



《操作系统精髓与设计原理·第五版》习题答案

第1章 计算机系统概述

1.1、图 1.3 中的理想机器还有两条 I/O 指令：

0011 = 从 I/O 中载入 AC

0111 = 把 AC 保存到 I/O 中

在这种情况下，12 位地址标识一个特殊的外部设备。请给出以下程序的执行过程（按照图1.4 的格式）：

1. 从设备 5 中载入 AC。
2. 加上存储器单元 940 的内容。
3. 把 AC 保存到设备 6 中。

假设从设备 5 中取到的下一个值为 3940 单元中的值为 2。

答案：存储器（16 进制内容）：300: 3005; 301: 5940; 302: 7006

步骤 1: 3005 → IR; 步骤 2: 3 → AC

步骤 3: 5940 → IR; 步骤 4: 3 + 2 = 5 → AC

步骤 5: 7006 → IR; 步骤 6: AC → 设备 6

1.2、本章中用 6 步来描述图 1.4 中的程序执行情况，请使用 MAR 和 MBR 扩充这个描述。

答案：1. a. PC 中包含第一条指令的地址 300，该指令的内容被送入 MAR 中。

- b. 地址为 300 的指令的内容（值为十六进制数 1940）被送入 MBR，并且 PC 增 1。这两个步骤是并行完成的。
- c. MBR 中的值被送入指令寄存器 IR 中。
2. a. 指令寄存器 IR 中的地址部分（940）被送入 MAR 中。
b. 地址 940 中的值被送入 MBR 中。
c. MBR 中的值被送入 AC 中。
3. a. PC 中的值（301）被送入 MAR 中。
b. 地址为 301 的指令的内容（值为十六进制数 5941）被送入 MBR，并且 PC 增 1。
c. MBR 中的值被送入指令寄存器 IR 中。
4. a. 指令寄存器 IR 中的地址部分（941）被送入 MAR 中。
b. 地址 941 中的值被送入 MBR 中。
c. AC 中以前的内容和地址为 941 的存储单元中的内容相加，结果保存到 AC 中。
5. a. PC 中的值（302）被送入 MAR 中。
b. 地址为 302 的指令的内容（值为十六进制数 2941）被送入 MBR，并且 PC 增 1。
c. MBR 中的值被送入指令寄存器 IR 中。
6. a. 指令寄存器 IR 中的地址部分（941）被送入 MAR 中。
b. AC 中的值被送入 MBR 中。
c. MBR 中的值被存储到地址为 941 的存储单元之中。

1.4、假设有一个微处理器产生一个 16 位的地址（例如，假设程序计数器和地址寄存器都是 16 位）并且具有一个 16 位的数据总线。

a. 如果连接到一个 16 位存储器上，处理器能够直接访问的最大存储器地址空间为多少？

b. 如果连接到一个 8 位存储器上，处理器能够直接访问的最大存储器地址空间为多少？

c. 处理访问一个独立的 I/O 空间需要哪些结构特征？

d. 如果输入指令和输出指令可以表示 8 位 I/O 端口号，这个微处理器可以支持多少 8 位 I/O 端口？

答案：对于(a)和(b)两种情况，微处理器可以直接访问的最大存储器地址空间为 $2^{16} = 64\text{K bytes}$ ，唯一的区别是 8 位存储器每次访问传输 1 个字节，而 16 位存储器每次访问可以传输一个字节或者一个 16 位的字。对于(c)情况，特殊的输入和输出指令是必要的，这些指令的执行体会产生特殊的“I/O 信号”（有别于“存储器信号”，这些信号由存储器类型指令的执行体产生）；在最小状态下，一个附加的输出针脚将用来传输新的信号。对于(d)情况，它支持 $2^8 = 256$ 个输入和 $2^8 = 256$ 个输出字节端口和相同数目的 16 位 I/O 端口；在任一情况，一个输入和一个输出端口之间的区别是通过被执行的输入输出

指令所产生的不同信号来定义的。

1.5、考虑一个32位微处理器，它有一个16位外部数据总线，并由一个8MHz的输入时钟驱动。假设这个微处理器有一个总线周期，其最大持续时间等于4个输入时钟周期。请问该微处理器可以支持的最大数据传送速度为多少？外部数据总线增加到21位，或者外部时钟频率加倍，哪种措施可以更好地提高处理器性能？请叙述你的设想并解释原因。

答案：时钟周期=1/(8MHz)=125ns

总线周期=4×125ns=500ns

每500ns传输2比特；因此传输速度=4MB/s

加倍频率可能意味着采用了新的芯片制造技术（假设每个指令都有相同的时钟周期数），加倍外部数据总线，在芯片数据总线驱动/锁存、总线控制逻辑的修改等方面手段广泛（或许更新）。在第一种方案中，内存芯片的速度要提高一倍（大约），而不能降低微处理器的速度；第二种方案中，内存的字长必须加倍，以便能发送/接受32位数量。

1.6、考虑一个计算机系统，它包含一个I/O模块，用以控制一台简单的键盘/打印机电传打字设备。CPU中包含下列寄存器，这些寄存器直接连接到系统总线上：

INPR：输入寄存器，8位

OUTR：输出寄存器，8位

FGI：输入标记，1位

FGO：输出标记，1位

IEN：中断允许，1位

I/O模块控制从打字机中输入击键，并输出到打印机中去。打字机可以把一个字母数字符号编码成一个8位字，也可以把一个8位字解码成一个字母数字符号。当8位字从打字机进入输入寄存器时，输入标记被置位；当打印一个字时，输出标记被置位。

a. 描述CPU如何使用这4个寄存器实现与打字机间的输入输出。

b. 描述通过使用IEN，如何提高执行效率？

答案：a. 来源于打字机的输入储存在INPR中。只有当FGI=0时，INPR才会接收来自打字机的数据。当数据接收后，被储存在INPR里面，同时FGI置为1。CPU定期检查FGI。如果FGI=1，CPU将把INPR里面的内容传送至AC，并把FGI置为0。

当CPU需要传送数据到打字机时，它会检查FGO。如果FGO=0，CPU处于等待。如果FGO=1，CPU将把AC的内容传送至OUTR并把FGO置为0。当数字符号打印后，打字机将把FGI置为1。

b. (A) 描述的过程非常浪费。速度远高于打字机的CPU必须反复不断的检查FGI和FGO。如果中断被使用，当打字机准备接收或者发送数据时，可以向CPU发出一个中断请求。IEN计数器可以由CPU设置（在程序员的控制下）。

1.7、实际上在所有包括DMA模块的系统中，DMA访问主存储器的优先级总是高于处理器访问主存储器的优先级。这是为什么？

答案：如果一个处理器在尝试着读或者写存储器时被挂起，通常除了一点轻微的时间损耗之外没有任何危害。但是，DMA可能从或者向设备（例如磁盘或磁带）以数据流的方式接收或者传输数据并且这是不能被打断的。否则，如果DMA设备被挂起（拒绝继续访问主存），数据可能会丢失。

1.9、一台计算机包括一个CPU和一台I/O设备D，通过一条共享总线连接到主存储器M，数据总线的宽度为1个字。CPU每秒最多可执行 10^6 条指令，平均每条指令需要5个机器周期，其中3个周期需要使用存储器总线。存储器读/写操作使用1个机器周期。假设CPU正在连续不断地执行后台程序，并且需要保证95%的指令执行速度，但没有任何I/O指令。假设1个处理器周期等于1个总线周期，现在要在M和D之间传送大块数据。

a. 若使用程序控制I/O，I/O每传送1个字需要CPU执行两条指令。请估计通过D的I/O数据传送的最大可能速度。

b. 如果使用DMA传送，请估计传送速度。

答案：a. 处理器只能分配 5% 的时间给 I/O，所以最大的 I/O 指令传送速度是 $10e6 \times 0.05 = 50000$ 条指令/秒。
因此 I/O 的传送速率是 25000 字/秒。

b. 使用 DMA 控制时，可用的机器周期下的数量是

$$10e6 (0.05 \times 5 + 0.95 \times 2) = 2.15 \times 10e6$$

如果我们假设 DMA 模块可以使用所有这些周期，并且忽略任何设置和状态检查时间，那么这个值就是最大的 I/O 传输速率。

1.10、考虑以下代码：

```
for (i = 0; i < 20; i++)
    for (j = 0; j < 10; j++)
        a[i] = a[i]*j
```

a. 请举例说明代码中的空间局部性。

b. 请举例说明代码中的时间局部性。

答案：a. 读取第二条指令是紧跟着读取第一条指令的。

b. 在很短的间歇时间内， $a[i]$ 在循环内部被访问了十次。

1.11、请将附录 1A 中的式 (1.1) 和式 (1.2) 推广到 n 级存储器层次中。

答案：定义：

C_i = 存储器层次 i 上每一位的存储单元平均花销

S_i = 存储器层次 i 的规模大小

T_i = 存储器层次 i 上访问一个字所需时间

H_i = 一个字在不高于层次 i 的存储器上的概率

B_i = 把一个数据块从层次 $i+1$ 的存储器上传输到层次 i 的存储器上所需时间

高速缓冲存储器作为是存储器层次；主存为存储器层次 2；针对所有的 N 层存储器层以此类推。有：

$$C_s = \frac{\sum_{i=1}^n C_i S_i}{\sum_{i=1}^n S_i}$$

T_s 的引用更复杂，我们从概率论入手：所期望的值 $x = \sum_{i=1}^n i \Pr[x = 1]$ ，由此我们可以写出：

$$T_s = \sum_{i=1}^n T_i H_i$$

我们需要清楚如果一个字在 M1（缓存）中，那么对它的读取非常快。如果这个字在 M2 而不在 M1 中，那么数据块需要从 M2 传输到 M1 中，然后才能读取。因此， $T_2 = B_1 + T_1$

