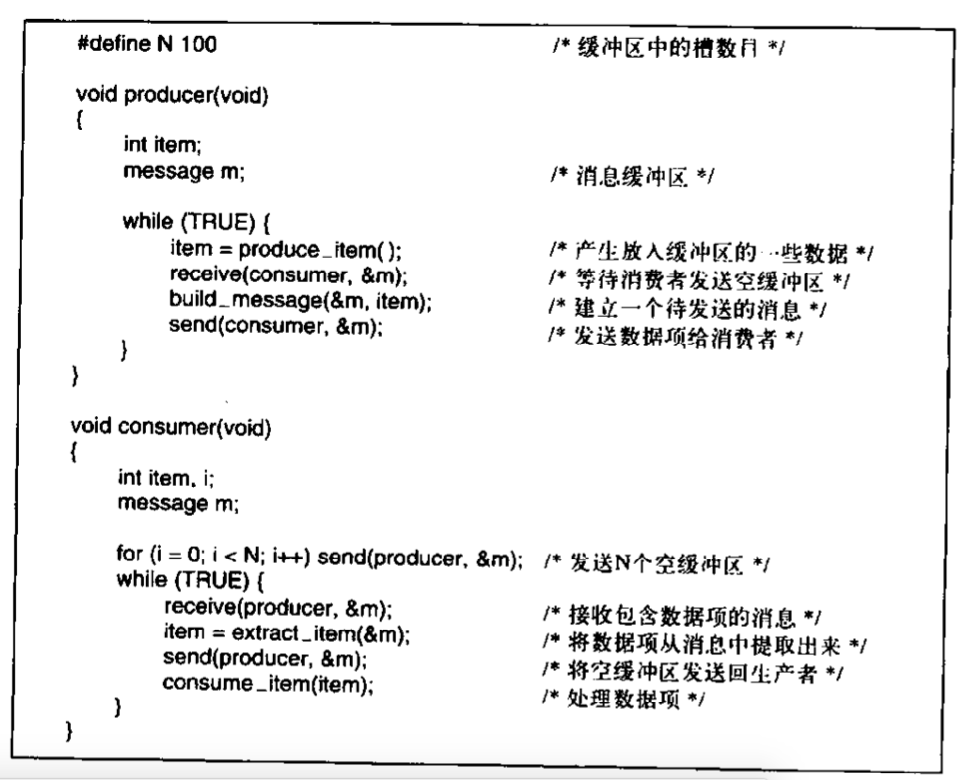
**消息传递**

send(destination, &message)/receive(source, &message);

使用消息传递解决生产者-消费者问题

保证信箱总数不变，当生产者速度大于消费者 生产者被阻塞

反之消费者被阻塞



**调度**

多个进程或者线程同时竞争CPU，因此为了完成选择工作，操作系统使用了**调度程序**，该程序使用的算法称为**调度算法；**

进程行为

I/O密集型：在等待I/O处理上花费绝大多的时间；

计算密集型：花费时间在计算上

何时调度：

1）创建一个新进程

2）进程退出是做出调度决策

3）当一个进程阻塞在I/O和信号量或者由于其他原因阻塞时，必须选择另外一个进程运行。

4）在一个I/O中断发生时，必须做出调度决策

调度算法：

抢占式多任务调度（Linux， 交互， 实时，服务器）

非抢占式多任务调度（批处理系统）

用于检查批处理操作系统的三个指标：**吞吐量， 周转时间以及CPU利用率**

对于交互式系统的指标：**最小响应时间**

实时系统主要满足所有的截止时间要求

**批处理系统中的调度**

**1）**最简单的是非抢占式的先来先服务（frist-come first-serverd）算法。进程按它们请求CPU的顺序使用CPU。单链表记录所有就绪的进程，每次就绪的进程放在链表尾部

2)最短作业优先：适用于运行时间可以预知的另一个非抢占式算法

3）最短剩余时间优先：抢占式版最短作业优先

**交互式系统中的调度**

1. 轮转调度（最古老，最简单，最公平）：每个进程分配一个时间段即时间片，运行进程在该时间段内运行。如果时间过后该进程还在运行则强制挂起该进程，分配CPU给另外一个进程（使用队列，每次轮转到队列尾）

