# 第3章 进程管理

## 3.1进程的概念

### 3.1.0程序的顺序执行

程序：一个在时间上按严格次序前后相继的操作序列，是一个静态的概念。

顺序执行：具有独立功能的程序独占处理机直至最终结束的过程。

**处理机与CPU的区别：**

处理机

处理机是计算机系统中存储程序和数据，并按照程序规定的步骤执行指令的部件。程序是描述处理机完成某项任务的指令序列。指令则是处理机能直接解释、执行的信息单位。**处理机包括中央处理器（cpu），主存储器,输入-输出接口。**处理机加接外围设备就构成完整的计算机系统

cpu

中央处理器（CPU，Central Processing Unit）是一块超大规模的集成电路，是一台计算机的运算核心（Core）和控制核心（ Control Unit）。它的功能主要是解释计算机指令以及处理计算机软件中的数据。

顺序执行特点：

(1) 顺序性

程序执行过程可看作一系列严格按程序规定的状态转移过程。

(2) 封闭性

程序执行得到的最终结果由给定的初始条件决定，不受外界因素的影响。

(3) 可再现性

只要输入的初始条件相同，则无论何时重复执行该程序都会得到相同的结果。

### 3.1.1 程序的并发执行

• 程序执行环境的重要特点：

(1) 程序的并发执行（独立性）

每道程序都是逻辑上独立的，它们之间不存在逻辑上的制约关系。

(2) 系统的用户随机地使用（随机性）

在多道程序环境下，特别是在多用户环境下，程序和数据的输入与执行开始时间都是随机的。

(3) 系统所拥有的资源被共享（资源共享）

资源共享将导致对进程执行速度的制约。

• 程序的并发执行

(1) 一组在逻辑上互相独立的程序或程序段在执行过程中，其执行时间在客观上互相重叠，即一个程序段的执行尚未结束，另一个程序段的执行已经开始的这种执行方式。

并发性与并行性区别：

并发性：两个或多个事件在同一时间间隔内发生

并行性：两个或多个事件在同一时刻发生（一定发生在多核处理器上）

语句S1和S2**并发执行条件**：对于语句S1和S2，有

① R(S1)∩ W(S2)={∮} R：读

② W(S1)∩ R(S2)={∮} W：写

③ W(S1)∩ W(S2)={∮} 同时成立

并发执行示例:

(2) 程序的并发执行所带来的影响

并发执行充分地利用了系统资源，从而提高了系统的处理能力。

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。如果并发执行的程序段不按照特定的规则和方法进行资源共享和竞争，则其执行结果将不可避免地失去封闭性和可再现性。

举例如下：

例：设有堆栈S，栈指针top，栈中存放内存中相应数据块地址（如下图3.1(a)）设有两个程序段

procedure getaddr(top) 读数据

begin

local r

r ←(top)

top ← top-1

return(r)

end

procedure reladdr(blk) 写数据

begin

top ← top+1

(top)← blk

end

图3.1 堆栈的取数和存数过程

上例中的程序段并发执行可能出现错误是由于两程序段共享资源堆栈S。一般情况下，并发执行的各程序段如果共享软、硬件资源，都会造成其执行结果受执行速度影响的局面。

为了使得在并发执行时不出现错误结果，必须采取某些措施来制约、控制各并发程序段的执行速度，控制和协调各程序段执行过程中的软、硬件资源的共享和竞争。

于是进程概念应运而生。

3.1.2 进程的定义

并发执行的程序在执行过程中分配和管理资源的基本单位。简单的说，程序的一次运行。

• 进程和程序区别和联系：

(1) 进程是一个动态概念，而程序则是一个静态概念。

(2) 进程具有并发特征，而程序没有。

(3) 进程是竞争计算机系统资源的基本单位，其并行性受到系统自己的制约。

(4) 不同的进程可以包含同一程序，只要该程序所对应的数据集不同。如何理解？

• 作业与进程的区别和关系：

(1) 作业是用户向计算机提交任务的任务实体。而进程则是完成用户任务的执行实体，是资源分配的基本单位。

(2) 一个作业可由一个或多个进程组成。

(3) 作业的概念主要用在批处理系统中，而进程的概念则用在几乎所有的多道系统中。

3.2 进程的描述

进程的静态描述由三部分组成：

进程控制块PCB：全部或部分常驻内存

程序段：

数据结构集：外存中，执行时调入内存

3.2.1 进程控制块PCB

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。当一个进程完成其功能之后，系统则释放PCB，进程也随之消亡。

不同阶段，进程的 PCB所包含的内容有所不同。但下面内容是必需的：

(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

分为初始态、就绪态、执行态和等待状态、终止状态。

② 进程优先级

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

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。

(4) CPU 现场保护结构

当前进程因等待某个事件而进入等待状态或因某种事件发生被中止在处理机上的执行时，需要保护当前进程的 CPU现场（或称进程上下文）。PCB 中设有专门的 CPU现场保护结构，以存储退出执行时的进程现场数据。

由于PCB 中包含有较多的信息，因此，一个PCB表往往要占据较大的存储空间（一般占几百到几千个字节）。在有的系统中，为了减少 PCB对内存的占用量，只允许PCB中最常用的部分，如CPU现场保护、进程描述信息、控制信息等常驻内存。PCB 结构中的其他部分则存放于外存之中，待该进程将要执行时与其他数据一起装入内存。

3.2.2 进程上下文

• 概念

上文：已执行过的进程指令和数据在相关寄存器与堆

栈中的内容；

正文：正在执行的指令和数据在相关寄存器与堆栈中的内容；

下文：待执行指令和数据在相关寄存器与堆栈中的内容；

进程上下文包括计算机系统中与执行该进程有关的各种寄存器的值、程序段在经过编译之后形成的机器指令代码集（或称正文段）、数据集及各种堆栈值和PCB结构(图3.2)。

图3.2 进程上下文结构

图3.3 UNIX System Ⅴ进程上下文组成

切换  
 进程上下文切换发生在不同的进程之间。

3.2.3 进程空间

任一进程，都有一个自己的地址空间，把该空间称为进程空间或虚空间。

程序的执行都在进程空间内进行。

在UNIX以及Linux等操作系统中，进程空间还被划分为用户空间和系统空间两大部分。

图3.5 进程空间

用户程序在用户空间内执行，而操作系统内核程序则在进程的系统空间内执行。

为了防止用户程序访问系统空间，造成访问出错，计算机系统还通过程序状态寄存器等设置不同的执行模式，即用户模式和系统模式来进行保护。人们也把用户执行模式和系统执行模式分别称为用户态和系统态。

3.3 进程状态及其转换

3.3.1 进程状态

至少具有五种基本状态：初始状态、就绪状态、执行状态、等待状态和终止状态。

• 就绪状态

已经得到除 CPU之外的其他资源，只要由调度得到处理机，便可立即投入执行。

在有些系统中，就绪状态又可进一步分为内存就绪状态和外存就绪状态。只有处于内存就绪状态的进程在得到处理机后才能立即投入执行。而处于外存就绪状态的进程只有先成为内存就绪状态后，才可能被调度执行。

执行状态

任一时刻处于执行状态的进程只能有一个。只有处于就绪状态的进程经调度选中之后才可进入执行状态。

等待状态

进程因等待某个事件发生而放弃处理机进入等待状态。

等待状态可根据等待事件的种类而进一步划分为不同的子状态，例如内存等待、设备等待、文件等待和数据等待等。这样做的好处是系统控制简单，发现和唤醒相应的进程较为容易。但系统中设置过多的状态又会造成系统参数和状态转换过程的增加。

3.3.2 进程状态转换

进程的状态转换是一个非常复杂的过程。从一个状态到另一个状态的转换除了要使用不同的控制过程（将在下节中讲述），有时还要借助于硬件触发器才能完成。例如，在 UNIX 系统中，从系统态到用户态的转换要借助硬件触发器完成。