进一步， $T_3 = B_2 + T_2 = B_1 + B_2 + T_1$

$$\text{以此类推: } T_i = \sum_{j=1}^{i-1} B_j + T_1$$

$$\text{所以, } T_s = \sum_{i=2}^n \sum_{j=1}^{i-1} (B_j H_i) + T_1 \sum_{i=1}^n H_i$$

$$\text{但是, } \sum_{i=1}^n H_i = 1$$

$$\text{最后, } T_s = \sum_{i=2}^n \sum_{j=1}^{i-1} (B_i H_i) + T_1$$

1.12、考虑一个存储器系统，它具有以下参数：

$$\begin{array}{ll} T_c = 100 \text{ ns} & C_c = 0.01 \text{ 分/位} \\ T_m = 1200 \text{ ns} & C_m = 0.001 \text{ 分/位} \end{array}$$

a. 1MB 的主存储器价格为多少？

b. 使用高速缓冲存储器技术，1MB 的主存储器价格为多少？

c. 如果有效存取时间比高速缓冲存储器存取时间多10%，命中率H为多少？

答案：a. 价格 = $C_m \times 8 \times 10^6 = 8 \times 10^3 \text{ } \mathcal{C} = \80

b. 价格 = $C_c \times 8 \times 10^6 = 8 \times 10^4 \text{ } \mathcal{C} = \800

c. 由等式 1.1 知： $1.1 \times T_1 = T_1 + (1-H)T_2$

$$(0.1)(100) = (1-H)(1200)$$

$$H = 1190/1200$$

1.13、一台计算机包括包括高速缓冲存储器、主存储器和一个用做虚拟存储器的磁盘。如果要存取的字在高速缓冲存储器中，存取它需要20ns；如果该字在主存储器中而不在高速缓冲存储器中，把它载入高速缓冲存储器需要 60ns（包括最初检查高速缓冲存储器的时间），然后再重新开始存取；如果该字不在主存储器中，从磁盘中取到内存需要 12ms，接着复制到高速缓冲存储器中还需要 60ns，再重新开始存取。高速缓冲存储器的命中率为0.9，主存储器的命中率为0.6，则该系统中存取一个字的平均存取时间是多少（单位为 ns）？

答案：有三种情况需要考虑：

字所在的位置	概率	访问所需时间 (ns)
在缓存中	0.9	20
不在缓存，在主存中	$(0.1)(0.6) = 0.06$	$60 + 20 = 80$
不在缓存也不在主存中	$(0.1)(0.4) = 0.04$	$12\text{ms} + 60 + 20 = 12,000,080$

所以平均访问时间是： $\text{Avg} = (0.9)(20) + (0.06)(80) + (0.04)(12000080) = 480026 \text{ ns}$

1.14、假设处理器使用一个栈来管理过程调用和返回。请问可以取消程序计数器而用栈指针代替吗？

答案：如果栈只用于保存返回地址。或者如果栈也用于传递参数，这种方案只有当栈作为传递参数的控制单元而非机器指令时才成立。这两种情况下可以取消程序计数器而用栈指针代替。在后者情况中，处理器同时需要一个参数和指向栈顶部的程序计数器。

第 2 章 操作系统概述

2.1 假设我们有一台多道程序的计算机，每个作业有相同的特征。在一个计算周期T 中，一个作业有一半时间花费在 I/O 上，另一半用于处理器的活动。每个作业一共运行N 个周期。假设使用简单的循环法调度，并且 I/O 操作可以与处理器操作重叠。定义以下量：

- 时间周期=完成任务的实际时间
- 吞吐量=每个时间周期 T 内平均完成的作业数目
- 处理器使用率=处理器活跃（不是处于等待）的时间的百分比

当周期 T 分别按下列方式分布时，对1 个、2 个和 4 个同时发生的作业，请计算这些量：

- 前一般用于 I/O，后一半用于处理器。
- 前四分之一和后四分之一用于I/O，中间部分用于处理器。

答：(a) 和 (b) 的答案相同。尽管处理器活动不能重叠，但I/O 操作能。

一个作业 时间周期=NT 处理器利用率=50 %

两个作业 时间周期=NT 处理器利用率=100 %

四个作业 时间周期= (2N-1) NT 处理器利用率=100 %

2.2 I/O 限制的程序是指如果单独运行，则花费在等待I/O 上的时间比使用处理器的时间要多的程序。处理器限制的程序则相反。假设短期调度算法偏爱那些在近期石油处理器时间较少的算法，请解释为什么这个算法偏爱 I/O 限制的程序，但是并不是永远不受理处理器限制程序所需的处理器时间？

受 I/O 限制的程序使用相对较少的处理器时间，因此更受算法的青睐。然而，受处理器限制的进程如果在足够长的时间内得不到处理器时间，同一算法将允许处理器去处理此进程，因为它最近没有使用过处理器。这样，一个处理器限制的进程不会永远得不到处理器。

2.3 请对优化分时系统的调度策略和用于优化多道程序批处理系统的调度策略进行比较。

分时系统关注的是轮转时间，时间限制策略更有效是因为它给所有进程一个较短的处理时间。批处理系统关心的是吞吐量，更少的上下文转换和更多的进程处理时间。因此，最小的上下文转换最高效。

2.4 系统调用的目的是什么？如何实现与操作系统相关的系统调用以及与双重模式（内核模式和用户模式）操作相关的系统调用？

系统调用被应用程序用来调用一个由操作系统提供的函数。通常情况下，系统调用最终转换成在内核模式下的系统程序。

2.5 在 IBM 的主机操作系统 OS/390 中，内核中的一个重要模块是系统资源管理程序 (System Resource Manager, SRM)，它负责地址空间（进程）之间的资源分配。SRM 是 OS/390 在操作系统中具有特殊性，没有任何其他的主机操作系统，当然没有任何其他类型的操作系统可以比得上 SRM 所实现的功能。资源的概念包括处理器、实存和 I/O 通道，SRM 累计处理器、I/O 通道和各种重要数据结构的利用率，它的目标是基于性能监视和分析提供最优的性能，其安装设置了以后的各种性能目标作为 SRM 的指南，这会基于系统的利用率动态的修改安装和作业性能特点。SRM 依次提供报告，允许受过训练的操作员改进配置和参数设置，以改善用户服务。

现在关注 SRM 活动的一个实例。实存被划分为成千上万个大小相等的块，称为帧。每个帧可以保留一块称为页的虚存。SRM 每秒大约接受 20 次控制，并在互相之间以及每个页面之间进行检查。如果页未被引用或被改变，计数器增 1。一段时间后，SRM 求这些数据的平均值，以确定系统中一个页面未曾被触及的平均秒数。这样做的目的是什么？SRM 将采取什么动作？

操作系统可以查看这些数据已确定系统的负荷，通过减少加在系统上的活跃作业来保持较高的平均利用率。典型的平均时间应该是两分钟以上，这个平均时间看起来很长，其实并不长。

第 3 章 进程描述和控制

3.1. 给出操作系统进行进程管理时的五种主要活动，并简单描述为什么需要它们。

答：用户进程和系统进程创建及删除。系统中的进程可以为信息共享、运算加速、模块化和方便并发地执行。而并发执行需要进程的创建和删除机制。当进程创建或者运行时分配给它需要的资源。当进程终止时，操作系统需要收回任何可以重新利用的资源。

进程的暂停和继续执行。在进程调度中，当进程在等待某些资源时，操作系统需要将它的状态改变为等待或就绪状态。当所需要的资源可用时，操作系统需要将它的状态变为运行态以使其继续执行。提供进程的同步机制。合作的进程可能需要共享数据。对共享数据的并行访问可能会导致数据冲突。操作系统必须提供进程的同步机制以使合作进程有序地执行，从而保证数据的一致性。

提供进程的通信机制。操作系统下执行的进程既可以是独立进程也可以是合作进程。合作进程之间必须具有一定的方式进行通信。

提供进程的死锁解决机制。在多道程序环境中，多个进程可能会竞争有限的资源。如果发生死锁，所有的等待进程都将永远不能由等待状态再变为运行态，资源将被浪费，工作永远不能完成。

3.2. 在[PINK89] 中为进程定义了以下状态：执行（运行）态、活跃（就绪）态、阻塞态和挂起态。当进程正在等待允许使用某一资源时，它处于阻塞态；当进程正在等待它已经获得的某种资源上的操作完成时，它处于挂起态。在许多操作系统中，这两种状态常常放在一起作为阻塞态，挂起态使用本章中给出的定义。请比较这两组定义的优点。

答：[PINK89] 中引用了以下例子来阐述其中阻塞和挂起的定义：

假设一个进程已经执行了一段时间，它需要一个额外的磁带设备来写出一个临时文件。在它开

始写磁带之前，进程必须得到使用某一设备的许可。当它做出请求时，磁带设备可能并不可用，这种情况下，该进程就处于阻塞态。假设操作系统在某一时刻将磁带设备分配给了该进程，这时进程就重新变为活跃态。当进程重新变为执行态时要对新获得的磁带设备进行写操作。这时进程变为挂起态，等待该磁带上当前所进行的写操作完成。

这种对等待某一设备的两种不同原因的区别，在操作系统组织其工作时是非常有用的。然而这并不能表明那些进程是换入的，那些进程是换出的。后一种区别是必需的，而且应该在进程状态中以某种形式表现出来。

- 3.3.** 对于图 3.9 (b) 中给出的 7 状态进程模型，请仿照图 3.8 (b) 画出它的排队图。

答：图 9.3 给出了单个阻塞队列的结果。该图可以很容易的推广到多个阻塞队列的情形。

- 3.4.** 考虑图 3.9 (b) 中的状态转换图。假设操作系统正在分派进程，有进程处于就绪态和就绪挂起态，并且至少有一个处于就绪/挂起态的进程比处于就绪态的所有进程的优先级都高。有两种极端的策略：

(1) 总是分派一个处于就绪态的进程，以减少交换；(2) 总是把机会给具有最高优先级的进程，即使会导致在不需要交换时进行交换。请给出一种能均衡考虑优先级和性能的中间策略。

答：对于一个就绪/挂起态的进程，降低一定数量（如一或两个）优先级，从而保证只有当一个就绪/挂起态的进程比就绪态的进程的最高优先级还高出几个优先级时，它才会被选做下一个执行。

- 3.5.** 表 3.13 给出了 VAX/VMS 操作系统的进程状态。

a. 请给出这么多种等待状态的理由。

b. 为什么以下状态没有驻留和换出方案：页错误等待、也冲突等待、公共事件等待、自由页等待和资源等待。

c. 请画出状态转换图，并指出引发状态转换的原因。

答：

- a. 每一种等待状态都有一个单独的队列与其相关联。当影响某一等待进程的事件发生时，把等待进程分成不同的队列就减少了定位这一等待进程所需的工作量。例如，当一个页错误完成时，调度程序就可以在页错误等待队列中找到等待的进程。
- b. 在这些状态下，允许进程被换出只会使效率更低。例如，当发生页错误等待时，进程正在等待换入一个页从而使其可以执行，这是将进程换出是毫无意义的。
- c. 可以由下面的进程状态转换表得到状态转换图。

当前状态		下一状态			
	当前正在执行	可计算（驻留）	可计算（换出）	各种等待状态（驻留）	各种等待状态（换出）
当前正在执行		重调度		等待	
可计算（驻留）	调度		换出		
可计算（换出）		换入			
各种等待状态（驻留）		事件发生			换出
各种等待状态（换出）			事件发生		