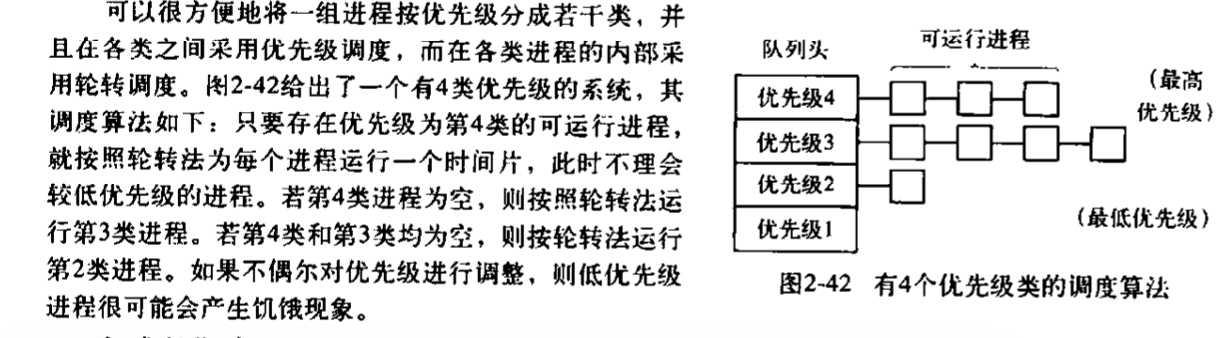
时间片

2）优先级调度：每个进程被赋予优先级，允许级别最高的进程先运行

为防止高优先级进程无止境的运行下去，调度程序可以在每个时钟中断降低当前进程的优先级。

为达到某种目的，优先级也可以由系统动态确定。

使I/O密集型进程获取较好的服务的简单算法：设其优先级为1/f， f为该进程上一时间片中所占的部分。



3）最短进程优先

4）保证调度

5）彩票调度

6）公平分享调度

**实时系统中的调度**

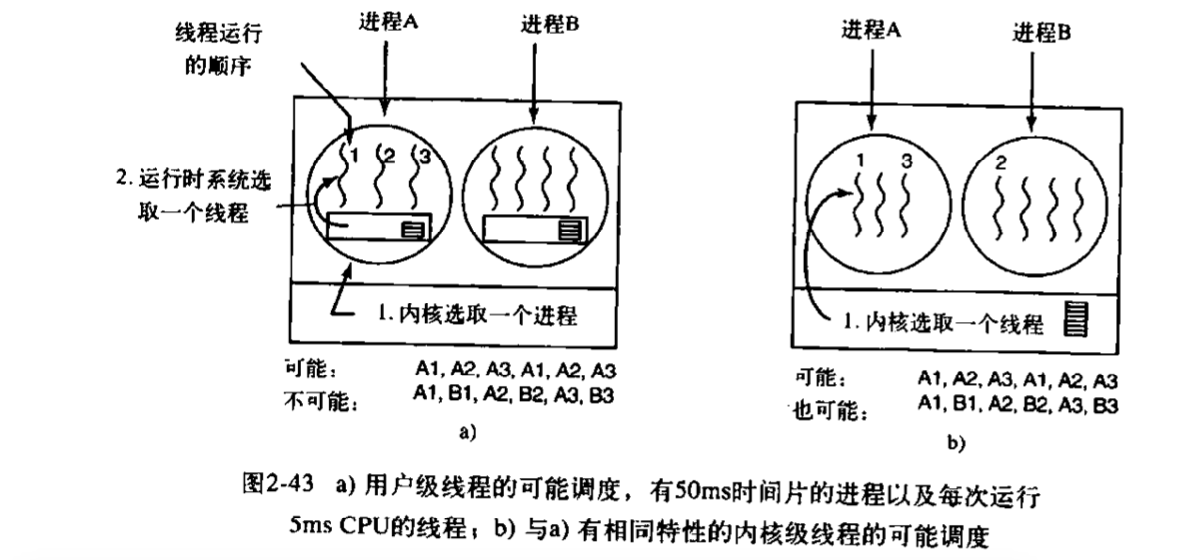
**：是一种时间为主导的系统**

软实时：虽然不希望偶尔错失截止时间，但是可以容忍

硬实时：绝对的截止时间

**调度策略与调度机制分离**

调度算法以某种形式参数化，而参数由用户进程填写。



用户级线程调度：把整个线程实现部分放在用户空间中，内核对线程一无所知，内核看到的就是一个单线程进程

内核级线程：在内核中有一个用来记录系统中所有线程的线程表（TCB，进程表PCB），当某个线程希望创建一个新线程或撤销一个已有线程时，它进行一个系统调用，这个系统调用通过对线程表的更新完成线程的创建或撤销工作。

最大的差别在与性能的使用。

内核级的线程需要的完整的上下文切换，修改内存映像，使cache失效；一旦线程阻塞不需要像用户级进程一样挂起整个进程；

用户级线程可以专门为应用程序定制的线程调度程序。

用户级线程优点：

1）整个用户级线程的切换发生在用户空间，这样的线程切换至少比陷入内核要快一个数量级（不需要陷入内核、不需要上下文切换、不需要对内存高速缓存进行刷新，这就使得线程调度非常快捷）

2）用户级线程有比较好的可扩展性，线程能够利用的表空间和堆栈空间比内核级线程多，这是因为在内核空间中内核线程需要一些固定的表格空间和堆栈空间，如果内核线程的数量非常大，就会出现问题。

3）可以在不支持线程的操作系统中实现。

4）创建和销毁线程、线程切换代价等线程管理的代价比内核线程少得多, 因为保存线程状态的过程和调用程序都只是本地过程

5）允许每个进程定制自己的调度算法，线程管理比较灵活。这就是必须自己写管理程序，与内核线程的区别

6）线程的调度不需要内核直接参与，控制简单。

缺点：

1）一个线程阻塞，会阻塞该进程中其他所有的线程（具体，举个例子）

比如：线程发生I/O或页面故障引起的阻塞时，如果调用阻塞系统调用则内核由于不知道有多线程的存在，而会阻塞整个进程从而阻塞所有线程

页面失效也会产生类似的问题。

2）如果一个线程开始运行，那么该进程中其他线程就不能运行，除非第一个线程自动放弃CPU。因为在一个单独的进程内部，没有时钟中断，所以不能用轮转调度（轮流）的方式调度线程

内核级线程优点：

1）当一个线程阻塞时，内核根据选择，可以运行同一个进程或其他进程

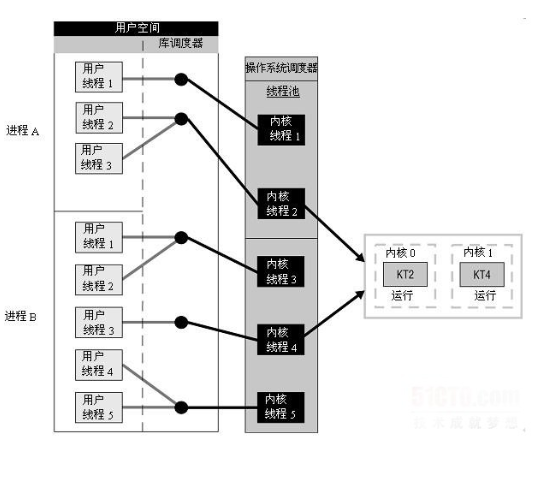
缺点：

1）在内核中创建和撤销线程的开销比较大，速度慢

3、用户级、核心级线程相结合：

在一些系统中，使用组合方式的多线程实现, 线程创建完全在用户空间中完成，线程的调度和同步也在应用程序中进行. 一个应用程序中的多个用户级线程被映射到一些（小于或等于用户级线程的数目）内核级线程上。

下图说明了用户级与内核级的组合实现方式, 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



4、用户级、核心级线程优缺点、区别：

1）内核支持线程是OS内核可感知的，而用户级线程是OS内核不可感知的。

2）用户级线程的创建、撤消和调度不需要OS内核的支持，是在语言（如Java）这一级处理的；而内核支持线程的创建、撤消和调度都需OS内核提供支持，而且与进程的创建、撤消和调度大体是相同的。