图3.6 进程状态转换

3.4 进程控制

进程控制定义

就是系统使用一些具有特定功能的程序段来创建、撤消进程以及完成进程各状态间的转换，从而达到多进程高效率并发执行和协调、实现资源共享的目的。

原语

系统态下执行的某些具有特定功能的程序段，被高层软件所调用。

为两类：

一类是机器指令级的，其特点是执行期间不允许中断；

另一类是功能级的，其特点是作为原语的程序段不允许并发执行。

用于进程控制的原语：

► 创建原语

► 撤消原语

► 阻塞原语

► 唤醒原语

3.4.1 进程创建与撤消

进程创建

创建方式：

(1) 由系统程序模块统一创建，例如在批处理系统 中，由操作系统的作业调度程序为用户作业创建相应的进程以完成用户作业所要求的功能。

(2) 由父进程创建。

创建过程：

调用创建原语来实现。实现过程如图3.7。

图3.7 创建原语流图

2. 进程撤消

以下几种情况导致进程被撤消：

(1) 该进程已完成所要求的功能而正常终止。

(2) 由于某种错误导致非正常终止。

(3) 祖先进程要求撤消某个子进程。

如果被撤消的进程有自己的子进程，则撤消原语先撤消其子进程的 PCB结构并释放子进程所占用的资源之后，再撤消当前进程的 PCB结构和释放其资源。如图3.8。

图3.8 撤消原语流图

3.4.2 进程的阻塞与唤醒

阻塞原语在一个进程期待某一事件发生，但发生条件尚不具备时，被该进程自己调用来阻塞自己。由于该进程正处于执行状态，故应先中断处理机和保存该进程的CPU现场。然后将被阻塞进程置“阻塞”状态后插入等待队列中，再转进程调度程序选择新的就绪进程投入运行。阻塞原语的实现过程如图3.9。

图3.9 阻塞原语图

当等待队列中的进程所等待的事件发生时，等待该事件的所有进程都将被唤醒。

唤醒一个进程有两种方法：

一种是由系统进程唤醒；

另一种是由事件发生进程唤醒；

实现过程如图3.10。

图3.10 唤醒原语

3.5 进程互斥

3.5.1 资源共享所引起的制约

进程的并发执行会导致对有限性、共享资源的竞争，同时对进程的执行过程进行制约。

临界区概念

看下面例子理解临界区：

设有两个计算进程ＰA，ＰB共享内存ＭS。其中 ＭS分为三个领域，即系统区、进程工作区和数据区。这里数据区被划分大小相等的块，每个块中既可能放有数据，也有可能未放有数据。系统区主要是堆栈Ｓ，其中存放那些空数据块的地址（如图3.11）。

图3.11 多进程共享内存栈区示例

操作getspace: begin local g

g←stack ［ top ］

top←top-1

end

操作release(ad):begin

top ← top+1

stack ［ top ］←ad

end

设时刻t0时，getspace和 release(ad)按以下顺序执行：

首先 release(ad)的第一句执行，

t0：top ← top+1 → top=h0+1；

接着getspace 执行，得：

t1：g ← stack ［top］ → g=stack ［ h0+1］；

t2：top ← top-1 → top=h0；

再是 release(ad)的第二句执行，得：

t3：stack ［top］← ad → stack ［ h0 ］ ← ad；

其结果是调用getspace的进程取到的是h0+1中的一个未定义值。

把不允许多个并发进程交叉执行的一段程序称为临界部分（critical section ）或临界区（critical region）。

临界区是由属于不同并发进程的程序段共享公用数据或公用数据变量而引起的，例如上例中就是因为过程 getspace 和 release(ad)共同访问栈Ｓ中的数据而引起的。临界区不可能用增加硬件的方法来解决。因此，临界区也可以被称为访问公用数据的那段程序。

2. 间接制约

可以把那些不允许交叉执行的临界区按不同的公用数据划分为不同的集合。上例中，以公用数据栈Ｓ划分的临界区集合是{getspace,release}。把这些集合称为类（class）。

可用下列标准形式来描述临界区：

when〈类名〉do〈临界区〉od

把这种由于共享某一公有资源而引起的在临界区内不允许并发进程交叉执行的现象，称为由共享公有资源而造成的对并发进程执行速度的间接制约。

“间接”二字主要是指各并发进程的速度受公有资源制约，而不是进程间直接制约的意思。

显然，对于每一类，系统应有相应的分配和释放相应公有资源的管理办法，以制约并发进程。这就是互斥。

3. 什么是互斥

互斥定义：不允许两个以上的共享该资源的并发进程同时进入临界区称为互斥。

互斥执行时必须满足如下准则：

(1) 各并发进程享有平等的、独立的竞争共有资源的权利。（机会均等）

(2) 并发进程中的某个进程不在临界区时，它不阻止其他进程进入临界区。（有空则进）

(3) 并发进程中的若干个进程申请进入临界区时，只能允许一个进程进入。（两者择一）

(4) 并发进程中的某个进程申请进入临界区时开始，应在有限时间内得以进入临界区。(有限等待)

间接制约：竞争公有资源而引起的进程之间互斥；

直接制约：互相共享对方的私有资源所，要求各并

发进程同步执行。

3.5.2 互斥的加锁实现

人们可能认为只需把临界区中的各个过程按不同的时间排列调用就行了。但事实上这是不可能的。因为这要求该组并发进程中的每个进程事先知道其他并发进程与系统的动作，这是不可能的。

解决方法1：对临界区加锁以实现互斥

当某个进程进入临界区之后，它将锁上临界区，直到它退出临界区时为止。并发进程在申请进入临界区时，首先测试该临界区是否是上锁的。如果该临界区已被锁住，则该进程要等到该临界区开锁之后才有可能获得临界区。

加锁后的临界区程序描述如下：

key[S]表示该锁定位属于类名为Ｓ的临界区

lock(key ［Ｓ］)

〈临 界 区〉

unlock(key ［Ｓ］)

设key[S]=1时表示类名为Ｓ的临界区可用，

key[S]=0时表示类名为Ｓ的临界区不可用。

解决方法2：简便

lock (x)=begin local v

repeat

v←x

untilv=1

x←0

end

这种实现方法是不能保证并发进程互斥执行所要求的准则(3)的。因为当同时有几个进程调用lock(key[S])时，在x←0语句执行之前，可能已有两个以上的多个进程由于key[S]=1而进入临界区。为解决这个问题有些机器在硬件中设置了“测试与设置指令，保证第一步和第二步执行不可分离。

3.5.3 信号量和ＰＶ原语

1. 信号量（semaphore）

加锁方法缺点：

(1) 影响系统可靠性和执行效率的问题。循环

测试锁定位将损耗较多的 CPU计算时间。

(2) 在某些情况下出现不公平现象。例如：进程

PA和PB反复使用临界区的情况：

PA

A：lock(key［Ｓ］)

〈Ｓ〉

unlock(key［Ｓ］)

Goto A

PB

B：lock(key ［Ｓ］)

＜Ｓ＞

unlock(key ［Ｓ］)

Goto B

当进程PA执行完unlock(key[S])之后，由于紧接着是一转向语句，进程PA将又立即去执行lock(key[S])过程，永远占着资源。

问题根源：

正如某个学生想使用某个人人都可借用、且不规定使用时间的教室时一样，他首先申请获得使用该教室的权利，然后再到教室看看该教室是不是被锁上了。如果该教室被锁上了，他只好下次再来观察，这种反复将持续到他进门后为止。

解决加锁法所带来的问题的方法：

一种最直观的办法是，设置一个教室管理员。登记未能如愿使用教室学生的名字，并等到教室门一打开则通知该学生进入。这样，既减少了学生多次来去教室检查门是否被打开的时间，又减少了因为学生自发地检查造成的不公平现象。

在操作系统中，这个管理员就是信号量。信号量管理相应临界区的公有资源，它代表可用资源实体。

信号量概念和ＰＶ原语

荷兰科学家E.W.Dijkstra提出来的。信号是铁路交通管理中的一种常用设备，交通管理人员利用信号颜色的变化来实现交通管理。

在操作系统中，信号量sem是一整数。在sem大于等于零时代表可供并发进程使用的资源实体数，但sem小于零时则表示正在等待使用临界区的进程数。用于互斥的信号量sem的初值应该大于零。