- 3.6.** VAX/VMS 操作系统采用了四种处理器访问模式，以促进系统资源在进程间的保护和共享。访问模式确定：

- 指令执行特权：处理器将执行什么指令。
- 内存访问特权：当前指令可能访问虚拟内存中的哪个单元。

四种模式如下：

- 内核模式：执行 VMS 操作系统的内核，包括内存管理、中断处理和I/O 操作。
- 执行模式：执行许多操作系统服务调用，包括文件（磁盘和磁带）和记录管理例程。
- 管理模式：执行其他操作系统服务，如响应用户命令。
- 用户模式：执行用户程序和诸如编译器、编辑器、链接程序、调试器之类的实用程序。

在较少特权模式执行的进程通常需要调用在较多特权模式下执行的过程，例如，一个用户程序需要一个操作系统服务。这个调用通过使用一个改变模式（简称CHM）指令来实现，该指令将引发一个中断，把控制转交给处于新的访问模式下的例程，并通过执行 REI(Return from Exception or Interrupt, 从异常或中断返回) 指令返回。

- 很多操作系统有两种模式，内核和用户，那么提供四种模式有什么优点和缺点？
- 你可以举出一种有四种以上模式的情况吗？

答：

- 四种模式的优点是对主存的访问控制更加灵活，能够为主存提供更好的保护。缺点是复杂和处理的开销过大。例如，程序在每一种执行模式下都要有一个独立的堆栈。
- 原则上，模式越多越灵活，但是四种以上的模式似乎很难实现。

3.7. 在前面习题中讨论的 VMS 方案常常称为环状保护结构，如图3.18 所示。3.3 节所描述的简单的内核/用户方案是一种两环结构，[SILB04]指出了这种方法的问题：环状（层次）结构的主要缺点是它不允许我们实施须知原理，特别地，如果一个对象必须在域 D_j 中可访问，但在域 D_i 中不可访问，则必须有就 $j > i$ 。这意味着在 D_i 中可访问的每个段在 D_j 中都可以访问。

- 请清楚地解释上面引文中提出的问题。
- 请给出环状结构操作系统解决这个问题的一种方法。

答：

- 当 $j > i$ 时，运行在 D_i 中的进程被禁止访问 D_j 中的对象。因此，如果 D_j 中包含的信息比 D_i 中的更具有特权或者要求的安全性更高，那么这种限制就是合理的。然而，通过以下方法却可以绕过这种安全策略。一个运行在 D_j 中的进程可以读取 D_j 中的数据，然后把数据复制到 D_i 中。随后， D_i 中的进程就可以访问这些信息了。
- 有一种解决这一问题的方法叫做可信系统，我们将在16 章中进行讨论。

3.8. 图 3.7 (b) 表明一个进程每次只能在一个事件队列中。

- 是否能够允许进程同时等待一个或多个事件？请举例说明。
- 在这种情况下，如何修改图中的排队结构以支持这个新特点？

答：

- 一个进程可能正在处理从另一个进程收到的数据并将结果保存到磁盘上。如果当前在另一个进程中正有数据在等待被取走，进程就可以继续获得数据并处理它。如果前一个写磁盘操作已经完成，并且有处理好的数据在等待写出，那么进程就可以继续写磁盘。这样就可能存在某一时刻，进程即在等待从输入进程获得数据，又在等待磁盘可用。
- 有很多种方法解决这一问题。可以使用一种特殊的队列，或者将进程放入两个独立的队列中。不论采用哪种方法，操作系统都必须处理好细节工作，使进程相继地关注两个事件的发生。

3.9. 在很多早期计算机中，中断导致寄存器值被保存在与给定的中断信息相关联的固定单元。在什么情况下这是一种实用的技术？请解释为什么它通常是不方便的。

答：这种技术是基于被中断的进程A 在中断响应之后继续执行的假设的。但是，在通常情况下，中断可能会导致另一个进程B 抢占了进程A。这是就必须将进程A 的执行状态从与中断相关的位置复制到与 A 相关的进程描述中。然而机器却有可能仍将它们保存到前一位置。参考:[BRIN73]。

3.10. 3.4 节曾经讲述过，由于在内核模式下执行的进程是不能被抢占的，因此UNIX 不适用于实时应用。请阐述原因。

答：由于存在进程不能被抢占的情况（如在内核模式下执行的进程），操作系统不可能对实时需求给予迅速的反应。

第4章 线程、对称多处理和微内核

- 4.1. 一个进程中的多个线程有以下两个优点：(1) 在一个已有进程中创建一个新线程比创建一个新进程所需的工作量少；(2) 在同一个进程中的线程间的通信比较简单。请问同一个进程中的两个线程间的模式切换与不同进程中的两个线程间的模式切换相比，所需的工作量是否要少？
答：是的，因为两个进程间的模式切换要储存更多的状态信息。
- 4.2. 在比较用户级线程和内核级线程时曾指出用户级线程的一个缺点，即当一个用户级线程执行系统调用时，不仅这个线程被阻塞，而且进程中的所有线程都被阻塞。请问这是为什么？
答：因为对于用户级线程来说，一个进程的线程结构对操作系统是不可见的，而操作系统的调度是以进程为单位的。
- 4.3. 在OS/2中，其他操作系统中通用的进程概念被分成了三个独立类型的实体：会话、进程和线程。一个会话是一组与用户接口（键盘、显示器、鼠标）相关联的一个或多个进程。会话代表了一个交互式的用户应用程序，如字处理程序或电子表格，这个概念使得PC用户可以打开一个以上的应用程序，在屏幕上显示一个或更多个窗口。操作系统必须知道哪个窗口，即哪个会话是活跃的，从而把键盘和鼠标的输入传递给相应的会话。在任何时刻，只有一个会话在前台模式，其他的会话都在后台模式，键盘和鼠标的所有输入都发送给前台会话的一个进程。当一个会话在前台模式时，执行视频输出的进程直接把它发送到硬件视频缓冲区。当一个会话在后台时，如果该会话的任何一个进程的任何一个线程正在执行并产生屏幕输出，则这个输出被送到逻辑视频缓冲区；当这个会话返回前台时，屏幕被更新，为新的前台会话反映出逻辑视频缓冲区中的当前内容。
有一种方法可以把OS/2中与进程相关的概念的数目从3个减少到2个。删去会话，把用户接口（键盘、显示器、鼠标）和进程关联起来。这样，在某一时刻，只有一个进程处于前台模式。为了进一步地进行构造，进程可以被划分成线程。
a. 使用这种方法会丧失什么优点？
b. 如果继续使用这种修改方法，应该在哪里分配资源（存储器、文件等）在进程级还是线程级？
答：
a. 会话的使用非常适合个人计算机和工作站对交互式图形接口的需求。它为明确图形输出和键盘鼠标输入应该被关联到什么位置提供了一个统一的机制，减轻了操作系统的工作负担。
b. 应该和其他的进程/线程系统一样，在进程级分配地址空间和文件。
- 4.4. 考虑这样一个环境，用户级线程和内核级线程呈一对一的映射关系，并且允许进程中的一个或多个线程产生会引发阻塞的系统调用，而其他线程可以继续运行。解释为什么这个模型可以使多线程程序比在单处理器机器上的相应的单线程程序运行速度更快？
答：问题在于机器会花费相当多的时间等待I/O操作的完成。在一个多线程程序中，可能一个内核级线程会产生引发阻塞的系统调用，而其他内核级线程可以继续执行。而在单处理器机器上，进程则必须阻塞知道所有的系统调用都可以继续运行。参考：[LEWI96]
- 4.5. 如果一个进程退出时，该进程的某些线程仍在运行，请问他们会继续运行吗？
答：不会。当一个进程退出时，会带走它的所有东西——内核级线程，进程结构，存储空间——包括线程。参考：[LEWI96]
- 4.6. OS/390主机操作系统围绕着地址空间和任务的概念构造。粗略说来，一个地址空间对应于一个应用程序，并且或多或少地对应于其他操作系统中的一个进程；在一个地址空间中，可以产生一组任务，并且它们可以并发执行，这大致对应于多线程的概念。管理任务结构有两个主要的数据结构。地址空间控制块（ASCB）含有OS/390所需要的关于一个地址空间的信息，而不论该地址空间是否在执行。ASCB中的信息包括分派优先级、分配给该地址空间的实存和虚存、该地址空间中就绪的任务数以及是否每个都被换出。一个任务控制块（TCB）标识一个正在执行的用户程序，它含有在一个地址空间中管理该任务所需要的信息，包括处理器状态信息、指向该任务所涉及到的程序的指针和任务执行结构。ASCB是在系统存储器中保存的全局结构，而TCB是保存在各自的地址空间中的局部结构。请问把控制信息划分成全局和局部两部分有什么好处？

答：关于一个地址空间的尽可能多的信息可以随地址空间被换出，从而节约了主存。

- 4.7. 一个多处理系统有8个处理器和20个附加磁带设备。现在有大量的作业提交给该系统，完成每个作业最多需要4个磁带设备。假设每个作业开始运行时只需要3个磁带设备，并且在很长时间内都只需要这3个设备，而只是在最后很短的一段时间内需要第4个设备以完成操作。同时还假设这类作业源源不断。

- a. 假设操作系统中的调度器只有当4个磁带设备都可用时才开始一个作业。当作业开始时，4个设备立即被分配给它，并且直到作业完成时才被释放。请问一次最多可以同时执行几个作业？采用这种策略，最多有几个磁带设备可能是空闲的？最少有几个？
- b. 给出另外一种策略，要求其可以提高磁带设备的利用率，并且同时可以避免系统死锁。分析最多可以有几个作业同时执行，可能出现的空闲设备的范围是多少。

答：

- a. 采用一个保守的策略，一次最多同时执行 $20/4=5$ 个作业。由于分配各一个任务的磁带设备最多同时只有一个空闲，所以在同一时刻最多有5个磁带设备可能是空闲的。在最好的情况下没有磁带设备空闲。
- b. 为了更好的利用磁设备，每个作业在最初只分配三个磁带设备。第四个只有的需要的时候才分配。在这种策略中，最多可以有 $20/3=6$ 个作业同时执行。最少的空闲设备数量为0，最多有2个。参考：Advanced Computer Architectrue,K.Hwang,1993.