3）用户级线程执行系统调用指令时将导致其所属进程被中断，而内核支持线程执行系统调用指令时，只导致该线程被中断。

4）在只有用户级线程的系统内，CPU调度还是以进程为单位，处于运行状态的进程中的多个线程，由用户程序控制线程的轮换运行；在有内核支持线程的系统内，CPU调度则以线程为单位，由OS的线程调度程序负责线程的调度。

5） 用户级线程的程序实体是运行在用户态下的程序，而内核支持线程的程序实体则是可以运行在任何状态下的程序。

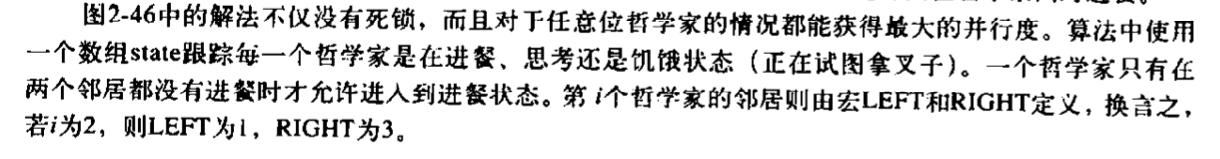
5、当一个多线程进程创建新的进程时面临的问题：

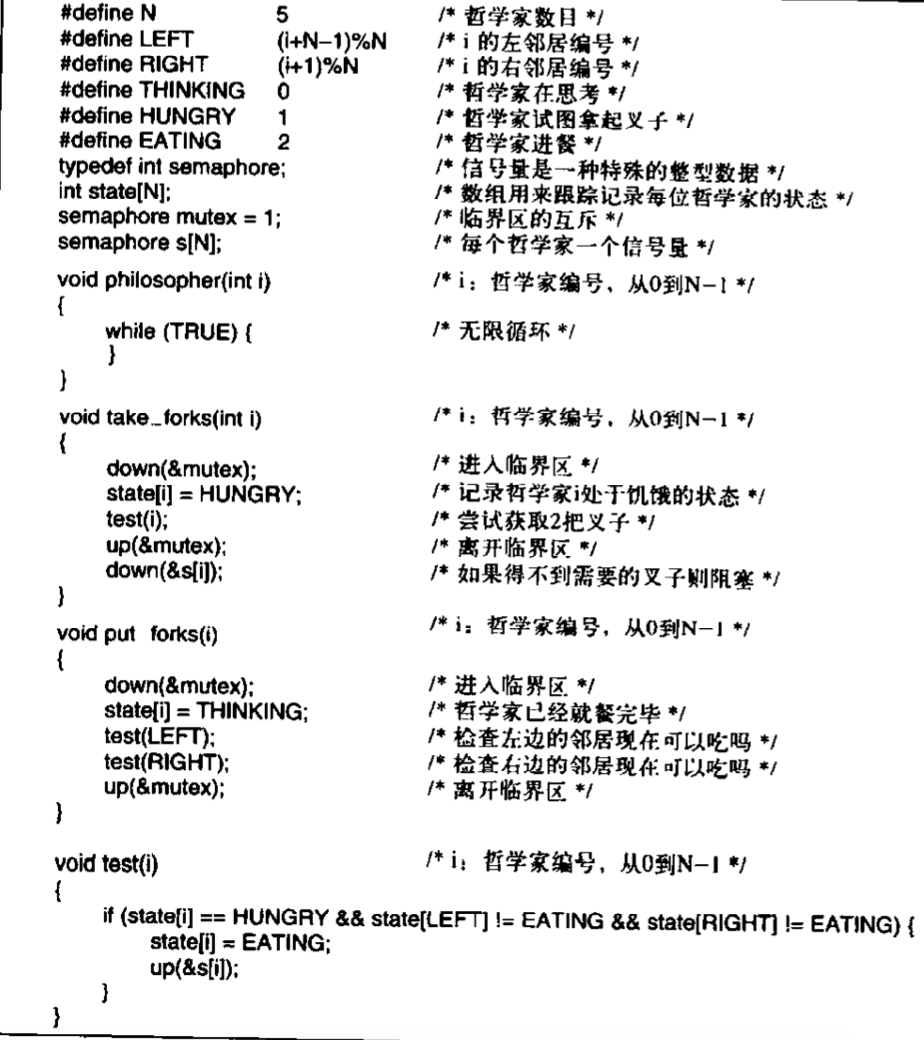
1）子进程是否继承所有的线程；

2）信号是发给进程而不是线程的，那么当一个信号到达时，由那个线程来处理，线程可以“注册”它们感兴趣的信号

**经典的IPC问题**

1）哲学家就餐问题（Dijkstra）（同步问题）：互斥访问有限资源的竞争问题

对于5个哲学家，需要两个叉子一起才可以用餐，放下叉子则思考



**读者-写者问题**

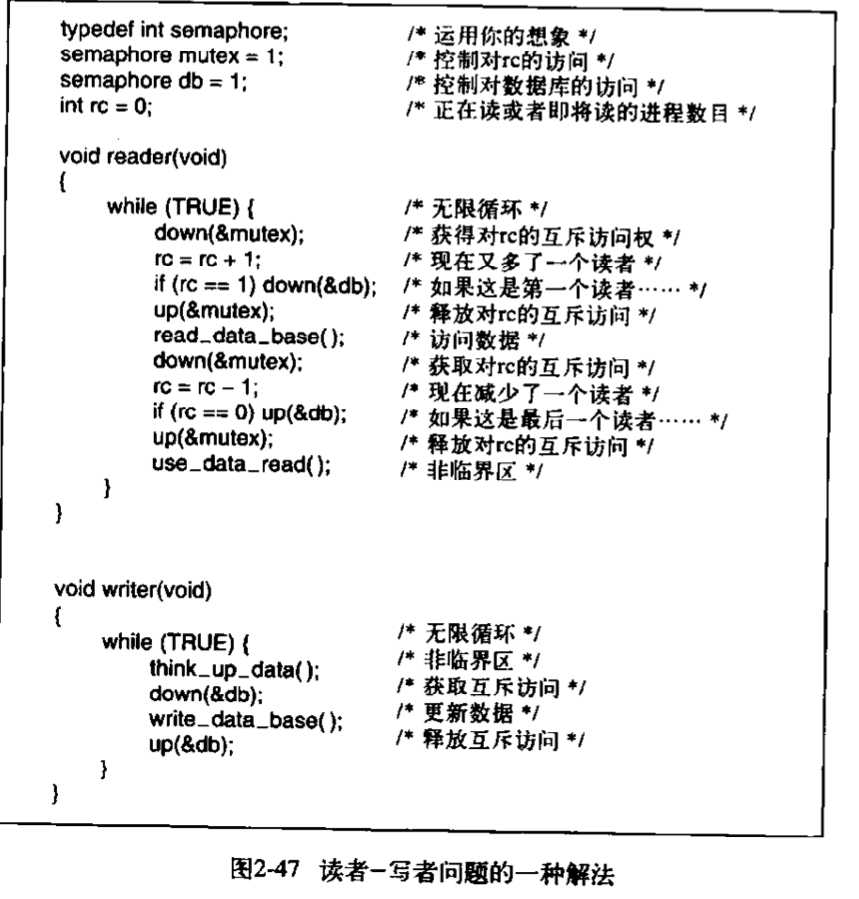
：为数据库访问建立了模型

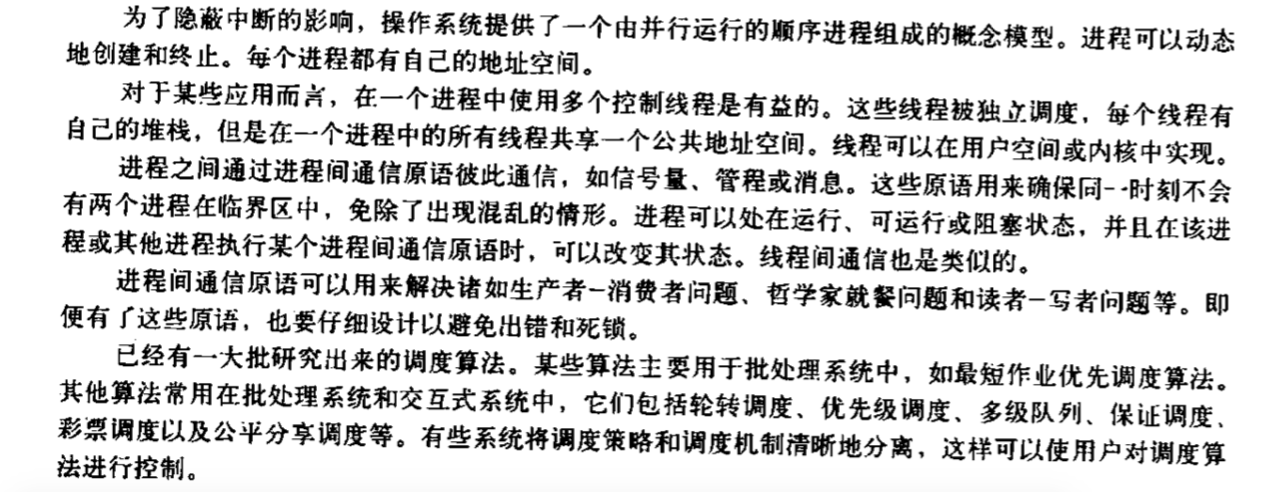
问题描述：对于多个读者访问数据库没有问题，当是一个写者到来的时候只能被挂起，等待读者完成；

**1.利用记录型信号量解决读者--写者问题**

* 互斥信号量wmutex: 实现Reader与Writer进程间在读或写时的互斥，整型变量Readcount: 表示正在读的进程数目;  
  由于只要有一个Reader进程在读，便不允许Writer进程写。所以，仅当Readcount=0，即无Reader进程在读时，Reader才需要执行Wait(wmutex)操作。若Wait(wmutex)操作成功，Reader进程便可去读，相应地，做Readcount+1操作。  
  同理，仅当Reader进程在执行了Readcount减1操作后其值为0时，才需执行signal(wmutex)操作，以便让Write进程写