2. Ｐ，Ｖ原语

信号量的数值仅能由Ｐ，Ｖ原语操作改变(Ｐ和Ｖ分别是荷兰语 Passeren 和Verhoog 的头一个字母，相当于英文的pass和increment的意思)。

采用Ｐ，Ｖ原语，可以把类名为Ｓ的临界区描述为When Ｓ do Ｐ(sem)临界区Ｖ(sem)od。

sem：临界区内使用的公用资源有关的信号量。

一次Ｐ原语操作：信号量sem减1；

一次Ｖ原语操作：使得信号量sem加1。

当有好几个进程执行Ｐ原语未通过而进入等待状态之后，如有某进程作了Ｖ原语操作，则等待进程中的一个可以进入临界区，但其他进程必须等待

Ｐ原语操作的主要动作是：

(1) sem减 1；

(2) 若sem减1后仍大于或等于零，则进程继续执行；

(3) 若sem减1后小于零，则该进程被阻塞后与该信号相对应的队列中，然后转进程调度。

Ｐ原语操作的功能框图如图3.12。

图3.12 Ｐ原语操作功能 图 3.13Ｖ原语操作功能

Ｖ原语的操作主要动作是：

(1) sem加1；

(2) 若相加结果大于零，进程继续执行；

(3) 若相加结果小于或等于零，则从该信号的等待队列中唤醒一等待进程，然后再返回原进程继续执行或转进程调度。

Ｖ原语操作的功能框图如图3.13。

为什么Ｐ，Ｖ过程要以原语方式实现？

如果不这样，当多个进程同时调用Ｐ操作或Ｖ操作的话，则有可能在Ｐ操作刚把sem-1而未把对应进程送入等待队列时，Ｖ操作开始执行。从而，Ｖ操作将无法发现等待进程而返回。

因此，Ｐ，Ｖ操作都必须以原语实现，且在Ｐ，Ｖ原语执行期间不允许中断发生。

不同操作系统PV原语实现方法不同

P操作—等待操作(wait) V操作—唤醒操作(rouse)

wait()，down() signal()，up()

Ｐ、Ｖ原语实现方法：有多种

一种使用加锁法的软件实现方法：

Ｐ（sem）：

begin

封锁中断；

lock(lockbit)

val［sem］=val［sem］-1

if val［sem］<0

保护当前进程CPU现场

当前进程状态置为″等待″

将当前进程插入信号sem等待队列

转进程调度

fi

unlock(lockbit)；开放中断

end

Ｖ（sem）：

begin

封锁中断；

lock(lockbit)

va［sem］=val［sem］+1

if val［sem］≤0

localk

从sem等待队列中选取一等待进程，将其指针置入k中

将k插入就绪队列

进程状态置“就绪”

fi

unlock(lockbit)；开放中断

end

3.5.4 用Ｐ，Ｖ原语实现进程互斥

利用Ｐ，Ｖ原语和信号量，可以方便地解决并发进程的互斥问题，而且不会产生使用加锁法解决互斥问题时所出现的问题。

设信号量sem是用于互斥的信号量，且其初值为1。

当一个进程想要进入临界区时，它必须先执行Ｐ原语操作以将信号量sem减1。如果在该进程未执行Ｖ原语操作之前如有另一进程想进入临界区，它也应先执行 Ｐ 原语操作，从而使sem 的值变为-1，因此，第二个进程将被阻塞。直到第一个进程执行Ｖ原语操作之后，执行Ｖ原语操作以释放它所占用的临界区，sem 的值变为0，从而唤醒第二个进程进入就绪队列。

用信号量实现两并发进程PA，PB互斥的描述如下：

1) 设定信号量初始值

设 sem为互斥信号量，其取值范围为(1,0,-1)。

sem=1：表示进程PA和PB都未进入类名为Ｓ的临界区；

sem=0：表示进程PA或PB已进入类名为Ｓ的临界区；

sem=-1：表示进程PA和PB中，一个进程已进入临界区，而另一个进程等待进入临界区。

特别说明：

信号量初始值表示初始资源数目，不一定为1。

2) PV操作描述：

PA：

Ｐ(sem)

〈Ｓ〉

Ｖ(sem)

：

：

PB：

Ｐ(sem)

〈Ｓ〉

Ｖ(sem)

：

：

用ＰＶ原语实现进程互斥问题举例：

例1：“过独木桥”问题

同一方向的行人每次只能一人过桥，当某一方向有人过桥时，另一方向的行人必须等待；当某一方向无人过桥时，另一方向的行人可以过桥。

解答：

将独木桥的两个方向分别标记为A和B，PA表示A方向过独木桥的进程， PB表示B方向过独木桥的进程，信号量mutex 用来实现两个方向行人对临界资源独木桥的互斥使用，初始mutex=1，表示独木桥一次过一人，刚开始为空。

A 方向行人的动作描述为：

PA ：

P(mutex);

过独木桥;

V(mutex);

B 方向行人的动作描述为：

PB ：

P(mutex);

过独木桥;

V(mutex);

3.6 进程同步

3.6.1 同步的概念

问题

除了对公有资源的竞争而引起的间接制约(互斥)之外，并发进程之间是否还存在着其他制约关系影响执行速度呢？看下面的例子。

计算进程和打印进程共同使用同一缓冲区Buf。计算进程反复地把每次计算结果放入 Buf中，而打印进程则把计算进程每次放入 Buf中的数据通过打印机打印输出。

其相应的控制段可以描述为：

PC : Buffer中数据→Buf

A: local Buffer

Repeat

检查Buf

Until Buf=空

计算

得到计算结果

Buf ← 计算结果(Buffer中)

Goto A

PP: Buf中数据→打印

B: local Pri

Repeat

检查 Buf

Until Buf ≠ 空

Pri ← Buf

打印 Pri中的数据

清除 Buf中的数据

Goto B

分析：

如果假定进程PC和PP对公用缓冲区Buf 已采取了互斥措施。

因为其中包含有二处反复测试语句，会造成CPU时间的极大浪费。CPU 时间的浪费主要是由于进程PC和PP的执行互相制约所引起的，PC的输出结果是PP的执行条件，反过来，PP的执行结果也是PC的执行条件。

一组在异步环境下的并发进程，各自的执行结果互为对方的执行条件，从而限制各进程的执行速度的过程称为并发进程间的直接制约。

解决方法

直接制约的进程互相给对方进程发送执行条件已经具备的信号。

被制约进程即可省去对执行条件的测试，只要收到了制约进程发来的信号便开始执行，而在未收到制约进程发来的信号时便进入等待状态。

同步概念

把异步环境下的一组并发进程，因直接制约而互相发送消息而进行互相合作、互相等待，使得各进程按一定的速度执行的过程称为进程间的同步。

具有同步关系的一组并发进程称为合作进程，

合作进程间互相发送的信号称为消息或事件。

过程wait(消息名）:

表示进程等待合作进程发来的消息

过程signal（消息名）:

表示向合作进程发送消息

利用过程wait和signal描述进程PC和打印进程PP的同步关系：

(1) 设消息名Bufempty表示Buf空，消息名Buffull表示Buf中装满了数据。

(2) 初始化Bufempty=true，Buffull=false 。

(3) 描述： 计算进程PC → Buf → 打印进程PP

3.6.2 私用信号量

前面用wait（消息名）与signal（消息名）的方式，描述了进程同步的一种实现方法。

使用 3.5节介绍的信号量的方法也可实现进程间的同步。

把各进程之间发送的消息作为信号量看待。与进程互斥时不同的是，这里的信号量只与制约进程及被制约进程有关而不是与整组(所有)并发进程有关。因此，称该信号量为私用信号量（Private Semaphvre），同步时使用。

与私用信号量相对应，称互斥时使用的信号量为公用信号量。

3.6.3 用Ｐ，Ｖ原语操作实现同步

步骤和方法

利用Ｐ，Ｖ原语实现进程同步的方法与利用wait和signal过程时相同，也是分为三步。

(0) 画出各进程的流程图，分析哪些步骤可能存在互斥同步的问题；

(1) 为各并发进程设置私用信号量；

(2) 为私用信号量赋初值；

(3) 利用Ｐ，Ｖ原语和私用信号量规定各进程的执行顺序。

用ＰＶ原语实现进程同步问题举例

例1：通过缓冲区队列发送和接收消息问题

设进程PA和PB通过缓冲区队列传递数据（如图3.14）。

PA为发送进程，PB为接收进程。

PA发送数据时调用发送过程deposit(data)，PB接收数据时调用过程remove(data)。且数据的发送和接收过程满足如下条件：

(1) 在PA至少送一块数据入一个缓冲区之前，PB不可能从缓冲区中取出数

据（假定数据块长等于缓冲区长度）；

（2) PA往缓冲队列发送数据时，至少有一个缓冲区是空的；

（3) 由PA发送的数据块在缓冲队列中按先进先出（FIFO）方式排列。

解：由题意可知，进程PA调用的过程deposit(data)和进程PB调用的过程remove(data)必须同步执行，因为过程 deposit(data)的执行结果是过程remove(data)的执行条件，而当缓冲队列全部装满数据时，remove(data)的执行结果又是deposit(data)的执行条件，满足同步定义。