- 4.8. 在描述 Solaris 用户级线程状态时，曾表明一个用户级线程可能让位于具有相同优先级的另一个线程。请问，如果有一个可运行的、具有更高优先级的线程，让位函数是否还会导致让位于具有相同优先级或更高优先级的线程？

答：任何一个可能改变线程优先级或者使更高优先级的线程可运行的调用都会引起调度，它会依次抢占低优先级的活跃线程。所以，永远都不会存在一个可运行的、具有更高优先级的线程。参考：[LEVI96]

第5章 并发性：互斥和同步

5.1

答：b. 协同程序 read 读卡片，将字符赋给一个只有一个字大小的缓冲区rs 然后在赋给 squash 协同程。协同程序 Read 在每副卡片图像的后面插入一个额外的空白。协同程序 squash 不需要知道任何关于输入的八十个字符的结构，它简单的查找成对出现的星号，然后将更改够的字符串经由只有一个字符大小的缓冲sp，传递给协同程序 print。最后协同程序 print 简单的接受到来的字符串，并将他们打印在包含125个字符的行中。

- 5.2. 考虑一个并发程序，它有两个进程p 和 q，定义如下。A.B.C.D 和 E 是任意的原子语句。假设住程序执行两个进程的 parbegin

```
Void p()
{
    A;
    B;
    C;
}

void q()
{
    D;
    E;
}
```

答：ABCDE;ABDCE;ABDEC;ADBCE;ADBEC;ADEBC;DEABC;DAEBC;DABEC;DABCE;

5.3 考虑下面的程序

```
const int n=50;
int tally;
void total()
{
    int count;
    for(count=1;count <=n;count ++)
```

```

    {tally++;
}
}

void main()
{
    tally =0;
    parbegin(total(),total());
    write(tally);
}

```

答：a.随意一看,tally 值的范围好像是落在[50,100]这个区间里,因为当没有互斥时可以从 0 直接增加到 50. 这一基本论点是当并发的运行这两进程时,我们不可能得到一个比连续执行单一某进程所得 tally 值还低的一个最终 tally 值.但是考虑下面由这两进程按交替顺序执行载入增加,存储的情况,同时变更这个共享变量的取值:

1.进程 A 载入 tally 值,tally 值加到 1,在此时失去处理器(它已经增加寄存器的值到1,但是还没有存储这个值).

2.进程 B 载入 tally 值(仍然是 0)然后运行完成 49 次增加操作,在它已经将 49 这个值存储给共享变量 tally 后,失去处理器控制权.

3.进程 A 重新获得处理器控制权去完成它的第一次存储操作用 1 去代替先前的 49 这个 tally 值),此时被迫立即放弃处理器.

4.进程 B 重新开始,将 1(当前的 tally 值)载入到它自己的寄存器中,但此时被迫放弃处理器(注意这是 B 的最后一次载入).

5.进程 A 被重新安排开始,但这次没有被中断,直到运行完成它剩余的49次载入,增加和存储操作,结果是此时 tally 值已经是 50.

6.进程 B 在它终止前完成仅有的最后一次增加和存储操作它的寄存器值增至 2,同时存储这个值做为这个共享变量的最终结果.

一些认为会出现低于2 这个值的结果,这种情况不会出现这样 tally 值的正确范围是[2,100].

b.对一般有 N 个进程的情况下,tally 值的最终范围是[2,N*50]因为对其他所有进程来说,从最初开始运行到在第五步完成但最后都被进程B 破坏掉它们的最终结果

5.4.忙等待是否总是比阻塞等待效率低（根据处理器的使用时间）？请解释。

答：就一般情况来说是对的因为忙等待消耗无用的指令周期然而,有一种特殊情况,当进程执行到程序的某一点处,在此处要等待直到条件满足,而正好条件已满足,此时忙等待会立即有结果,然而阻塞等待会消耗操作系统资源在换出与换入进程上

5.5 考虑下面的程序

```

boolean blocked[2];
int turn;
void P(int id)
{
    While (true)
    {
        While(turn!=id);
        {
            While(blocked[1-id]
                  /*do nothing*/;
            Turn=id;
        }
    }
}

```

```

Void main()
{
    Blocked[0]=false;
    Blocked[1]=false;
    Turn=0;
    Parbegin(P(0),P(1));
}

```

这是【HYMA66】中提出的解决互斥问题的一种方法。请举出证明该方法不正确的一个反例。

答：考虑这种情况：此时 turn=0，进程 P(1) 使布尔变量 blocked[1] 的值为 true，在这时发现布尔变量 blocked[0] 的值为 false，然后 P(0) 会将 true 值赋予 blocked[0]

，此时 turn=0，P(0) 进入临界区，P(1) 在将 1 赋值给 turn 后，也进入了临界区。

5.6 解决互斥的另一种软件方法是 lamport 的面包店（bakery）算法，之所以起这个名字，是因为它的思想来自于面包店或其他商店中，每个顾客在到达时都得到一个有编号的票，并按票号依次得到服务，算法如下：

```

Boolean choosing[n];
Int number[n];
While (true)
{
    Choosing[i]=true;
    Number[i]=1+getmax(number[],n) ;
    Choosing[i]=false;
    For(int j=0;j
    {
        While (choosing[j])
        {}
        While ((number[j]!=0)&&(number[j],j)<(number[i],i)
        {}
    }
    /*critical section*/
    Number[i]=0;
    /*remainder*/;
}

```

数组 choosing 和 number 分别被初始化成 false 和 0，每个数组的第 i 个元素可以由进程 i 读或写，但其他进程只能读。符号 (a, b) < (c, d) 被定义成

(a, c) 或 (a=c 且 b)

- A. 用文字描述这个算法。
- B. 说明这个算法避免了死锁。
- C. 说明它实施了互斥。

答：a. 当一个进程希望进入临界区时，它被分配一个票号。分配的票号是通过在目前那些等待进入临界区的进程所持票号和已经在临界区的进程所持票号比较所得最大票号再加 1 得到的。有最小票号的进程有最高的优先级进入临界区。当有多个进程拥有同样的票号时，拥有最小数字号进入临界区。当一个进程退出临界区时，重新设置它的票号为 0。

b. 如果每个进程被分配唯一的一个进程号，那么总会有一个唯一的、严格的进程顺序。因此，死锁可以避免。

c. 为了说明互斥，我们首先需要证明下面的定理：如果 P_i 在它的临界区， P_k 已经计算出来它的 $number[k]$ ，并试图进入临界区，此时就有下面的关系式： $(number[i], i) < (number[k], k)$ 。为证明定理，定义下

面一些时间量:

Tw1:Pi 最后一次读 choosing[k], 当 $j=k$, 在它的第一次等待时, 因此我们在 Tw1 处有 choosing[k] = false.

Tw2:Pi 开始它的最后执行, 当 $j=k$, 在它的第二次 while 循环时, 因此我们有 Tw1 < Tw2.

Tk1:Pk 在开始 repeat 循环时; Tk2:Pk 完成 number[k] 的计算;

Tk3:Pk 设置 choosing[k] 为 false 时. 我们有 Tk1

因为在 Tw1 处, choosing[k]=false 我们要么有 Tw1 要么有 Tk3 在第一种情况下, 我们有 number[i] 因为 Pi 在 Pk 之前被分配号码, 这个满足定理条件. 在第二种情况下, 我们有 Tk2 < Tk3 < Tw1 < Tw2 因此有 Tk2 这意味着在 Tw2 时, Pi 已经读了当前 number[k] 的值. 而且, 因为 Tw2 是当 $j=k$ 第二次 while 循环执行发生的时刻, 我们有 $(number[i], i) < (number[k], k)$, 这样完成了定理的证明. 现在就很容易说明实施了互斥. 假定 Pi 在临界区, Pk 正试图进入临界区. Pk 将不能进入临界区, 因为它会发现 number[i] 不等于 0, 并且 $(number[i], i) < (number[k], k)$.

5.7 当按图 5.2 的形式使用一个专门机器指令提供互斥时, 对进程在允许访问临界区之前必须等待多久没有控制. 设计一个使用 testset 指令的算法, 且保证任何一个等待进入临界区的进程在 $n-1$ 个 turn 内进入, n 是要求访问临界区的进程数, turn 是指一个进程离开临界区而另一个进程获准访问这个一个事件。

答: 以下的程序由 [SILB98] 提供:

```
var j: 0..n-1;
key: boolean;
repeat
    waiting[i] := true;
    key := true;
    while waiting[i] and key do key := testset(lock);
    waiting[i] := false;
    < critical section >
    j := i + 1 mod n;
    while (j ≠ i) and (not waiting[j]) do j := j + 1 mod n;
    if j = i then lock := false
    else waiting := false;
    < remainder section >
Until
```

这个算法用最普通的数据结构: var waiting: array [0..n - 1] of boolean

Lock: boolean

这些数据结构被初始化成假的, 当一个进程离开它的临界区, 它就搜索 waiting 的循环队列

5.8 考虑下面关于信号量的定义:

```
Void semWait(s)
{
    If (s.count>0)
    {
        s.count--;
    }
    Else
    {
        Place this process in s.queue;
        Block;
    }
}
```

```

Void semSignal(s)
{
    If (there is at least one process blocked on semaphore)
    {
        Remove a process P from s.queue;
        Place process P on ready list;
    }
    Else
        s.count++;
}

```

比较这个定义和图 5.3 中的定义, 注意有这样一个区别: 在前面的定义中, 信号量永远不会取负值。当在程序中分别使用这两种定义时, 其效果有什么不同? 也就是说, 是否可以在不改变程序意义的前提下, 用一个定义代替另一个?

答: 这两个定义是等价的, 在图 5.3 的定义中, 当信号量的值为负值时, 它的值代表了有多少个进程在等待; 在此题中的定义中, 虽然你没有关于这方面的信息, 但是这两个版本的函数是一样的。

5.9 可以用二元信号量实现一般信号量。我们使用 **semWaitB** 操作和 **semSignalB** 操作以及两个二元信号量 **delay** 和 **mutex**。考虑下面的代码

```

Void semWait(semaphore s)
{
    semWaitB(mutex);
    s--;
    if (s<0)
    {
        semSignalB(mutex);
        semWaitB(delay);
    }
    Else
        Semsignalb(mutex)
}

Void semSignal(semaphore s);
{
    semWaitB(mutex);
    s++;
    if(s<=0)
        semSignalB(delay);
    semSignalB(mutex);
}

```

最初。S 被设置成期待的信号量值, 每个 **semwait** 操作将信号量减 1, 每个 **sem signal** 操作将信号量加 1. 二元信号量 **mutex** 被初始化成 1, 确保在更新在更新 s 时保证互斥, 二元信号量 **delay** 被初始化成 0, 用于挂起进程,

上面的程序有一个缺点, 证明这个缺点, 并提出解决方案。提示: 假设两个进程, 每个都在初始化为 0 时调用 **semwait (s)**, 当第一个刚刚执行了 **sem signalb (mutex)** 但还没有执行 **sem waitb (delay)**, 第二个调用 **semwait (s)** 并到达同一点。现在需要做的就是移动程序的一行