* 互斥信号量rmutex: Reader进程间互斥访问Readcount

总结：  
wmutex:是读写的互斥信号量  
rmutex: 是读进程互斥访问Readcount的信号量

另外一个解法

**总结**

**课后习题及答案**

1、图2-2中给出了三个进程状态，在理论上，三个状态可以有六种转换，每个状态两个。但是，图中只给出了四种转换。有没有可能发生其他两种转换中的一个或两个？

答：从阻塞到运行的转换是可以想象的。假设某个进程在I/O上阻塞，而且I/O结束，如果此时CPU空闲，该进程就可以从阻塞态直接转到运行态。而另外一种转换（从阻塞态到就绪态）是不可能的。一个就绪进程是不可能做任何会产生阻塞的I/O或者别的什么事情。只有运行的进程才能被阻塞。

2、假设要设计一种先进的计算机体系结构，它使用硬件而不是中断来完成进程切换。CPU需要哪些信息？请描述用硬件完成进程切换的工作过程。

答：应该有一个寄存器包含当前进程表项的指针。当I/O结束时，CPU将把当前的机器状态存入到当前进程表项中。然后，将转到中断设备的中断向量，读取另一个过程表项的指针（服务例程），然后，就可以启动这个进程了。

3、在所有当代计算机中，至少有部分中断处理程序是用汇编语言编写的。为什么？

答：通常，高级语言不允许访问CPU硬件，而这种访问是必需的。例如，中断处理程序可能需要禁用和启用某个特定设备的中断服务，或者处理进程堆栈区的数据。另外，中断服务例程需要尽快地执行。

4、当中断或系统调用把控制转给操作系统时，通常将内核堆栈和被中断进程的运行堆栈分离。为什么？

答：内核使用单独的堆栈有若干的原因。其中两个原因如下：

首先，不希望操作系统崩溃，由于某些用户程序不允许足够的堆栈空间。

第二，如果内核将数据保留在用户空间，然后从系统调用返回，那么恶意的用户可能使用这些数据找出某些关于其它进程的信息。

5、多个作业能够并行运行，比它们顺序执行完成的要快。假设有两个作业同时开始执行，每个需要10分钟的CPU时间。如果顺序执行，那么最后一个作业需要多长时间可以完成？如果并行执行又需要多长时间？假设I/O等待占50%。