按以下三步描述过程deposit(data)和remove(data)：

（1) 设Bufempty为进程PA的私用信号量，Buffull 为进程PB的私用信号量；

（2) 令Bufempty的初始值为n（n 为缓冲队列的缓冲区个数），Buffull 的初始值为0；

（3) 描述：

PA: deposit(data):

begin local x

P(Bufempty)；

按FIFO方式选择一个空缓冲区Buf(x)；

Buf(x)← data

Buf(x)置满标记

V(Buffull)

end

PB: remove(data):

Begin local x

P(Buffull)；

按FIFO方式选择一个装满数据的缓冲区Buf(x)

data ← Buf(x)

Buf(x)置空标记

V(Bufempty)

end

这里，局部变量x用来指明缓冲区的区号，给Buf(x)置标志位是为了便于区别和搜索空缓冲区及非空缓冲区。

思考：在该题中需要考虑互斥吗？为什么？如果每次只允许一个进程对缓冲队列进行操作时怎么办？

例2：生产者-消费者问题（producer-consumer problems）

生产者-消费者问题是并发进程的同步和互斥问题一般化后得到的一般抽象模型。

系统中使用某一类资源的进程称为该资源的消费者，而把释放同类资源的进程称为该资源的生产者。

资源既可以是像外设、内存及缓冲区等硬件资源，也包括临界区、数据、例程等软件资源。

图3.15 生产者-消费者问题

生产者生产者进程deposit(data)和消费者进程remove(data)是一个同步问题。需满足如下条件：

(1) 消费者想接收数据时，有界缓冲区中至少有一个单元是满的；

(2) 生产者想发送数据时，有界缓冲区中至少有一个单元是空的。

另外，由于有界缓冲区是临界资源，因此，各生产者进程和各消费者进程之间必须互斥执行。

解：

(1) 设公用信号量mutex保证生产者进程和消费者进程之间的互斥，表示可用有界缓冲区的个数，初值为 1；设信号量avail为生产者进程的私用信号量，表示有界缓冲区中的空单元数，初值为n；信号量full为消费者进程的私用信号量，表示有界缓冲区中非空单元数，初值为0。

(2) 描述如下：

注意：

由于一个过程中包含有几个公用、私用信号量，因此，Ｐ、Ｖ原语的操作次序要非常小心。

一般说来，由于Ｖ原语是释放资源的，所以可以以任意次序出现。

但Ｐ原语则不然，如果次序混乱，将会造成进程之间的死锁。关于死锁，将在3.8 中介绍。

void producer(void)

{ int item;

while (TRUE)

{ item=produce\_item();

down(&avail);

down(&mutex);

insert\_item(item);

up(&mutex);

up(&full);

}

}

例3：“过独木桥”问题复杂化

同一方向的行人可连续过桥。当某一方向有人过桥时，另一方向的行人必须等待；当某一方向无人过桥时，另一方向的行人可以过桥。

解答：

将独木桥的两个方向分别标记为A和B，PA表示A方向过独木桥的进程， PB表示B方向过独木桥的进程，并用整型变量countA,countB 分别表示A、B 方向上已在独木桥上的行人数，它们的初值为0。用mutex 实现两个方向行人对独木桥的互斥使用，初始mutex=1，表示独木桥刚开始为空；MA用来实现对countA的互斥访问，MB用来实现对countB的互斥访问。

问题：

如果允许同方向在独木桥上的行人数最多为5人，

如何修改？

例4：超市收银员和顾客问题

在某超市里有一个收银员，且同时最多允许有n个顾客购物，可以将顾客和收银员看成是两类不同的进程，且工作流程如下页图所示。

为了利用PV操作正确地协调这两类进程之间的工作，设置了二个信号量实现同步：S1=0 (收银员进程私用信号量)、S2=0 (顾客进程私用信号量)；一个信号量(实现互斥)Sn=n(超市最多允许进入的顾客数目)。

说明：()中注释是自己加的，题目中无。

(1) A. P(S1) B. P(S2)

C. P(Sn) D. P(Sn)、P(S1)

(2) A. P(Sn)、V(S2) B. P(Sn)、V(S1)

C. P(S2)、V(S1) D. V(S1)、P(S2)

(3) A. P(S1)、V(S2) B. P(Sn)、V(S1)

C. P(S2)、V(S1) D. V(S1)、P(S2)

答案：

(1) C

(2) D

(3) A

例5：哲学家问题—The Dining Philosophers Problem（P78习题3.14）

In 1965, Dijkstra posed and solved a synchronization problem he called the dining philosophers problem.  
 Five philosophers are seated around a circular table. Each philosopher has a plate of spaghetti. The spaghetti is so slippery that a philosopher needs two forks to eat it. Between each pair of plates is one fork. The life of a philosopher consists of alternate periods of eating and thinking.

3.6.4 管程(monitor) （补充内容，书上无，了解即可）

• 管程概念的提出

管程由Hoare (1974) and Brinch Hansen (1975)提出。

信号量机制功能强大，但使用时对信号量的操作分散，而且难以控制，读写和维护都很困难。因此后来又提出了一种集中式同步进程——管程。

其基本思想是将共享变量和对它们的操作集中在一个模块中，操作系统或并发程序就由这样的模块构成。这样模块之间联系清晰，便于维护和修改，易于保证正确性。   
 从语言的角度看，管程主要有以下特性：  
   (1) 模块化。管程是一个基本程序单位，可以单独编译；   
   (2) 抽象数据类型。管程中不仅有数据，而且有对数据的操作（类似于面向对象的类）；   
   (3) 信息掩蔽。管程外可以调用管程内部定义的一些函数，但函数的具体实现外部不可见。

对于管程中定义的共享变量的所有操作都局限在管程中，外部只能通过调用管程的某些函数来间接访问这些变量。因此管程有很好的封装性。  
 为了保证共享变量的数据一致性，管程应互斥使用。

管程通常是用于管理资源的，因此管程中有进程等待队列和相应的等待和唤醒操作。在管程入口有一个等待队列，称为入口等待队列。当一个已进入管程的进程等待时，就释放管程的互斥使用权；当已进入管程的一个进程唤醒另一个进程时，两者必须有一个退出或停止使用管程。在管程内部，由于执行唤醒操作，可能存在多个等待进程(等待使用管程)，称为紧急等待队列，它的优先级高于入口等待队列。

• 管程工作原理

因此，一个进程进入管程之前要先申请，一般由管程提供一个enter过程；离开时释放使用权，如果紧急等待队列不空，则唤醒第一个等待者，一般也由管程提供外部过程leave。  
 管程内部有自己的等待机制。管程可以说明一种特殊的条件型变量：var c:condition；实际上是一个指针，指向一个等待该条件的PCB队列。对条件型变量可执行wait和signal操作：  
     wait(c)—若紧急等待队列不空，唤醒第一个等待者，否则释放管程使用权。执行本操作的进程进入c队列尾部；  
     signal(c)—若c队列为空，继续原进程，否则唤醒队列第一个等待者，自己进入紧急等待队列尾部。

管程操作及实例

Cwait(Conditional\_Variable\_Name)—等待操作  
 Csignal(Conditional\_Variable\_Name)—唤醒操作

管程实例1：分配和释放资源

管程如下：

Monitor Resource\_Allocator ;

Var Resource\_In\_Use: Boolean;

Resource\_Is\_Free: Condition;

Procedure Get\_Resource; 请求资源

Begin

if Resource\_In\_Use Then Cwait(Resource\_Is\_Free);

Resource\_In\_Use:=True

End;

Procedure Return\_Resource; 释放资源

Begin

Resource\_In\_Use:=False;

Csignal(Resource\_Is\_Free);

End;

Begin

Resource\_In\_Use:=False;

End;

用户程序如下：

…

Resource\_Allocator. Get\_Resource;

使用资源；

Resource\_Allocator. Return\_Resource;

管程实例2：生产者与消费者

管程如下：

Monitor Ring\_Buffer\_Monitor;

Var Ring\_Buffer: Array[0..Slots-1] of Stuff;

Slot\_In\_Use: 0..Slots; 可用空单元号(=slots时为满)

Next\_Slot\_To\_Fill: 0..Slots-1; 空单元已经使用几个

Next\_Slot\_To\_Empty: 0..Slots-1;

Ring\_Buffer\_Has\_Data, Ring\_Buffer\_Has\_Space: Condition;

Procedure Fill\_A\_Slot(Slot\_Data: Stuff); 生产者

Begin

if Slot\_In\_Use=Slots Then Cwait(Ring\_Buffer\_Has\_Space);

Ring\_Buffer[Next\_Slot\_To\_Fill]:=Slot\_Data;