答: 假设两个进程, 每个都在 s 被初始化成 0 时调用 **semWait (s)**, 当第一个刚执行了 **semSignalB (mutex)** 但还没有执行 **semWaitB (delay)** 时, 第二个调用 **semWait (s)** 并到达同一点。因为 s=-2 mutex 没有锁定, 假如有另外两个进程同时成功的调用 **semSignal (s)**, 他们接着就会调用 **semSignalb (delay)**,

但是第二个 `semSignalB` 没有被定义。

解决方法就是移动 `semWait` 程序中 `end` 前的 `else` 一行到 `semSignal` 程序中最后一行之前。因此 `semWait` 中的最后一个 `semSignalB(mutex)` 变成无条件的，`semSignal` 中的 `semSignalB(mutex)` 变成了有条件的。

5.10 1978 年，dijkstra 提出了一个推测，即使用有限数目的弱信号量，没有一种解决互斥的方案，使用于数目未知但有限的进程且可以避免饥饿。1979 年，j.m.morris 提出 了一个使用三个弱信号量的算法，反驳了这个推测。算法的行为可描述如下，如果一个或多个进程正在 `semwait(s)` 操作上等待，另一个进程正在执行 `semSignal(s)`，则信号量 `s` 的值未被修改，一个等待进程被解除阻塞，并且这并不取决于 `semwait(s)`。除了这三个信号量外，算法使用两个非负整数变量，作为在算法特定区域的进程的计数器。因此，信号量 **A** 和 **B** 被初始化为 1，而信号量 **M** 和计数器 **NA**, **NM** 被初始化成 0.一个试图进入临界区的进程必须通过两个分别由信号量 **A** 和 **M** 表示路障，计数器 **NA** 和 **NM** 分别含有准备通过路障 **A** 以及通过路障 **A** 但还没有通过路障 **M** 的进程数。在协议的第二部分，在 **M** 上阻塞的 **NM** 个进程将使用类似于第一部分的串联技术，依次进入他们的临界区，定义一个算法实现上面的描述。

答：这个程序由[RAYN86]提供：

```
var a, b, m: semaphore;
na, nm: 0 ... +∞;
a := 1; b := 1; m := 0; na := 0; nm := 0;
semWait(b); na ← na + 1; semSignal(b);
semWait(a); nm ← nm + 1;
semwait(b); na ← na - 1;
if na = 0 then semSignal(b); semSignal(m)
else semSignal(b); semSignal(a)
endif;
semWait(m); nm ← nm - 1;
;
if nm = 0 then semSignal(a)
else semSignal(m)
endif;
```

5.11 下面的问题曾被用于一个测试中：

侏罗纪公园有一个恐龙博物馆和一个公园，有 m 个旅客和 n 辆车，每辆车只能容纳一名旅客。旅客在博物馆逛了一会儿，然后派对乘坐旅客车。当一辆车可用时，它载入一名旅客，然后绕公园行驶任意长的时间。如果 n 辆车都已被旅客乘坐游玩，则想坐车的旅客需要等待；如果一辆车已经就绪，但没有旅客等待，那么这辆车等待。使用信号量同步 m 个旅客进程和 n 个进程。下面的代码框架是在教室的地板上发现的。忽略语法错误和丢掉的变量声明，请判定它是否正确。注意，`p` 和 `v` 分别对应于 `semwait` 和 `semSignal`。

Resource Jurassic_Park()

```
Sem car_avail:=0,car_taken:=0,car_fillde:=0,passenger_released:=0
```

```
Process passenger(i:=1 to num_passengers)
```

```
Do true->nap(int(random(1000*wander_time)))
    P(car_avail);V(car_taken);P(car_filled)
    P(passenger_released)
```

```
Od
```

```
End passenger
```

```
Process car(j:=1 to num_cars)
```

```
Do true->V(car_avail);P(car_taken);V(car_filled)
    Nap(int(random(1000*ride_time)))
    V(passenger_released)
```

```

Od
End car
End Jurassic_Park

```

答：这段代码有一个重要问题在 process car 中的代码 V(passenger_released) 能够解除下面一种旅客的阻塞，被阻塞在 P(passenger_released) 的这种旅客不是坐在执行 V() 的车里的旅客。

5.12 在图 5.9 和 5.3 的注释中，有一句话是“仅把消费者临界区（由 s 控制）中的控制语句移出还是不能解决问题，因为这将导致死锁”，请用类似于表 5.3 的表说明。

答：

	Producer	Consumer	s	n	delay
1			1	0	0
2	SemWaitB(S)		0	0	0
3	n++		0	1	0
4	If(n==1) (semSignalB(delay))		0	1	1
5	semSignalB(s)		1	1	1
6		semWaitB(delay)	1	1	0
7		semWaitB(s)	0	1	0
8		n--		0	0
9	semWaitB(s)	If(n==0) (semWaitB(delay))			
10					

生产者和消费者都被阻塞。

5.13 考虑图 5.10 中定义的无限缓冲区生产者/消费者问题的解决方案。假设生产者和消费者都以大致相同的速度运行，运行情况如下：

生产者： append;semSignal;produce;· · · append;semSignal

消费者： consume;take;semWait;consume;take;semWait;

生产者通常管理给换成区一个元素，并在消费者消费了前面的元素后发信号。生产者通常添加到一个空缓冲区中，而消费者通常取走缓冲区中的唯一元素。尽管消费者从不在信号量上阻塞，但必须进行大量的信号量调用，从而产生相当多的开销。

构造一个新程序使得能在这种情况下更加有效。

提示：允许 n 的值为 -1，这表示不仅缓冲区为空，而且消费者也检测到这个事实并将被阻塞，直到生产者产生新数据。这个方案不需要使用图 5.10 中的局部变量 m。

答：

这个程序来自于 [BEN82]

```

program producerconsumer;
var n: integer;
s: (*binary*) semaphore (= 1);
delay: (*binary*) semaphore (= 0);
procedure producer;
begin
repeat
produce;
semWaitB(s);
append;
n := n + 1;
if n=0 then semSignalB(delay);

```

```

semSignalB(s)
forever
end;
procedure consumer;
begin
repeat
semWaitB(s);
take;
n := n - 1;
if n = -1 then
begin
semSignalB(s);
semWaitB(delay);
semWaitB(s)
end;
consume;
semSignalB(s)
forever
end;
begin (*main program*)
n := 0;
parbegin
producer; consumer
parend
end.

```

5.14 考虑图 5.13. 如果发生下面的交换，程序的意义是否会发生改变？

- a.semWait(e);semWait(s)
- b.semSignal(s);semSignal(n)
- c.semWait(n);semWait(s)
- d.semSignal(s);semSignal(e)

答：只要交换顺序都会导致程序错误。信号量 **s** 控制进入临界区，你只想让临界区区域包括附加或采取功能。

5.15 在讨论有限缓冲区（见图 5.12）生产者/消费者问题时，注意我们的定义允许缓冲区中最多有 **n-1** 个入口？

- a. 这是为什么？
- b. 请修改程序，以不久这种低调？

答：如果缓冲区可以容纳 **n** 个入口，问题在于如何从一个满的缓冲区中区分出一个空的缓冲区，考虑一个有六个位置的缓冲区，且仅有一个入口，如下：

			A		
			Out	in	

然后，当一个元素被移出，**out=in** 现在假设缓冲区仅有一个位置为空：

D	E		A	B	C
		in	out		

这样，**out=in+1**但是，当一个元素被添加，**in** 被加 1 后，**out=in** 当缓冲区为空时同理。

- b. 你可以使用一个可以随意增加和减少的辅助的变量，**count**。

5.16 这个习题说明了使用信号量协调三类进程。圣诞老人在他北极的商店中睡眠，他只能被一下两种情况之一唤醒：(1) 所有九头驯鹿都从南太平洋的假期回来了，或者 (2) 某些小孩在制作玩具时遇到了困难。为了让圣诞老人多睡会，这些孩子只有在是那个人都遇到困难时才唤醒他。当三个孩子的问题得到解决时，其他想访问圣诞老人的孩子必须等到那些孩子返回。如果圣诞老人醒来后发现在三个孩子在他的店门口等待，并且最后一头驯鹿已经从热带回来。则圣诞老人决定让孩子门等到圣诞节之后，因为准备最后一天哦 iuxunlu 必须与其他 unlu 在暖棚中等待并且还没有套上缰绳做成雪橇前回来。请用信号量解决这个问题。

答：santa:圣诞老人 reindeer:驯鹿 elf:小孩子 sleigh:雪橇 toys:玩具

```
#define REINDEER 9 /* max # of reindeer
*/
#define ELVES 3 /* size of elf group */
/* Semaphores */
only_elves = 3, /* 3 go to Santa */
emutex = 1, /* update elf_cnt */
rmutex = 1, /* update rein_ct */
rein_semWait = 0, /* block early arrivals
back from islands */
sleigh = 0, /* all reindeer semWait
around the sleigh */
done = 0, /* toys all delivered */
santa_semSignal = 0, /* 1st 2 elves semWait
on
this outside Santa's shop
*/
santa = 0, /* Santa sleeps on this
blocked semaphore
*/
problem = 0, /* semWait to pose
the
question to Santa */
elf_done = 0; /* receive reply */
/* Shared Integers */
rein_ct = 0; /* # of reindeer back */
*/
elf_ct = 0; /* # of elves with problem
*/
/* Reindeer Process */
for (;;) {
tan on the beaches in the Pacific until
Christmas is close
semWait (rmutex)
rein_ct++
if (rein_ct == REINDEER) {
semSignal (rmutex)
semSignal (santa)
}
else {
semSignal (rmutex)
semWait (rein_semWait)
}
/* all reindeer semWaiting to be attached to
sleigh */
semWait (sleigh)
fly off to deliver toys
semWait (done)
head back to the Pacific islands
} /* end "forever" loop */
/* Elf Process */
for (;;) {
semWait (only_elves) /* only 3 elves
"in" */
semWait (emutex)
elf_ct++
if (elf_ct == ELVES) {
semSignal (emutex)
semSignal (santa) /* 3rd elf wakes
Santa */
}
else {
semSignal (emutex)
semWait (santa_semSignal) /*
semWait outside
Santa's shop door */
}
semWait (problem)
ask question /* Santa woke elf up */
semWait (elf_done)
semSignal (only_elves)
} /* end "forever" loop */
/* Santa Process */
for (;;) {
semWait (santa) /* Santa "rests" */
/* mutual exclusion is not needed on rein_ct
because if it is not equal to REINDEER,
then elves woke up Santa */
if (rein_ct == REINDEER) {
semWait (rmutex)
rein_ct = 0 /* reset while blocked */
semSignal (rmutex)
for (i = 0; i < REINDEER - 1; i++)
semSignal (rein_semWait)
for (i = 0; i < REINDEER; i++)
semSignal (sleigh)
deliver all the toys and return
for (i = 0; i < REINDEER; i++)
semSignal (done)
}
else { /* 3 elves have arrive */
for (i = 0; i < ELVES - 1; i++)
semSignal (santa_semSignal)
semWait (emutex)
elf_ct = 0
semSignal (emutex)
for (i = 0; i < ELVES; i++) {
semSignal (problem)
answer that question
semSignal (elf_done)
}
}
}
} /* end "forever" loop */