答：CPU利用率计算公式：CPU利用率 = 1 - pn。设运行作业所需要的时间为T。

顺序执行：T = (10 + 10) / 0.5 = 40(分钟)；

并行执行：p = 0.5，n = 2，CPU利用率 = 1 - 0.52 = 0.75，则T = (10 + 10) / 0.75 = 26.7(分钟)。

6、在本章中说明的图2-11a的模式不适合用于使用内存高速缓存的文件服务器。为什么不适合？每个进程可以有自己的高速缓存吗？

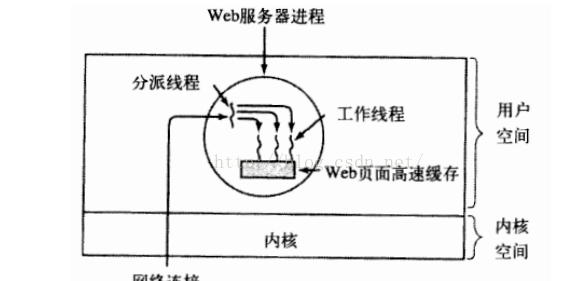
---------------------

：即使是有可能实现，也是很难保持文件系统的一致性。假设某个客户进程给服务器进程1发送请求要更新文件。该进程更新其内存的cache项。然后，另一个客户进程给服务器进程2发送请求读取该文件。不幸的是，如果该文件还在 cache中，服务器进程2对此毫不知情，将返回过时的数据。如果第一个进程在缓冲后将文件写到磁盘中， 而服务器进程2每次读取时检查磁盘其缓存的备份是否是最新的，系统还可以工作，但是需要避免磁盘访问的所有缓存系统。

7、如果创建一个多线程进程，若子进程得到全部父进程线程的副本，会出现问题。假如原有线程之一正在等待键盘输入，现在则成为两个线程在等待键盘输入，每个进程有一个。在单线程进程中也会发生这种问题吗？

答：不会。如果单线程进程在键盘上阻塞，就不能创建子进程。

8、在图2-8中，给出了一个多线程Web服务器。如果读取文件的惟一途径是正常的阻塞read系统调用，那么Web服务器应该使用用户级线程还是内核级线程？为什么？



9、在本章中，我们介绍了多线程Web服务器，说明它比单线程服务器和有限状态机服务器更好的原因。存在单线程服务器更好一些的情形吗？请给出一个例子。

答：是的。如果服务器是完全CPU绑定的，则不需要多线程。这只会增加不必要的复杂性。假设某个百万人口区域的电话查号系统(类似于114)，如果每个(姓名，电话号码)记录为64个字符，整个的数据库则为64MB，这就很容易全部读入服务器内存中以提供快速的查询。

10、在图2-12中寄存器集合按每个线程中的内容列出而不是按每个进程中的内容列出。为什么？毕竟机器只有一套寄存器。

答：当一个线程停止时，它在寄存器中有值。它们必须被保存，就像进程停止时，必须保存寄存器。多线程和多进程没有什么不同，所以每个线程需要自己的寄存器保存区。

11、为什么线程要通过调用thread\_yield自愿放弃CPU? 毕竟，由干没有周期性的时钟中断，线程可以不交回CPU。

答：进程中的线程合作。它们彼此不敌对。如果应用程序需要阻塞以运行得更好，那么一个线程可以调用thread\_yield自愿放弃CPU。毕竟，同一个进程中的线程的全部代码通常是一个程序员写的。

12、线程可以被时钟中断抢占吗？如果可以，什么情形下可以？如果不可以，为什么不可以？

答：用户级线程不能被时钟剥夺，除非整个进程的时间片用完。内核级线程可以单独地被剥夺。在后一种情况下，如果线程运行过久，时钟将中断该当前进程，因而当前线程也被中断。内核可以自由地从同一个进程中选取其他线程运行。

13、在本习题中，要求对使用单线程文件服务器和多线程文件服务器读取文件进行比较。假设所需要的数据都在块高速缓存中，花费15ms获得工作请求，分派工作，并处理其余必要工作。如果在三分之一时间时，需要一个磁盘操作，要另外花费75ms，此时该线程进入睡眠。在单线程情形下服务器每秒钟可以处理多少个请求？如果是多线程呢？

答：在单线程情况下，cache命中需15ms，cache未命中需要90ms。其加权平均为2/3\*15+1/3\*90。因此，平均请求为40ms，而服务器每秒可处理25个。对于多线程服务器，所有磁盘等待都是重叠的，因此每个请求都耗时15ms，而服务器每秒可处理66.6666个请求。

14、在用户空间实现线程，其最大的优点是什么？最大的缺点是什么？

答：最大的优势就是效率。不需要陷入内核来切换线程。最大的缺点是，如果一个线程阻塞，整个进程都会阻塞。

15、在图2-15中创建线程和线程打印消息是随机交织在一起的。有没有方法可以严格按照以下次序运行：创建线程1，线程1打印消息，线程1结束，创建线程2，线程2打印消息，线程2结束，以此类推；如果有，是什么方法，如果没有请解释原因。

答：是的，这是可以做到的。每次执行pthread\_create后，主程序可以调用pthread\_join等待刚刚创建的线程退出后再创建下一个线程。

16、在讨论线程中的全局变量时，曾使用过程create\_global将存储分配给指向变量的指针，而不是变量自身。这是必需的，还是由于该过程也需要使用这些值？

答：指针是确实必要的，因为全局变量的大小是未知的。它可能是从字符到浮点数数组的任何类型。如果保存其值，就不得不把其大小传递给create\_global，这都没有问题，但是必须将其类型作为set\_global的第二个参数，那么read\_global返回值的类型是什么呢。

17、考虑线程全部在用户空间实现的一个系统，其中运行时系统每秒钟得到一个时钟中断。假设在该运行时系统中，当某个线程正在执行时发生一个时钟中断，此时会出现什么问题？你有什么解决该问题的建议吗？