Slot\_In\_Use:=Slot\_In\_Use+1;

Next\_Slot\_To\_Fill:=(Next\_Slot\_To\_Fill+1) mod Slots; 下个空单元号

Csignal(Ring\_Buffer\_Has\_ Data);

End;

Procedure Empty\_A\_Slot(Var Slot\_Data: Stuff); 消费者

Begin

if Slot\_In\_Use=0 Then Cwait(Ring\_Buffer\_Has\_ Data);

Slot\_Data:= Ring\_Buffer[Next\_Slot\_To\_Empty];

Slot\_In\_Use:=Slot\_In\_Use-1;

Next\_Slot\_To\_ Empty:=(Next\_Slot\_To\_Empty+1) mod Slots;

Csignal(Ring\_Buffer\_Has\_Space);

End;

Begin

Slot\_In\_Use:=0;

Next\_Slot\_To\_Fill:=0;

Next\_Slot\_To\_Empty:=0;

End;

3.7 进程通信

通信(communication): 进程间传送数据。

根据通信内容可以分为二种：

• 低级通信—进程间控制信息的交换。一般只传

送一个或几个字节的信息，以达到

控制进程执行速度的作用。

 高级通信—进程间大批量数据的交换，目的交换

信息。

3.7.1 进程的通信方式

在单机系统中，进程间通信可分为4种形式：

(1) 主从式；

(2) 会话式；

(3) 消息或邮箱机制；

(4) 共享存储区方式。

主从式通信系统

主要特点：

① 主进程可自由地使用从进程的资源或数据；

② 从进程的动作受主进程的控制；

③ 主进程和从进程的关系是固定的。

典型例子:终端控制进程和终端进程。

(2) 会话系统

通信进程双方可分别称为使用进程和服务进程。

其中，使用进程调用服务进程提供的服务。

特点：

① 使用进程在使用服务进程所提供的服务之前，必须得到服务进程的许可；

② 服务进程根据使用进程的要求提供服务，但对所提供服务的控制由服务进程自身完成。

③ 使用进程和服务进程在通信时有固定连接关系。

典型例子：用户进程与磁盘管理进程。

各用户进程向磁盘管理进程提出使用要求后，才可以使用相应的存储区。由磁盘管理进程自身完成对磁盘存储区的管理和控制。另外，只有在用户进程要求使用磁盘存储区时才有通信关系。

(3) 消息或邮箱机制(3.7.2、3.7.3细讲)

无论接收进程是否已准备好接收消息，发送进程都将把所要发送的消息送入缓冲区或邮箱。

消息由个部分组成：发送进程名、接收进程名、数据和有关数据的操作（图3.16）。

特点：

① 只要存在空缓冲区或邮箱，发送进程就可以发送消息。

② 与会话系统不同，发送进程和接收进程之间无直接连接关系，接收进程可能在收到某个发送进程发来的消息之后，又转去接收另一个发送进程发来的消息。

③ 发送进程和接收进程之间存在缓冲区或邮箱（图3.17）用来存放被传送消息。

图3.16消息的组成

图3.17缓冲区或邮箱通信结构

(4) 共享存储区方式

特点：

不要求数据移动。两个需要互相交换信息的进程通过对同一共享数据区（shared memory）的操作来达到互相通信的目的。这个共享数据区是每个互相通信进程的一个组成部分。

具体工作原理和技术细节有兴趣参看补充材料：

“分布式系统中消息中间件的客户机的设计和实现”

3.7.2 消息缓冲机制

工作过程（参见下页图）

发送进程在发送消息前，先在自己的内存空间设置一个发送区，把欲发送的消息填入其中，然后再用发送过程将其发送出去；接收进程则在接收消息之前，在自己的内存空间内设置相应的接收区，然后用接收过程接收消息。

由于使用的缓冲区(队列)为公用缓冲区，两通信进程必须满足如下条件：

① 在发送进程把消息写入缓冲区和把缓冲区挂入消息队列时，应禁止其他进程对该缓冲区消息队列的访问。当接收进程正从消息队列中取消息缓冲时，也应禁止其他进程对该队列的访问。

② 当缓冲区中无消息存在时，接收进程不能接收到任何消息。

发送进程能是发送消息，取决于是否申请到缓冲区

消息发送与接收

描述

设公用信号量mutex 为控制对缓冲区(队列)访问的互斥信号量，其初值为1 ；

设SM为接收进程的私用信号量，表示等待接收的消息个数，其初值为0 。

设发送进程调用过程send(m)将消息m 送往缓冲区，接收进程调用过程Receive(m)将消息m从缓冲区读往自己的数据区，则Send(m)和Receive(n)可分别描述为：

Send(m):

begin

向系统申请一个消息缓冲区

Ｐ（mutex） 实现互斥

将发送区消息m送入新申请的消息缓冲区

把消息缓冲区挂入接收进程的消息队列

Ｖ（mutex）

Ｖ（SM） 实现同步

end

Receive(n):

begin

Ｐ（SM）

Ｐ（mutex）

摘下消息队列中的消息n

将消息n从缓冲区复制到接收区

释放缓冲区

Ｖ（mutex）

end

为什么发送进程不设私有信号量，实现和接收进程的同步？

尽管系统中可利用的缓冲区总数是已知的，但由于消息队列是按接收进程排列，因而，在同一时间内，系统中存在着多个消息队列；且这些队列的长度是不固定的。因此，发送进程无法在Send过程用Ｐ操作判断信号量SM。

3.7.3 邮箱通信

工作过程

发送进程把消息送往邮箱，接收进程从邮箱中取出消息，从而完成进程间信息交换。

邮箱由邮箱头和邮箱体组成。其中邮箱头描述邮箱名称、邮箱大小、邮箱方向以及拥有该邮箱的进程名等。邮箱体主要用来存放消息（图3.18）。

图3.18 邮箱通信结构

对于只有一发送进程和一接收进程使用的邮箱，则进程间通信应满足如下条件：

① 发送进程发送消息时，邮箱中至少要有一个空格能存放该消息。

② 接收进程接收消息时，邮箱中至少要有一个消息存在。

描述

信号量fromnum 为发送进程的私用信号量，初值为信箱的空格数 n；

信号量mesnum为接收进程的私用信号量，初值为 0；

为什么调用过程deposit(m)的进程与调用过程remove(m)的进程之间存在着同步制约关系而不是互斥制约关系？

如果存在着多个发送进程和多个接收进程共享邮箱的情况。这时需要对过程deposit(m)和remove(m)作相应的改动。

3.7.4 进程通信实例——用户终端和控制台的通信(了解)

通用计算机中，包含用户终端和由系统操作员控制的控制台终端。存在如下通信关系：

◇用户终端和控制台通信：各用户进程将消息送到控制台进程，控制台操作员在读到这些消息后对消息进行应答，应答信息返回给用户。两者通信由会话控制进程CCP 控制完成。

◇控制台终端由键盘和显示器组成，由键盘控制进程KCP和显示控制进程和DCP负责通信。

操作员在控制台键盘输入应答信息后，控制台终端内部需同时完成两个工作：(1) 通过echobuf直接在控制台显示器显示；(2) 通过inbuf把应答信息交给CCP，最终返回给用户。

图3.19 用户终端和控制台通信示例

(1) 控制台内部：KCP和DCP的动作

当操作员打键时，KCP将对应的数据从键盘送入输入缓冲inbuf中，同时，也将键入数据送echobuf在显示器上显示。KCP要同时处理这两者情况。

类似，DCP也要同时处理这两者情况。

要解决二方面的同步问题：

(a) 键盘KP和键盘控制进程KCP的同步问题；

(b) 显示器DP和显示器控制进程DCP的同步问题；

键盘KP和键盘控制进程KCP的同步问题

(a-1) 键盘动作KP

设T-Ready 为键盘KP的私用信号量，初值为0；

T-Busy为键盘控制进程KCP的私用信号量，初值为1。

repeat

local x

Ｐ（T-Busy）

把键入字符放入数据传输缓冲x

Ｖ（T-Ready）

until 终端关闭

(a-2) 键盘控制进程KCP

初始化｛清除所有inbuf 和echobuf}

begin

local x

Ｐ（T-Ready）

从键盘数据传输缓冲x中取出字符m记为x.m

Send(x.m)