```

5.17 通过一下步骤说明消息传递和信号量具有同等的功能：

- 用信号量实现消息传递。提示：利用一个共享缓冲区保存信箱，每个信箱由一个消息槽数组成的。
- 用消息传递实现信号量。提示：引入一个独立的同步进程。

答：b.这个方法来自于[TANE97].同步进程维护了一个计数器和一个等待进程的清单。进程调用相关用于向同步进程发送消息的生产者，wait 或 signal，来实现 WAIT/HOLD SIGNAL. 然后生产者执行 RECEIVE 来

接受来自于同步进程的回复。

当消息到达时，同步进程检查计数器看需要的操作是否已经足够，**SIGNALs** 总是可以完成，但是假如信号值为 0 时，**WAITs** 将会被阻塞。假如操作被允许，同步进程就发回一个空消息，因此解除调用者的阻塞。假如操作是 **WAIT** 并且信号量的值为 0 时，同步进程进入调用队列，并且不发送回复。结果是执行 **WAIT** 的进程被阻塞。当 **SIGNAL** 被执行，同步进程选择一个进程在信号量上阻塞，要不就以先进先出顺序，要不以其他顺序，并且发送一个回复。跑步条件被允许因为同步进程一次只需要一个。

第 6 章 并发性：死锁和饥饿

6.1 写出图 6.1 (a) 中死锁的四个条件。

解：互斥：同一时刻只有一辆车可以占有一个十字路口象限。占有且等待：没有车可以倒退；在十字路口的每辆车都要等待直到它前面的象限是空的。非抢占：没有汽车被允许挤开其他车辆。循环等待：每辆汽车都在等待一个此时已经被其他车占领的十字路口象限。

6.2 按照 6.1 节中对图 6.2 中路径的描述，给出对图 6.3 中 6 种路径的简单描述。

解：1. Q 获得 B 和 A，然后释放 B 和 A。当 P 重新开始执行的时候，它将会能够获得两个资源。

2. Q 获得 B 和 A，P 执行而且阻塞在对 A 的请求上。Q 释放 B 和 A。当 P 重新开始执行的时候，它将会能够获得两个资源。

3. Q 获得 B，然后 P 获得和释放 A。Q 获得 A 然后释放

B 和 A。当 P 重新开始行的时候，它将会能够获得 B。

4. P 获得 A 然后 Q 获得 B。P 释放 A。Q 获得 A 然后释放

B。P 获得 B 然后释放 B。

5. P 获得，然后释放 A。P 获得 B。Q 执行而且阻塞在对 B 的请求上。P 释放 B。当 Q 重新开始执行的

r1	r2	r3	r4	r1	r2	r3	r4	r1	r2	r3	r4
----	----	----	----	----	----	----	----	----	----	----	----

时候，，它
将会能够
获得两个
资源。

6. P 获得
A 而且释

放 A 然后获得并且释放 B。当 Q 重新开始实行，它将会能够获得两个资源。

6.3 图 6.3 反映的情况不会发生死锁，请证明。

证明：如果 Q 获得 B 和 A（在 P 之前请求 A），那么 Q 能使用这些两类资源然后释放它们 允许 A 进行。如果 P 在 Q 之前请求 A 获得 A，然后 Q 最多能执行到请求 A 然后被阻塞。然而，一旦 P 释放 A，Q 能进行。一旦 Q 释放 B，A 能进行。

6.4 考虑下面的一个系统，当前不存在未满足的请求。

2	1	0	0
---	---	---	---

可用

r1 r2 r3 r4

当前分配

最大需求

仍然需求

0	0	1	2	0	0	1	2				
2	0	0	0	2	7	5	0				
0	0	3	4	6	6	5	6				
2	3	5	4	4	3	5	6				
0	3	3	2	0	6	5	2				

a计算每个进程仍然可能需要的资源，并填入标为

“仍然需要”的列中。

b系统当前是处于安全状态还是不安全状态，为什么。

c系统当前是否死锁？为什么？

d哪个进程（如果存在）是死锁的或可能变成死锁的？

e如果P3的请求(0, 1, 0, 0)到达，是否可以立即安全地同意该请求？在什么状态（死锁，安全，不安全）下可以立即同意系统剩下的全部请求？如果立即同意全部请求，哪个进程（如果有）是死锁的或可能变成死锁的？

解： a. 0 0 0 0

```

0 7 5 0
6 6 2 2
2 0 0 2
0 3 2 0

```

- b. 系统当前处于安全状态，因为至少有一个进程执行序列，不会导致死锁，运行顺序是p1, p4, p5, p2, p3.
- c. 系统当前并没有死锁，因为P1进程当前分配与最大需求正好相等，P1进程可以运行直至结束，接下来运行其他进程
- d. P2, P3, P4, P5可能死锁
- e. 不可以，当进程P1, P4, P5执行完可用资源为(4, 6, 9, 8)，P2, P3将死锁，所以不安全，完全不可以立即同意系统剩下的全部请求。

6.5 请把6.4中的死锁检测算法应用于下面的数据，并给出结果。

Available=(2 1 0 0)

$\begin{matrix} 2 & 0 & 0 & 1 \\ \text{Request} = & 1 & 0 & 1 & 0 \\ & 2 & 1 & 0 & 0 \end{matrix}$	$\begin{matrix} 0 & 0 & 1 & 0 \\ \text{Allocation} = & 2 & 0 & 0 & 1 \\ & 0 & 1 & 2 & 0 \end{matrix}$
--	---

解： 1. $W = (2 1 0 0)$

2. Mark P3; $W = (2 1 0 0) + (0 1 2 0) = (2 2 2 0)$
3. Mark P2; $W = (2 2 2 0) + (2 0 0 1) = (4 2 2 1)$
4. Mark P1; no deadlock detected 没有死锁

6.6 一个假脱机系统包含一个输入进程I，用户进程进程P和一个输出进程O，它们之间用两个缓冲区连接。进程以相等大小的块为单位交换数据，这些块利用输入缓冲区和输出缓冲区之间的移动边界缓存在磁盘上，并取决于进程的速度。所使用的通信原语确保满足下面的资源约束： $i+o \leq max$ 其中，max表示磁盘中的最大块数，i表示磁盘中的输入块数目，o表示磁盘中的输出块数目。

以下是关于进程的知识：

1. 只要环境提供数据，进程 I 最终把它输入到磁盘上（只要磁盘空间可用）。
2. 只要磁盘可以得到输入，进程 P 最终消耗掉它，并在磁盘上为每个输入块输出有限量的数据（只要磁盘空间可用）。
3. 只要磁盘可以得到输出，进程 O 最终消耗掉它。说明这个系统可能死锁。

解：当 I 的速度远大于 P 的速度，有可能使磁盘上都是输入数据而此时 P 进程要处理输入数据，即要将处理数据放入输出数据区。于是 P 进程等待磁盘空间输出，I 进程等待磁盘空间输入，二者死锁。

6.7 给出在习题 6.6 中预防死锁的附加资源约束，仍然通过输入和输出缓冲区之间的边界可以根据进程的要求变化。

解：为输出缓冲区保留一个最小数目（称为 reso）块，但是当磁盘空间足够大时允许输出块的数目超过这一界限。资源限制现在变成

$$\begin{aligned} I + O &\leq max \\ I &\leq max - reso \end{aligned}$$

当 $0 < reso < max$

如果程序 P 正在等候递送输出给磁盘，程序 O 最后处理所有的早先输出而且产生至少 reso 页，然后让 P 继续执行。因此 P 不会因为 O 而延迟。

如果磁盘充满 I/O，I 能被延迟；但是迟早，所有的早先的输入可以被 P 处理完，而且对应的输出将会被 O 处理，因而可以让 I 继续执行。

6.8 在 T H E 多道程序设计系统中，一个磁鼓（磁盘的先驱，用做辅存）被划分为输入缓冲区，处理和输出缓冲区，它们的边界可以移动，这取决于所涉及的进程速度。磁鼓的当前状态可以用以下参数描述：

max 表示磁鼓中的最大页数，i 表示磁鼓中的输入页数，p 表示磁鼓中的处理页数，o 表示磁鼓中的输出页数，reso 表示保留的最小页数，resp 表示保留的最小页数。

解：

$$\begin{aligned} I + O + P &\leq max - \\ I + O &\leq max - resp \\ I + P &\leq max - reso \\ I &\leq max - (reso + resp) \end{aligned}$$

6.9 在 T H E 多道程序设计系统中，一页可以进行下列状态转换：

1. 空 → 输入缓冲区（输入生产）
2. 输入缓冲区 → 处理区域（输入消耗）
3. 处理区域 → 输出缓冲区（输出生产）
4. 输出缓冲区 → 空（输出生产）
5. 空 → 处理区域（输出消耗）
6. 处理区域 → 空（过程调用）

a 根据 I, O 和 P 的量定义这些转换的结果。

b如果维持习题 6.6 中关于输入进程，用户进程和输出进程的假设，它们中的任何一个转换是否会导致死锁。

解： 1. $i \leftarrow i + 1$
2. $i \leftarrow i - 1; p \leftarrow p + 1$
3. $p \leftarrow p - 1; o \leftarrow o + 1$
4. $o \leftarrow o - 1$
5. $p \leftarrow p + 1$
6. $p \leftarrow p - 1$

b. 结合在对问题 6.7 的解决办法被列出的资源限制 我们能总结下列各项：

6. 过程返回能立刻发生因为他们只释放资源。
5. 程序调用可能用尽磁盘片 ($p = \max - reso$) 导致死锁。
4. 当有输出时输出消耗能立刻发生。
3. 输出生产能暂时被延迟直到所有的早先输出被消耗而且为下一步的输出至少可以产生 $reso$ 页。
2. 只要有输入，输入消耗能立刻发生。
1. 输入生产被延迟直到所有先前输入和对应的输出已经被消耗。此时，当 $i = o = 0$, 如果消费进程还没有占用完磁盘空间 ($p < \max - reso$), 可以产生输入。