答：runtime系统可以正好在这一时刻阻塞或者解除阻塞某个线程，并且忙于处理调度队列。此时并不适合于时钟中断处理程序开始检查该队列是否应该进行线程切换，因为它们可能处于不一致的状态。解决方法可以是：当进入runtime系统后，设置一个标志。时钟处理程序将看到该标志，并且设置其自己的标志，然后返回。当runtime系统完成时，它将检测时钟标志，看是否有时钟中断发生，并且现在运行时钟处理程序。

18、假设一个操作系统中不存在类似于select的系统调用来提前了解在从文件、管道或设备中读取时是否安全，不过该操作系统确实允许设置报警时钟，以便中断阻塞的系统调用。在上述条件下，是否有可能在用户空间中实现一个线程包？请加以讨论。

答：这是可能的，不过效率很低。线程想要做一个系统调用，首先设定警报定时器，然后才执行调用。如果线程阻塞，定时器将控制归还给线程包。当然，大多数调用是不阻塞的，而定时器必须被清除。每个可能被阻塞的系统调用都必须作为3个系统调用来执行。如果定时器过早失效，各种问题都可能发生。用这种方法建立线程包并不好。

19、在2.3.4节中所讨论的优先级反转问题是否可能在用户级线程中发生？为什么？

答：当低优先级进程位于其临界区，而高优先级进程就绪并且被调度时，将发生优先级倒置问题。如果使用忙等待，高优先级进程将一直运行。对于用户级线程，不可能发生低优先级线程突然被剥夺而允许高优先级线程运行，因为是不可剥夺的。而内核级线程，就会出现这个问题。

20、在2.3.4节中，描述了一种有高优先级进程H和低优先级进程L的情况，导致了H陷入死循环。若采用轮转调度算法而不是优先级调度算法，还会发生这样问题吗？请给予讨论。

答：在轮转调度算法下。L迟早会运行，最终它将会离开临界区。关键是，在优先级调度算法下，L永远不会运行；循环，它定期得到一个正常的时间片，所以有机会离开其临界区。

21、在使用线程的系统中，若使用用户级线程，是每个线程一个堆栈还是每个进程一个堆栈？如果使用内核级线程倩况又如何呢？请给予解释。

答：每个线程都是自己调用例程，因此它必须有其自己的堆栈以保存局部变量、返回地址等等。这一点用户级线程和内核级线程是一样的。

22、在开发计算机时，通常首先用一个程序模拟，一次运行一条指令，甚至多处理器也严格按此模拟。在类似于这种没有同时事件发生的情形下，会出现竞争条件吗？

答：是的。模拟计算机也可以是多道程序设计的。例如，在进程A运行时，它读取一些共享变量。然后发生了一个模拟时钟周期和进程B运行。它也读取相同的变量，然后对变量进行了加1操作。当进程A运行时，如果它也对变量进行了加1操作，就发生了竞争条件。

23、两个进程在一个共享存储器多处理器（即两个CPU）上运行，当它们要共享一个公共内存时，图2-23所示的采用变量turn的忙等待解决方案还有效吗？

答：是，它还是有用的。当然，它依然是忙等待。

24、在进程调度是抢占式的情形下，图2-24中展示的互斥问题的Peterson解法能正常工作吗？如果是非抢占式的情况呢？

答：该方法对可剥夺调度完全没问题。事实上，它就是为这种情况设计的。当调度为不可剥夺的，该方法将会失败。假设初值为0，而进程1首先运行。它将一直循环，永不释放CPU。

25、给出一个可以屏蔽中断的操作系统如何实现信号量的框架。

答：执行信号量操作，操作系统首先要禁用中断。然后，它读取信号量的值。如果执行down操作，而信号量等于0，就将调用进程放入与信号量有关的阻塞进程列表中。如果执行up操作，必须检测看是否有任何进程在信号量上被阻塞。如果有一个或多个进程被阻塞，从阻塞进程的列表中移出一个，使之就绪。当所有这些操作都完成后，就可以开启中断了。

26、请说明计数信号量（即可以保持一个任意值的信号量）如何仅通过二元信号量和普通机器指令实现。

答：将每个计数信号量与2个二值信号量联合：M用于互斥；S用于阻塞。另外，每个计数信号量都组合一个用于保存up次数减去down次数的计数器，以及在该信号量上阻塞的进程列表。为了实现down操作，进程首先通过对M执行down操作，以获得对信号量、计数器以及列表的独占访问权。然后，将计数器减 1。如果大于等于0, 只需对M执行up操作并退出即可。如果M为负数，就将该进程放入阻塞进程列表中。接着，对M执行up操作，对S执行down操作来阻塞该进程。为了实现up操作，首先对M执行down操作以获得互斥，然后将计数器加1。如果计数器大于0, 则没有进程阻塞，就只需对M执行up操作。不过， 如果计数器小于等于0, 则必须从列表中移出某些进程。最后，按次序对B和M执行up操作。

27、如果一个系统只有两个进程，可以使用一个屏障来同步这两个进程吗？为什么？

答：如果程序操作按阶段执行，直到两个进程都完成当前阶段才能进入下一阶段，这时就应该使用屏障。

28、如果线程在内核中实现，可以使用内核信号量对同一个进程中的两个线程进行同步吗？如果线程在用户空间实现呢？假设在其他进程中没有线程必须访问该信号量。请讨论你的答案。

答：对于内核线程，线程可以在信号量上阻塞，而内核可以运行该进程中的其它线程。因而，使用信号量没有问题。而对于用户级线程，当某个线程在信号量上阻塞时，内核将认为整个进程都被阻塞，而且不再执行它。因此，进程失败。

29、管程内的同步机制使用条件变量和两个特殊操作wait和signal，一种更通用的同步形式是只用一条原语waituntil，它以任意的布尔谓词作为参数。例如

waituntilx < 0 or y + z < n

这样就不再需要signal原语。很显然这一方式比Hoare或Brinch Hansen方案更通用，但它从未被采用过。为什么？提示：请考虑其实现。

答：其实现的代价很高。 每次在某些等待变化的进程的谓词中出现的任何变量， runtime系统都必须重新计算该谓词，以判断该进程是否能够被解锁。而对于 Hoare和Brinch Hansen管程，则只需signal原语即可唤醒进程。