将x.m送入echonuf

Ｖ（T-Busy）

end

(b) 显示器DP和显示器控制进程DCP的同步问题

(b-1) 显示器控制进程DCP

设D-Ready 为DP的私用信号量，初值为0；

D-Busy为DCP的私用信号量，初值为1；

初始化 {清除输出缓冲outbuf，echo模式置false}

begin

if outbuf满

then

receive(k)/\* CCP k \*/

Ｐ （D-Busy）

把k送入显示器数据缓冲区

Ｖ（D-Ready）

else

echo模式置true

echobuf 中字符置入显示器数据缓冲区

fi

(b-2) 显示器动作DP

repeat

if echo模式

then

打印显示器数据缓冲区中字符

else

Ｐ （D-Ready）

打印显示器数据缓冲区中消息

Ｖ　（D-Busy）

Until 显示器关机

(2) CCP和KCP、DCP的同步问题

CCP主要工作：

CCP从inbuf （KCP提供）中读出消息，并传给用户缓冲队列；

CCP把用户缓冲队列中的消息写入outbuf，并交给DCP处理。

Read(x):

// 把inbuf 中的所有字符读到用户进程数据区x处

Begin

P (inbuf-full) CCP私用信号量，初值为0

Copy(inbuf into x)

Ｖ(inbuf-empty) KCP私用信号量，初值为1

end

Write(y):

// 把用户进程y 处的消息写到outbuf中

begin

Ｐ(outbuf-empty) CCP私用信号量，初值为1

Copy(outbuf from y)

Ｖ(outbuf-full) DCP私用信号量，初值为0

end

(3) CCP与用户进程的互斥和同步

CCP要从各用户进程那里得到提问和向提问的用户进程转达从控制台来的指示。因此，CCP和用户进程之间也存在着通信关系。

各用户进程把消息送入RQ时，必须互斥操作，否则将引起RQ队列混乱。因此，设互斥用信号量rq，初值为1。

另外，CCP只有在用户进程提问之后才负责向控制台转发提问和向用户进程转达控制台的指示。因此，为CCP设置一私用信号量question以计算用户进程所提出的问题数目，初值为0。

同时，设一私用信号量answeri以统计控制台发来的消息个数。

由于各用户进程在CCP发出回答消息之后，不一定马上就能处理CCP所发出的回答消息，因而，需设置相应的消息接收队列SQi。对SQi 队列的操作必须是互斥的，设互斥信号量sqi，其初值为1 。同理，对RQ 队列的操作也是互斥的。

SQi和RQ的关系如图3.20。

图3.20 CCP和用户进程接口

CCP和用户进程的接口可描述如下：

U-receive(m): 从RQ取出一个消息的过程

begin

Ｐ （question）

when rq do从RQ中取出mod 互斥变量rq作为临界区名

return (m)

end

S-answer(a，i):

// 把控制台发来的应答信息存入用户队列

begin

when sqi do把a插入SQi中od

Ｖ （answeri）

end

(4) CCP的动作

利用上述各种接口，完整的会话控制进程CCP如下：

local k，m，x

repeat

U-receive(m)

将消息m 的进程标号置入k中

将消息m解码变换到x

Write(x)

Read(x)

将x编码到m

S-answer(m，k)

until CCP结束

3.7.5 进程通信的实例——管道

1. 管道pipe

UNIX系统从SystemⅤ开始，提供有名管道和无名管道两种数据通信方式，这里介绍无名管道。

无名管道为建立管道的进程及其子孙提供一条以比特流方式传送消息的通信管道。该管道在逻辑上被看作管道文件，在物理上则由文件系统的高速缓冲区构成。

管道按FIFO（先进先出）方式传送消息，且只能单向传送消息（图3.21）。

图3.21 管道通信

建立一条同步通信管道。其格式为：

pipe(fd)

int fd[2]； fd[1] 为写入端，fd[0]为读出端

例1： C语言编程，建立一个pipe，同时父进程生成一个子进程，子进程向pipe中 写入一字符串，父进程从pipe中读出该字符串。

解：#include <stdio.h>

main()

{

int x，fd[2]；

char buf[30]，s[30]；

pipe(fd)； /\*创建管道\*/

while((x=fork())==-1)； /\*创建子进程失败时，循环\*/

if(x==0)

{

sprintf(buf，″This is an example＼n″)；

write(fd[1]，buf，30)； /\*把buf中字符写入管道\*/

exit(0)；

}

else /\*父进程返回\*/

{

wait(0)；

read(fd[0]，s，30)； /\*父进程读管道中字符\*/

printf(″%s″，s)；

}

}

例2： 编程，建立一个管道。父进程生成子进程P1，P2，这两个子进程分别向管道中写入各自的字符串，父进程读出它们（如图3.22）。

图3.22 父进程和子进程P1，P2通信例子

解：程序框图如图3.23所示，源程序如下：

图3.23 例2程序流图

#include <stdio.h>

main()

{

int i，r，p1，p2，fd[2]；

char buf[50]，s[50]；

pipe(fd)； /\*父进程建立管道\*/

while((p1=fork())==-1)；/\*创建子进程P1，失败时循环\*/

if(p1==0)　 /\*由子进程P1返回，执行子进程P1\*/

{

lockf(fd[1]，1，0)； /\*加锁锁定写入端\*/

sprintf(buf，″child process P1 is sending messages!＼n″)；

printf(″child processP1!＼n″)；

write(fd[1]，buf，50)；/\*把buf中的50个字符写入管道\*/

sleep(5)； /\*睡眠5秒，让父进程读\*/

lockf(fd[1]，0，0)； /\*释放管道写入端\*/

exit(0)； /\*关闭P1\*/

}

else /\*从父进程返回，执行父进程\*/

{

while((p2=fork())==-1)； /\*创建子进程P2，失败时循环\*/

if(p2==0) /\*从子进程P2返回，执行P2\*/

{

lockf(fd[1]，1，0)； /\*锁定写入端\*/

sprintf(buf，″child process P2 is sending messages＼n″)；

printf(″child process P2 ! ＼n″)；

write(fd[1]，buf，50)； /\*把buf中字符写入管道\*/

sleep(5)； /\*睡眠等待\*/

lockf(fd[1]，0，0)； /\*释放管道写入端\*/

exit(0)； /\*关闭P2\*/

}

wait(0)；

if(r=read(fd[0]，s，50)==-1) printf(″can′t read pipe＼n″)；

else printf(″%s＼n″,s)；

wait(0)；

if(r=read(fd[0]，s，50)==-1) printf(″can′t read pipe＼n″)；

else printf(″%s＼n″,s)；

exit(0)；

}

}

3.8 死锁问题

3.8.1 死锁的概念

1. 死锁的定义

定义：各并发进程彼此互相等待对方所拥有的资源，且这些并发进程在得到对方的资源之前不会释放自己所拥有的资源。从而造成大家都想得到资源而又都得不到资源，各并发进程不能继续向前推进的状态。

图3.24 死锁的概念

例子: 生产者/消费者

设生产者进程已获得对缓冲区队列的操作权，要进一步对某一空缓冲区进行置入操作。然而，此时缓冲队列内所有缓冲区都是满的，即只有消费者进程才能对它们进行取消息操作。因此，生产者进程进入等待状态。

反过来，消费者进程拥有对各缓冲区操作的操作权，为了对各缓冲区进行操作，它又要申请对缓冲队列操作的操作权。由于对缓冲队列的操作权被生产者进程掌握，且生产者进程不会自动释放它，从而消费者进程也只能进入等待状态而陷入死锁。

2. 死锁的起因

根本原因：

系统提供的资源个数少于并发进程所要求的该类资源数。

根本解决办法：

无。因为资源总是有限的，不可能为所有要求资源的进程无限制地提供资源。

可以采用适当的资源分配算法，以达到消除死锁的目的。

3. 产生死锁的必要条件

(1) 互斥条件。资源不能同时被两个以上进程使用或操作，进程对它所需要的资源进行排他性控制。

(2) 不剥夺条件。进程所获得的资源在未使用完毕之前，不能被其他进程强行剥夺，而只能由获得该资源的进程自己释放。

(3) 部分分配。进程每次申请它所需要的一部分资源，在等待新资源的同时继续占用已分配到的资源。

(4) 环路条件。存在一种进程循环链，链中每一个进程已获得的资源同时被下一个进程所请求。

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。

3.8.2 死锁的排除方法

三种：

• 预防(prevention)——静态、按序、剥夺式(分配资源)

 避免(avoidance)——动态，实际分配资源前进行预分配

采用银行家算法(E.W.Dijkstra1968年)

 检测(detection)——采用资源分配图RAG

 恢复(recovery)