结论：分配给消费进程的不受控制的存储空间是唯一可能引发死锁的因素。

6.1.0 考虑一个共有 150 个存储器单元的系统，其单元如下分配三个进程：

进程	最大	占用
1	70	45
2	60	40
3	60	15

使用银行家算法，以确定同意下面的任何一个请求是否安全。如果安全，说明能保证的终止序列；如果不安全，给出结果分配简表。

a. 第 4 个进程到达，最多需要 60 个存储单元，最初需要 25 个单元。

b. 第 4 个进程到达，最多需要 60 个存储单元，最初需要 35 个单元。

解： a. 若同意第 4 个进程请求，则储存器单元共用去 $25 + 15 + 40 + 45 = 125$ 个单元，还有 25 个存储单元，则可以安全执行全部进程。安全顺序是 1-2-3-4

b. 若同意第 4 个进程请求，则还有 15 个资源可以用，此时处于不安全状态，结果分配见表

进程	最大	占有	需要	空闲
1	70	45	25	15
2	60	40	20	
3	60	15	45	
4	60	35	25	

6.1.1 评价独家算法在实际应用中是否有用。

解：不切实际的：不能总是预先知道最大要求，进程数目和资源数目可能随着时间改变。大多数的操作系统忽视死锁。

6.1.2 有一个已经实现了的管道算法，使得进程 P_0 产生的 T 类型的数据元素经进程序列 $P_1 P_2 \dots P_{n-1}$ ，并且按该顺序在元素上操作。

a. 定义一个一般的消息缓冲区，包含所有部分消耗的数据元素，并按下面的格式为进程 P_i ($0 \leq i \leq n-1$) 写一个算法。

Repeat

从前驱接收

消耗

给后续发送

Forever

假设 P0 收到 Pn-1 发送的空元素。该算法能够使进程直接在缓冲区中保存的消息上操作，而无需复制。

b. 说明关于普通的缓冲区进程不会死锁。

```
解: ar buffer: array0..max-1 of sharedT;  
      available: shared array0..n-1 of 0..max;  
      "Initialization"  
var K: 1..n-1;  
region available do  
begin  
  available(0) := max;  
  for every k do available(k) := 0;  
end  
"Process i"  
var j: 0..max-1; succ: 0..n-1;  
begin  
  j := 0; succ := (i+1) mod n;  
  repeat  
    region available do  
      await available(i) > 0;  
      region buffer(j) do consume element;  
      region available do  
        begin  
          available(i) := available(i) - 1;  
          available(succ) := available(succ) + 1;  
        end  
        j := (j+1) mod max;  
    forever  
end
```

b. 死锁可以被解决通过

P0 waits for Pn-1 AND

P1 waits for P0 AND

.

Pn-1 waits for Pn-2

因为

(available(0) = 0) AND

(available(1) = 0) AND

.

(available(n-1) = 0)

但是如果 max > 0, 这个条件不成立, 因为临界域满足

$claim(1) + claim(2) + \dots + claim(n)$

$< available(1) + available(2) + \dots + available(n)$

= max

6.1.3 a. 3个进程共享4个资源单元，一次只能保留或释放一个单元。每个进程最大需要2个单元。说明不会死锁。

b. N个进程共享M个资源单元，一次只能保留或释放一个单元。每个进程最大需要单元数不超过M，并且所有最大需求的总和小于M+N。说明不会发生死锁。

解：a. 说明由于每个进程最多需要2个资源，最坏情况下，每个进程需要获得一个，系统还剩1个，这一个资源，无论分给谁都能完成。完成进程释放资源后，使剩余进程也完成，故系统不会死锁。

b. 假定每个进程最多申请X个资源，最坏情况下，每个进程都得到X-1个资源都在申请最后一个资源，这时系统剩余资源数量为M-N(X-1)，只要系统还有一个剩余资源，就可以使其中的一个进程获得所需要的全部资源，该进程运行结束以后释放资源，就可以使其他进程得到全部资源的满足，因此，当M-N(X-1)》1时系统不会发生死锁，解这个不等式X《(M+N-1)，系统不会发生死锁，因此，当所有进程的需求总和小于M+N时，系统是不会发生死锁的。

6.1.4 考虑一个由四个进程和一个单独资源组成的系统，当前的声明和分配矩阵是

$$\begin{array}{cc} & \begin{matrix} 3 & 1 \\ 2 & 1 \\ C = & A = \\ 9 & 3 \\ 7 & 2 \end{matrix} \end{array}$$

对于安全状态，需要的最小资源数目是多少？

解：最小资源数是3个，总共有10个资源。P2获得一个资源，完成后释放两个资源，P1获得三个资源，完成后释放三个资源，接下来P4获得五个资源，释放完资源后，P3获得所需的6个资源后完成。

6.1.5 考虑下列处理死锁的方法：1 银行家算法，2 死锁检测并杀死线程，释放所有资源，3 事先保留所有资源，4 如果线程需要等待，5 资源排序，6 重新执行检测死锁并退回线程的动作。

评价解释死锁的不同方法使用的一个标准是，哪种方法允许最大的并发。换言之，在没有死锁时，哪种方法允许最多数目的线程无需要等待继续前进？对下面列出的6种处理死锁的方法，给出从1到6的一个排序（1表示最大程序的并发），并解释你的排序。

另一个标准是效率；哪种方法需要最小的处理器开销？假设死锁很少发生，给出各种方法从1到6的一个排序（1表示最有效），并解释这样排序的原因。如果死锁发生很频繁，你的顺序需要改变吗？

解：a. 从最多并发事件到最少，有一个大概的次序如下：

1. 死锁检测并杀死线程，释放所有资源

发现死锁并退回线程的动作，如果线程需要等候那么重新开始线程而且释放所有的资源，在死锁发生之前，这些运算法则都不会限制并发，因为他们仰赖运行时间检查而非静态的限制。他们的效果在死锁被发现比较难以描述：他们仍然允许许多并发（在一些情形，他们增加它），但是很难有准确的估计。第三个运算法则是最奇怪的，因为如此后许多的它的并发将会是无用的重复，因为线程竞争执行时间，这一个运算法则也影响运行速度。因此在两者的极端，它以这顺序被列出两次。

2. 银行家的运算法则

资源排序

这些运算法则因为限制多种的可允许的计算，而相对早先两种法则会引起更多的不必要的等候。银行家的运算法则避免不安全的配置和资源排序限制配置序列以便线程在他们是否一定等候的时候有较少的选择。

3. 事先保留所有的资源

这一个运算法则相比前两个要允许更少的并发，但是比最坏的那一种有更少的缺点。因为要预先保留所有的资源，线程必须等候比较长的而且当他们工作的时候更有可能阻塞其他的线程因此系统上来说具有更多的线性。

4. 如果线程需要等待，则重新启动线程并且释放所有的资源

如上所述，这一个运算法则在区别最多和最少并发上有疑问。这具体要看并发的定义。

b从最高效率到最低，有如下大概的一个顺序：

1. 预先保留所有的资源

资源排序

因为他们没有包括运行时间经常开支，所以这些运算法则最有效率。

注意这是在相同的静态限制下的结果。

2. 银行家的运算法则

发现死锁而且杀死线程，释放它的资源

这些运算法则在概略的配置上包括运行时间检查

；银行家的运算法则运行搜寻查证复杂度在线程和配置的数字和死锁检测中是 $O(n \cdot m)$ ，死锁检测的复杂度是 $O(n)$ 。资源-从属链被线程数目，资源数目和分配数目限制。

3. 发现死锁并退回线程的动作

这一个运算法则运行也需要运行上述的相同的时间检查。

在写内存上的复杂度为 $O(n)$ 。

4. 如果线程需要等待，则重新启动线程并释放所有的资源

这一个运算法则是非常无效率，有如下两个理由。首先，因为线程有重新开始的危险，他们完成的可能性低。其次，他们与其他重新开始线程竞争有限的执行时间，因此整个系统运行的速度很慢。

6.1.6 评价下面给出的就餐问题的解决方案。一位饥饿的哲学家首先拿起他左边的叉子，如果他右边的叉子也是可用的，则拿起右边的叉子开始吃饭，否则他放下左边的叉子，并重复这个循环。

解：如果哲学家们步调完全一致地拿起左边叉子又放下的话，他们会重复这一过程，导致饥饿情况的出现。

6.1.7 假设有两种类型的哲学家。一类总是先拿起左边的叉子（左撇子），另一类总是先拿起右边的叉子（右撇子）。左撇子的行为和图 6.1.2 中定义的一致。右撇子的行为如下：

begin

```
repeat
    think;
    wait(fork[(i+1) mod 5]);
    wait(fork[i]);
    eat;
    signal(fork[i]);
    signal(fork[(i+1) mod 5]);
```

forever;

end;

证明：a 如果至少有一个左撇子或右撇子，则他们的任何就座安排都可以避免死锁。

b 如果至少有一个左撇子或右撇子，则他们的任何就座安排都可以防止饥饿。

解：a 假设存在死锁情况，设有 D 个哲学家，他们每人都有一支叉子而且另一支叉子被邻居占有。不失一般性，设 P_j 是一个左撇子。 P_j 抓牢他的左边叉子而且没有他的右边叉子，他的右边邻居 P_k 没有完成就餐因此也是一个左撇子。因此依次推理下去所有这 D 个哲学家都是左撇子。这与既有左撇子又有右撇子的条件矛盾，假设不成立，不存在死锁。

b

假设左撇子 P_j 饥饿，也就是说，有一部分人在就餐而 P_j 从不吃。假如 P_j 没有叉子。这样 P_j 的左边邻居 P_i 一定持续地占有叉子而始终不吃完。因此 P_i 是右撇子，

抓住他的右边叉子，但是从得不到他的左边叉子来完成就餐也就是说 P_i

也饥饿。现在 P_i 左边邻居也一定是持续占有右边叉子的右撇子。向左进行这样的推理，得出所有哲学家都是饥饿的右撇子，这同 P_j 是个左撇子矛盾。因此 P_j 一直拥有左边叉子而且在等待他的右边叉子。 P_j 的右边邻居 P_k 一直举着他的左边叉子而且从不完成一餐也就是， P_k 是也饥饿的左撇子。如果 P_k 不是

一直拿着他的左边叉子， P_j 就可以就餐；因此 P_k 拿着他的左边叉子。向右推理可得所有哲学家都是饥饿的左撇子。这与条件矛盾，因此假设不成立，没有人饥饿。

6.1.8 图 1.1.7 显示了另外一个使用管程解决哲学家就餐问题的方法。和图 6.1.4 比较并阐述你的结论。

解：图 6.1.4 是等待可用的叉子，图 6.1.7 是等待邻居吃完，在本质上逻辑是一样的，后者显得更加紧凑。

6.1.9 在表 6.1.3 中，Linux 的一些原子操作不会涉及到对同一变量的两次访问。比如 `atomic_read(atomic_t *)` 简单的读操作在任何体系结构中都是原子的。为什么该操作增加到了原子操作的指令表？