30、一个快餐店有四类雇员：(1) 领班，接收顿客点的菜单；（2）厨师，准备饭菜；（3）打包工，将饭菜装在袋子里；（4) 收银员，将食品袋交给顾客并收钱。每个雇员可被看作一个进行通信的顺序进程。它们采用的进程间通信方式是什么？请将这个模型与UNIX中进程联系起来。

答：雇员之间通过消息传递进行通信：在该例中，消息为订单、食物和袋子。在UNIX中，该4个进程通过管道连接。

31、假设有一个使用信箱的消息传递系统，当向满信箱发消息或从空信箱收消息时，进程都不会阻塞，相反，会得到一个错误代码。进程响应错误代码的处理方式为一遍一遍地重试，直到成功为止。这种方式会导致竞争条件吗？

答：它不会导致竞争条件（不会丢失任何东西），不过它是完全的忙等待。

32、CDC 6600计算机使用一种称作处理器共享的有趣的轮转调度算法，它可以同时处理多达10个I/O进程。每条指令结束后都进行进程切换，这样进程1执行指令1，进程2执行指令2，以此类推。进程切换由特殊硬件完成，所以没有开销。如果在没有竞争的条件下一个进程需要T秒钟完成，那么当有n个进程共享处理器时完成一个进程需要多长时间？

答：它需要nT sec。

33、是否可以通过分析源代码来确定进程是CPU密集型的还是I/O密集型的？如何能在运行时刻进行此项决定？

答：在简单的情况下是有可能通过看源代码来判断是否为 I/O 绑定的。例如，程序开始时，将其所有输入文件读入到缓冲器中，这种程序通常不是 I/O 绑定的；但是，对不同文件进行增量地读写（诸如编译程序）的问题很有可能是I/O绑定的。如果操作系统提供诸如 UNIX ps的命令，就可以得知被程序使用的CPU 时间的量，你能把这个时间量与整个的时间比较以判断。当然，最有意义就是你是系统中唯一的用户。

34、在“何时调度”一节中曾提到，有时一个重要进程可以在选择下一个被阻塞进程进入运行的过程中发挥作用，从而改善调度性能。请给出可以这样做的情形并解释如何做。

答：对于管道中的多个进程，普通的父进程可以将数据流的信息传递给操作系统。有了这个信息，OS就可以确定哪个进程可以向需要输入的阻塞进程提供输出。

35、对某系统进行监测后表明，当阻塞在I/O之前时，平均每个进程运行时间为T。一次进程切换需要的时间为S，这里S实际上就是开销。对于采用时间片长度为Q的轮转调度，请给出以下各种情况中CPU利用率的计算公式：

a) Q = ∞

b) Q > T

c) S < Q <T

d) Q = S

e) Q趋近于0

答：CPU的效率就是有用的CPU时间除以整个的CPU时间。当Q > T时，基本的周期就是进程运行T，然后进程切换S。因此，（a）和（b）的效率都是T/(T+S)。当时间片比T短时，每运行一次T就要求T/Q次进程切换，浪费时间为ST/Q。因此，其效率为

T/(T+ST/Q)

也就是下降到 Q/(Q+S)，这就是（c）的答案。至于（d），只需以S替代Q，就可以计算出其效率为50%。最后，（e)的效率趋近于0。

36、有5个待运行作业，估计它们的运行时N分別是9, 6, 3, 5和X。采用哪种次序运行这些作业将得到最短的平均响应时间？（答案将依赖于X。）

答：最短作业优先可以使得平均响应时间最短。

0 < X ≤ 3: X, 3, 5, 6, 9.

3 < X ≤ 5: 3, X, 5, 6, 9.

5 < X ≤ 6: 3, 5, X, 6, 9.

6 < X ≤ 9: 3, 5, 6, X, 9.

X > 9: 3, 5,6, 9, X.

37、有5个批处理作业A到E，它们几乎同时到达一个计算中心。估计它们的运行时间分別为10，6，2，4和8分钟。其优先级（由外部设定）分别为3，5，2，1和4，其中5为最高优先级。对于下列每种调度算法，计算其平均进程周转时间，可忽略进程切换的开销。

a) 轮转法。

b) 优先级调度。

c) 先来先服务（按照10，6，2，4，8次序运行)。

d) 最短作业优先。

对a)，假设系统具有多道程序处理能力，每个作业均公平共享CPU时间，对b)到d) ，假设任一时刻只有一个作业运行，直到结束。所有的作业都完全是CPU密集型作业。

答：对于时间片轮转，在头10分钟里，每个作业获得1/5的CPU时间。在第10 分钟时，C结束。在接下来的8分钟里，每个作业获得 1/4 的CPU时间，然后D完成，然后，在接下来的6分钟内，余下的3个作业各获得1/3的CPU时间，直到B结束，以此类推。因此，5个作业的完成时间分别为是10, 18, 24, 28和30, 平均为22分钟。对于优先级调度，5最先运行，6分钟完成。其它作业分别在第14, 24, 26和30分钟完成，平均为18.8分钟。如果作业按A->E的次序执行，则分别在第10,16, 18, 22和30分钟完成，因此，平均为19.2分钟。最后，最短作业优先调度的完成时间分别为第2, 6, 12, 20和30分钟，平均为14分钟。

38、运行在CTSS上的一个进程需要30个时间片完成。该进程必须被调入多少次，包括第一次(在该进程运行之前）？

答：第一次得到 1 个时间片。随后获得 2, 4, 8 和 15 个时间片，因此必须经过 5 次交换。

39、能找到一个使CTSS优先级系统不受随机回车链愚弄的方法吗？

答：可以检查程序是否期待输入，并且对输入进行处理。不期待输入也不对输入进行处理的程序将不得到任何特殊的优先级提升。

40、a = 1/2的老化算法用来预测运行时间。先前的四次运行，从最老的一个到最近的一个，其运行时间分别是40ms，20ms，40ms和15ms。下一次的预测时间是多少？