在实际操作系统中大都使用检测与恢复法。

1. 死锁预防

• 基本思路：静态

打破四个条件中的某一个。

方法1：打破资源的互斥和不可剥夺这两个必要条

件，允许进程同时访问某些资源，做不到；

方法2：打破资源的部分分配这个必要条件。预先分

配进程所需要的全部资源。但缺点很多。

方法3：打破死锁的环路条件。即把资源分类按顺序

排列，使进程在申请、保持资源时不形成环

路。缺点：限制了进程对资源的请求，而且

对资源的分类编序耗费系统开销。

2. 死锁避免

• 基本思路：动态预防

在实际分配资源之前进行预分配，预测出死锁发生的可能性并加以避免。

Banker’s Algorithm（银行家算法）－补充知识

Dijstra1968年提出，最著名基于动态预防的死锁避免方法

详见补充文档 “银行家算法详解”

基本思路

在运行过程中，当有进程申请资源时，给进程实际分配资源之前进行资源预分配，然后判断预分配后所有进程能否顺利执行完成。如果能完成则实际分配，否则不分配，达到避免死锁的目的。

银行家算法的数据结构

n 为进程的数目；

m 为资源类型的数目；

Available[m]： 每类资源可用数量

Max[n, m]： 每个进程Pi可以请求资源Rj的最大数目

Allocation[n, m]：进程Pi当前分配了资源Rj的数目

Need[n, m]： 进程Pi还需要资源Rj的最大数目

Need [i,j] = Max[i,j] – Allocation [i,j].

进程Pi的资源分配算法

Requesti—— 表示进程 Pi的申请向量。

If Requesti [j] = k, 表示进程 Pi需要申请 k个 Rj类资源

1. If Requesti ≤ Needi go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim.

2. If Requesti ≤ Available, go to step 3. Otherwise Pi must wait, since resources are not available.

3. Pretend to allocate requested resources to Pi by modifying the state as follows:

Available := Available – Requesti;

Allocationi := Allocationi + Requesti;

Needi := Needi – Requesti;

If safe  the resources are allocated to Pi.

If unsafe  Pi must wait, and the old resource-allocation state is restored.

Safety Algorithm（安全算法）

1. 设Work和Finish作为长度为m和n的向量

Work := Available

Finish [i] = false for i=1, 2, …, n.

2. Find an i such that both: （找到i）

(a) Finish [i] = false

(b) Needi ≤ Work

If no such i exists, go to step 4.

3. Work := Work +Allocationi（进程i释放所占的全部资源）

Finish[i] := true 说明进程i不会死锁

go to step 2.

4. If Finish [i] = true for all i, then the system is in a safe state.

Basic Facts（基本事实）

If a system is in safe state ⇒ no deadlocks.

(如果一个系统在安全状态，就没有死锁)

If a system is in unsafe state  possibility of deadlock.

(如果一个系统不是处于安全状态，就有可能死锁)

Avoidance  ensure that a system will never enter an unsafe state.

(避免：确保系统永远不会进入死锁状态)

银行家算法的例子

4个进程P1到P4；

3个资源类型A(9个)，B(3个)，C(6个)

假设时刻T0各进程分配资源如下图1所示：

用前述安全算法，可以计算得知，在时刻T0，存在安全序列P2、P1、P3、P4，确保系统处于安全状态，计算最后一步看到的结果如下图2所示：

回到图1状态，如果此时进程P1再请求资源， Request1= (1, 0, 1)，系统按银行家算法检查：

(1) Request1(1, 0, 1)<=Need(2,2,2)

(2) Request1 (1, 0, 1)<=Available(1,1,2)

(3) 假设系统把资源分配给进程P1，修改后数据见下表红字。

(4) 再按安全性算法检查，发现新状态是不安全的，决定对P1进程不分配资源，P1进程进入等待状态。

3. 死锁的检测和恢复

当进程进行资源请求时，死锁检测算法检查并发进程组是否构成资源的请求和保持环路。

RAG及等待图(RAG变型)、有限状态转移图和petriNet等技术都可用来有效地判断死锁发生。

死锁的恢复办法较多。最简单的办法是终止各锁住进程，或按一定的顺序中止进程序列，直至已释放到有足够的资源来完成剩下的进程时为止。另外，也可以从被锁住进程强迫剥夺资源以解除死锁。

■ 用RAG图检测死锁-补充知识(要求掌握)

Resource-Allocation Graph(资源分配图)

A set of vertexes V and a set of edges E.

(一个顶点的集合V和边的集合E)

• V is partitioned into two different types of nodes:

P = {P1, P2, …, Pn}， the set consisting of all the active processes in the system.

R = {R1, R2, …, Rm}，the set consisting of all resource types in the system.

• request edge——a directed edge Pi → Rj (请求边)

• assignment edge——a directed edge Rj → Pi (分配边)

Example of a Resource-Allocation Graph  
资源分配图的例子(无环路不会死锁)

■ 死锁定理——判定死锁的基本法则

假定对于进程集合P在某个时刻t关于资源集合R的使用状态S(t)可用相应的RAG表示。

1. 若RAG中没有任何环路，则S(t)为非死锁状态。

2. 若RAG中出现环路，且该环路中的各资源均为单一资源，则S(t)为死锁状态。

3. 若RAG中出现环路，但该环路中各资源不全为单一资源，则S(t)不一定是死锁状态。

由死锁定理可知：RAG中的环路是产生死锁的必要条件。

死锁定理的另一种描述：状态S(t)为死锁状态的充要条件是：当且仅当S(t)相应的RAG是不可完全化简的。

■ RAG化简方法

1. 从RAG中找一个只有分配边，或虽有请求边但该请求能立即得到满足的非孤立节点Pi。

2. 去掉Pi的全部有向边，使之成为孤立节点。

3. 假定Rj是Pi释放的某个资源节点，则另一进程Pk关于Rj的请求边可转换成分配边。

4. 如果经一系列转换后，Pk只有分配边，则再使Pk成为孤立节点。

5. 在实施一系列化简后，若可消除RAG中的全部有向边，使全部进程节点成为孤立节点，则该图是可完全化简的。

而当出现下面三种情况时：

1. 找不到只有分配边的进程节点。

2. 有分配边的进程也有请求边，但该请求不能得到满足。

3. RAG中的有向边都不可能去掉，全部进程节点都不能成为孤立节点。

该资源分配图不可化简，S(t)为死锁状态。

Resource-Allocation Graph With A Cycle But No Deadlock  
有环但没有死锁的资源分配图

Resource-Allocation Graph With A Deadlock  
有死锁的资源分配图

Basic Facts（基本事实）

If graph contains no cycles ⇒ no deadlock.

(如果图没有环，那么不会有死锁)

If graph contains a cycle  (如果图有环)

if only one instance per resource type, then deadlock.

(如果每一种资源类型只有一个实例，那么死锁发生)

if several instances per resource type, possibility of deadlock.

(如果一种资源类型有多个实例，可能死锁)

3.9 线程的概念

3.9.1 为什么引入线程

当一个进程需要另一个实体执行某件事情时，该进程可以派生(fork)一个子进程，让子进程进行处理。

但fork有它自身的一些问题：

(1) fork子进程是父进程的一个拷贝，在有大量fork子进程时，会耗费大量的系统资源。

(2) fork子进程后，父子进程就相对独立了，相互间需要借助IPC来传递信息。并且子进程之间涉及复杂的上下文切换，花费大量时间。

3.9.2 线程的基本概念

引入线程主要是为了提高系统的执行效率，减少处理机的空转时间和调度切换（保护现场信息）的时间，以及便于系统管理。

一个进程内的基本调度单位称为线程或轻权进程 (thread or light weight process)。

3.9.3 线程和进程的区别

(1) 线程属于某一个进程，并与进程内的其他线程

一起共享进程的资源。而不同的进程拥有不同的虚拟地

址空间。

(2) 线程控制表TCB所含状态信息比PCB少得多。

(3) 发生进程切换与发生线程切换时相比较，进程切换时将

涉及到有关资源指针的保存以及地址空间的变化等问

题，线程切换时，由于同一进程内的线程共享资源和地

址空间，将不涉及资源信息的保存和地址变化问题，从

而减少了操作系统的开销时间。而且，进程的调度与切

换都是由操作系统内核完成，而线程则既可由操作系统

内核完成，也可由用户程序进行。

多线程系统中进程与线程的关系如图3.25。

3.9.4 线程的适用范围

并不是在所有的计算机系统中线程都是适用的。在那些很少做进程调度和切换的实时系统、个人数字助理系统中，由于任务的单一性，设置线程相反会占用更多的内存空间和寄存器。

使用线程的最大好处是在有多个任务需要处理机处理时，减少处理机的切换时间；而且，线程的创建和结束所需要的系统开销也比进程的创建和结束要小得多。由此，可以推出最适合使用线程的系统是多处理机系统。在多处理机系统中，同一用户程序可以根据不同的功能划分为不同的线程，放在不同的处理机上执行。

几种典型的应用是：

(1) 服务器中的文件管理或通信控制。由于服务器同时可能接受许多个文件访问要求，则系统可以同时生成多个线程来进行处理。

(2) 前后台处理。线程可被用来减少处理机切换时间和提高执行速度。

(3) 异步处理。程序中的两部分如果在执行上没有顺序规定，则这两部分程序可用线程执行。

3.10 线程分类与执行

3.10.1 线程的分类

线程的两个基本类型是：用户级线程和系统级线程(核心级线型)。

用户级线程（user level threads）

管理过程全部由用户程序完成，操作系统内核只对进程进行管理。

操作系统提供一个在用户空间执行的线程库。该线程库提供创建、调度、撤销线程功能。同时该线程库也提供线程间的通信，线程的执行以及存储线程上下文的功能。

(2) 核心级线程（Kernel-level Threads）

由操作系统内核进行管理。操作系统内核给应用程序提供相应的系统调用和应用程序接口API，以使用户程序可以创建、执行、撤消线程。

与用户线程不同，核心级线程既可以被调度到一个处理机上并发执行，也可以被调度到不同的处理机上并行执行。

与用户级线程相比，核心级线程的上下文切换时间要大于用户级线程的上下文切换时间。图3.29给出了用户级线程、系统级线程，以及进程进行上下文切换时的各自的时间开销。

图3.29 线程、进程等的上下文切换开销

3.10.2 线程的执行特性

• 3个基本状态

执行、就绪和阻塞。

没有进程中的挂起状态。因为线程是一个只与内存和寄存器相关的概念，它的内容不会因交换而进入外存。

• 5种基本操作转换线程的状态

(1) 派生（spawn）：线程可由进程派生，也可由线程派生。用户一般用系统调用（或相应的库函数）派生自己的线程。

(2) 阻塞（Block）

(3) 激活（unblock）

(4) 调度（schedule）

(5) 结束（Finish）

线程的状态和操作关系如图3.30。

图3.30 线程的状态与操作

在某些情况下，某个线程被阻塞也可能导致该线程所属的进程被阻塞。

由于同一进程中的所有线程共享该进程的所有资源和地址空间，任何线程对资源的操作都会对其他相关线程带来影响。因此，系统必须为线程的的执行提供同步控制机制。

线程中所使用的同步控制机制与进程中所使用的同步控制机制相同。

习题

3.1 有人说，一个进程是由伪处理机执行的一个程序，这话对吗？为什么？

3.2 试比较进程和程序的区别。

3.3 我们说程序的并发执行将导致最终结果失去封闭性。这话对所有的程序都成立吗？试举例说明。

3.4 试比较作业和进程的区别。

3.5 UNIX System V中，系统程序所对应的正文段未被考虑成进程上下文的一部分，为什么？

3.6 什么是临界区？试举一临界区的例子。

3.7 并发进程间的制约有哪两种？引起制约的原因是什么？

3.8 什么是进程间的互斥？什么是进程间同步？

3.9 试比较Ｐ，Ｖ原语法和加锁法实现进程间互斥的区别。

3.10 设在书 3.6节中所描述的生产者-消费者问题中，其缓冲部分为m个长度相等的有界缓冲区组成，且每次传输数据长度等于有界缓冲区长度以及生产者和消费者可对缓冲区同时操作。重新描述发送过程deposit(data)和接收过程remove(data)。

3.11 两进程PA，PB通过两FIFO缓冲区队列连接（如图），每个缓冲区长度等于传送消息长度。进程PA，PB之间的通信满足如下条件：

(1) 至少有一个空缓冲区存在时，相应的发送进程才能发送一个消息。

(2) 当缓冲队列中至少存在一个非空缓冲区时，相应的接收进程才能接收一个消息。

试描述发送过程send(i,m)和接收过程receive(i,m)。这里i代表缓冲队列。

3.12 在和控制台通信的例中，设操作员不仅仅回答用户进程所提出的问题，而且还能独立地向各用户进程发出指示。对于这些指示，操作员不要求用户进程回答，但它们享有比其他消息优先传送的优先度。即如果inbuf 中有指示存在，系统不能进行下一次通信会话。试按上述要求重新描述CCP和KCP，DCP。

3.13 编写一个程序使用系统调用fork生成3个子进程，并使用系统调用pipe创建一管道，使得这3个子进程和父进程公用同一管道进行信息通信。

3.14 设有5个哲学家，共享一张放有五把椅子的桌子，每人分得一把椅子。但是，桌子上总共只有5支筷子，在每人两边分开各放一支。哲学家们在肚子饥饿时才试图分两次从两边拾起筷子就餐。

条件：

(1) 只有拿到两支筷子时，哲学家才能吃饭。

(2) 如果筷子已在他人手上，则该哲学家必须等待到他人吃完之后才能拿到筷子。

(3) 任一哲学家在自己未拿到两支筷子吃饭之前，决不放下自己手中的筷子。

试:

(1)　描述一个保证不会出现两个邻座同时要求吃饭的通信算法。

(2)　描述一个既没有两邻座同时吃饭，又没有人饿死（永远拿不到筷子）的算法。

(3) 在什么情况下，5 个哲学家全部吃不上饭？

3.15 什么是线程？试述线程与进程的区别。

3.16 使用库函数clone()与creat-thread()在Linux环境下创建两种不同执行模式的线程程序。

课外练习题

第一类读-写者问题

一些读者和一些写者同时对一个黑板进行读写，多个读者可以同时读，但一个时刻只能有一个写者，读者写者不能同时使用黑板，对使用黑板优先级的不同规定使读-写者问题又分为几类，第一类规定读者优先级高。

某商店有两种食品A和B, 最大数量各为m个. 该商店将A,B两种食品搭配出售, 每次各取一个. 为避免食品变质, 遵循先到食品先出售的原则, 有两个食品公司分别不断地供应A,B两种食品(每次一个). 为保证正常销售, 当某种食品的数量比另一种的数量超过k(k<m)个时, 暂停对数量大的食品进货, 补充数量少的食品.

(1) 问共需设置几个进程?

(2) 试用P,V操作解决上述问题中的同步和互斥关系.

共需要3个进程，商店、食品公司A、食品公司B （2分）

semaphore  S\_BuffNum\_A;  //A的缓冲区个数, 初值m

semaphore  S\_Num\_A;          // A的个数，初值为0

semaphore  S\_BuffNum\_B;  //B的缓冲区个数, 初值m

semaphore  S\_Num\_B;          // B的个数，初值为0

void  Shop( )

{

while(1)

        {

                P(S\_Num\_A);

                P(S\_Num\_B)；

                分别取出A、B食品各一个;

                V(S\_BuffNum\_A);

                V(S\_BuffNum\_A);

                搭配地销售这一对食品;

        }

}

// “A食品加1，而B食品不变”这种情形允许出现的次数(许可证的数量)，其值等于

//k-(A-B)，初值为k

semaphore  S\_A\_B;

// “B食品加1，而A食品不变”这种情形允许出现的次数(许可证的数量)，其值等于

//k-(B-A)，初值为k

semaphore  S\_B\_A;

void  Producer\_A ( )  
{

        while(1)

        {

                生产一个A食品；

                P(S\_BuffNum\_A)；

                P(S\_A\_B);

                向商店提供一个A食品；

                V(S\_Num\_A);

                V(S\_B\_A);

        }

}

void  Producer\_B ( )

{

        while(1)

        {

                生产一个B食品；

                P(S\_BuffNum\_B)；

                P(S\_B\_A);

                向商店提供一个B食品；

                V(S\_Num\_B);

                V(S\_A\_B);

        }

}

今有三个并发进程R，M，P，它们共享了一个可循环使用的缓冲区B，缓冲区B共有N个单元。进程R负责从输入设备读信息，每读一个字符后，把它存放在缓冲区B的一个单元中；进程M负责处理读入的字符，若发现读入的字符中有空格符，则把它改成“，”；进程P负责把处理后的字符取出并打印输出。当缓冲区单元中的字符被进程P取出后，则又可用来存放下一次读入的字符。请用PV操作为同步机制写出它们能正确并发执行的程序。