答：预测值的顺序为40，30，35，所以下一次是25。（把前两个数据加一起，除以2，然后把结果和下一个数据加一起，除以2。）

41、一个软实时系统有4个周期时间，其周期分别为50ms，100ms，200ms和250ms。假设这4个事件分别需要35ms，20ms，10ms和x ms的CPU时间。保持系统可调度的最大x值是多少？

答：所使用的 CPU 的片断为 35/50 + 20/100 + 10/200 + x/250。为了使得进程可调度，必须是总和小于因此，x必须小于12.5ms。

42、请解释为什么两级调度比较常用。

答：当内存太小不能载入所有就绪进程时，就需要使用两级调度。某些进程被载入内存，并且从中选择一个运行。内存中进程会随着时间调整。这种算法容易实现也非常有效，另外，时间片轮转调度并不管进程是否在内存中。

43、一个实时系统需要处理两个语音通信，每个运行5ms，然后每次突发消耗1ms CPU时间，加上25帧/秒的一个视频，每一帧需要20ms的CPU时间。这个系统是可调度的吗？

答：每个语音通信运行200次/秒，每次突发消耗1毫秒，所以每个语音通信需要200毫秒/秒或两个语音通信需要400毫秒/秒。视频运行25次/秒，每次使用了20毫秒的时间，总共为500毫秒/秒。它们一起消耗900毫秒/秒，所以还有时间剩余，系统是可调度的。

44、考虑一个系统，在这个系统中为了内核线程调度希望将策略和机制分离。请提出一个实现此目标的手段。

答：内核可以通过任何方式调度进程，但对于每个进程，严格按照进程的优先级顺序执行。通过让用户进程设置自己的优先级，让用户控制策略，而内核处理机制。

45、在哲学家就餐问题的解法 （图2-46）中，为什么在过程 take\_forks中将状态变量置为HUNGRY?

答：如果某个哲学家阻塞，其邻居稍后能够在test中检测其状态，发现他已经饥饿，当叉子可用时，就可以唤醒他了。

46、考虑图2-46中的过程pm\_forks，假设变量state[i]在对test的两次调用之后而不是之前被置为THINKING，这个改动会对解法有什么影响？

答：该变化将意味着在哲学家停止进餐后，他的邻居都不能接着被选择。事实上，他们永远不会被选择。假设哲学家2完成了进餐，他将为哲学家1和3运行test，而两者都不会被启动，即使他们两个都饿了而且两个叉子都是可用的。类似的，如果哲学家4完成进餐，哲学家3也不会被启动。他将无法启动。

47、按照哪一类进程何时开始，读者-写者问题可以有若干种方式求解。请详细描述该问题的三种变体，每一种变体偏好（或不偏好）某一类进程。对每种变体，请指出当一个读者或写者访问数据库时会发生什么，以及当一个进程结束对数据库的访问后又会发生什么？

答：变种1：读者优先。当读者活跃时，写者都无法启动。当一个新的读者出现时，它可以立即开始除非当前有写者是活跃的。当写者完成时，如果有读者在等待，他们全都启动，无论是否有写者存在。

变种2：写者优先。当有写者等待时，读者都不会开始。当最后活跃的进程结束，如果有作者，就启动它；否则，所有读者（如果有）全部开始。

变种3：平衡的版本。当有读者是活跃的，新的读者可以立即开始。当写者完成时，新的写者优先，如果有写者等待的话。也就是说，一旦开始读，就一直读到没有读者为止。同样地，一旦开始作，所有挂起的写者都被允许运行。

48、请编写一个shell脚本，通过读取文件的最后一个数字，对之加1，然后再将该数字附在该文件上，从而生成顺序数文件。在后台和前台分別运行该脚本的一个实例，每个实例访问相同的文件。需要多长时间才出现竞争条件？临界区是什么？请修改该脚本以避免竞争（提示：使用In file file.lock锁住数据文件。）

答：一个可行的shell脚本应该如下：

if [ ! –f numbers ]; then echo 0 > numbers; fi

count=0

while (test $count != 200 )

do

count=‘expr $count + 1’

n=‘tail –1 numbers’

expr $n + 1 >>numbers

done

       同时运行该脚本两次，通过后台启动它一次（使用）和再次在前台启动。然后检查文件编号。它可能开始看起来会像数字的有序列表，但在某一时刻，由于运行脚本的两个副本而创建的竞争条件，它就失去了它的规律。通过对脚本的每一个副本测试，并在进入关键区域之前设置一个锁，在离开关键区域时打开它，可以避免竞争。可以这样做：

if ln numbers numbers.lock

then

n=‘tail –1 numbers’

expr $n + 1 >>numbers

rm numbers.lock

fi

       此版本只是当文件无法访问时直接跳过，不同的解决方案可以让进程休眠，忙等待，或仅仅循环计数，使得操作成功。

49、假设有一个提供信号量的操作系统。请实现一个消息系统，编写发送和接收消息的过程。

答：略。

50、使用管程而不是信号量来解决哲学家就餐题。

答：略。

51、假设一个大学为了卖弄其政治上的正确性，准备把美国最高法院的信条“平等但隔离其本身就是不平等（Separate but equal is inherently unequal）”既运用在种族上也运在性别上，从而结束校园内长期使用的浴室按性別隔离的做法。但是，为了迁就传统习惯，学校颁布法令：当有一个女生在浴室里，那么其他女生可以进入，但是男生不行，反之亦然。在每个浴室的门上有一个滑动指示符号，表示当前处于以下三种可能状态之一：

•空。

•有女生。

•有男生。

用你偏好的程序设计语言编写下面的过程：

woman\_wants\_to\_enter，man\_wants\_to\_enter，woman\_leaves，man\_leaves。可以随意采用所希望的计数器和同步技术。

答：略。

52、重写图2-23中的程序，以便能够处理两个以上的进程。

答：略。

53、编写一个使用线程并共享一个公共缓冲区的生产者-消费者问题。但是，不要使用信号量或任何其他用来保护共享数据结构的同步原语。直接让每个线程在需要访问时就访问。使用sleep和wakeup来处理满和空的条件。观察需要多长时间会出现严重的竞争条件。例如，可以让生产者一会儿打印一个数字，每分钟打印不要超过一个数字，因为I/O会影响竞争条件。