解：原子操作是在原子数据类型上操作，原子数据类型有他们自己的内在的格式。因此，不能用简单的阅读操作，但是特别的阅读操作对于原子数据类型来说是不可或缺的。

6.2.0 考虑 Linux 系统中的如下代码片断：

```
read_lock(&mr_rwlock);  
write_lock(&mr_rwlock);
```

`mr_rwlock` 是读者写者锁。这段代码的作用是什么？

解：因为写者锁将会自旋，所以这段代码会导致死锁。等待所有的读者解锁，包括唤醒这个线程。

6.2.1 两个变量 `a` 和 `b` 分别有初始值 1 和 2，对于 Linux 系统有如下代码：

线程 1	线程 2
<code>a = 3;</code>	--
<code>mb();</code>	--
--	<code>c = b;</code>
--	<code>rmb();</code>
	<code>d = a;</code>

使用内存屏障是为了避免什么错误？

解：没有使用内存屏障，在一些处理器上可能 `c` 接到 `b` 的新值，而 `d` 接到 `b` 的旧值。举例来说，`c` 可以等于 4（我们期待的），然而 `d` 可能等于 1.（不是我们期待的）。使用 `mb()` 确保 `a` 和 `b` 按合适的次序被写，使用 `rmb()` 确保 `c` 和 `d` 按合适的次序被读。

第 7 章 内存管理

7.1.2.3 节中列出了内存管理的 5 个目标，7.1 节中列出了 5 中需求。请说明它们是一致的。

答：重定位≈支持模块化程序设计；

保护≈保护和访问控制以及进程隔离；

共享≈保护和访问控制；

逻辑组织≈支持模块化程序设计；

物理组织≈长期存储及自动分配和管理

7.2. 考虑使用大小相等分区的固定分区方案。分区大小为 $2e16$ 字节，贮存的大小为 $2e24$ 字节。使用一个进程表来包含每一个进程对应的分区。这个指针需要多少位？

答：分区的数量等于主存的字节数除以每个分区的字节数： $224/216 = 28$ ，需要 8 个比特来确定一个

分区大小为 28 中的某一个位置。

- 7.3. 考虑动态分区方案，说明平均内存中空洞的数量是段数量的一半。

答：设 n 和 h 为断数量和空洞数量的个数 在主存中, 每划分一个断产生一个空洞的概率是 0.5, 因为删除一个断和添加一个断的概率是一样的 假设 s 是内存中断的个数那么空洞的平均个数一定等于 $s/2$. 而导致空洞的个数一定小余断的数量的直接原因是相邻的两个断在删除是一定会产生一个空洞.

- 7.4. 在实现动态分区中的各种放置算法（见7.2节），内存中必须保留一个空闲块列表。分别讨论最佳适配、首次适配、临近适配三种方法的平均查找长度。

答：通过上题我们知道，假设 s 是驻留段的个数，那么空洞的平均个数是 $s/2$ 。从平均意义上讲，平均查找长度是 $s/4$ 。

- 7.5. 动态分区的另一种放置算法是最坏适配，在这种情况下，当调入一个进程时，使用最大的空闲存储块。该方法与最佳适配、首次适配、邻近适配相比，优点和缺点各是什么？它的平均查找长度是多少？

答：一种对最佳适配算法的评价即是为固定分配一个组块后和剩余空间是如此小以至于实际上已经没有什么用处。最坏适配算法最大化了在一次分配之后，剩余空间的大小仍足够满足另一需求的机率，同时最小化了压缩的概率。这种方法的缺点是最大存储块最早被分配，因此大空间的要求可能无法满足。

- 7.6. 如果使用动态分区方案，下图所示为在某个给定的时间点的内存配置：

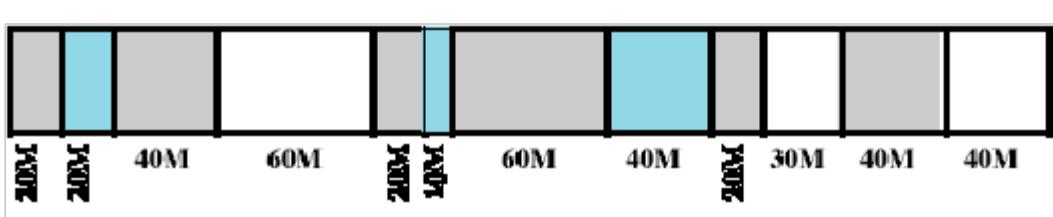
阴影部分为已经被分配的块；空白部分为空闲块。接下来的三个内存需求分别为 40MB, 20MB 和 10MB。

分别使用如下几种放置算法，指出给这三个需求分配的块的起始地址。

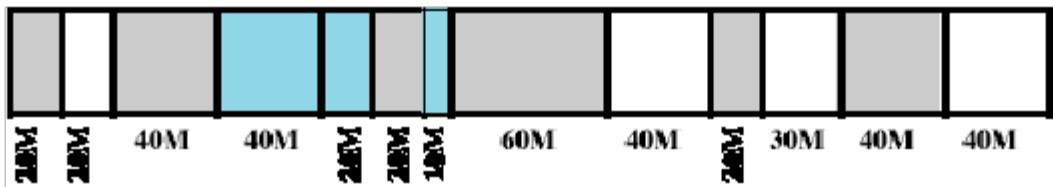
- 首次适配
- 最佳适配
- 临近适配（假设最近添加的块位于内存的开始）
- 最坏适配

答：

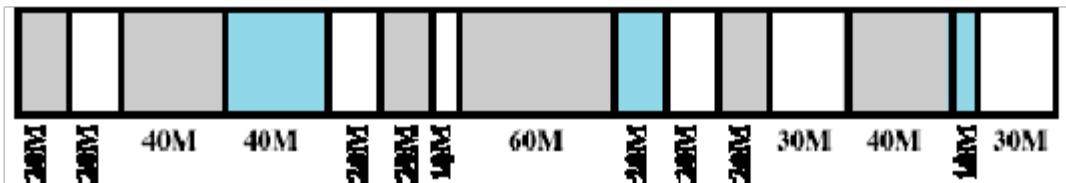
- 40M 的块放入第 2 个洞中，起始地址是 80M. 20M 的块放入第一个洞中. 起始地址是 20M. 10M 的块的起始地址是 120M。
- 40M, 20N, 10M 的起始地址分别为 230M, 20M 和 160M.



- 40M, 20M, 10M 的起始地址是 80M, 120M, 160M.



- 40M, 20M, 10M, 的起始地址是 80M, 120M, 160M.



- 7.7. 使用伙伴系统分配一个 1MB 的存储块。

- 利用类似于图 7.6 的图来说明按下列顺序请求和返回的结果：请求 70；请求 35；请求 80；返回 A；请求 60；返回 B；返回 D；返回 C。

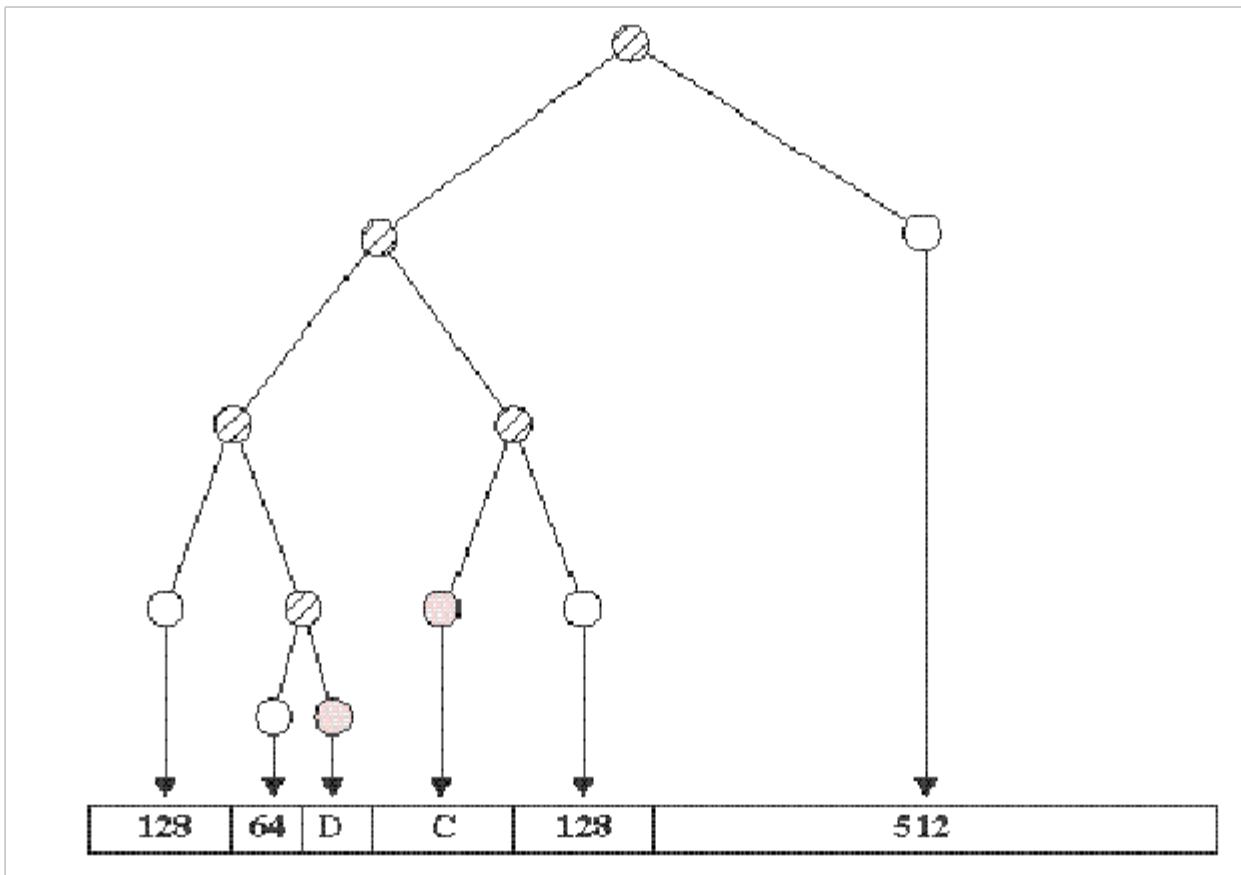
b. 给出返回 B 之后的二叉树表示。

答:

a.

Request 70	A	128	256	512
Request 35	A	B	64	256
Request 80	A	B	64	C
Return A	128	B	64	C
Request 60	128	B	D	C
Return B	128	64	D	C
Return D	256		C	128
Return C				512
				1024

b.



7.8. 考虑一个伙伴系统，在当前分配下的一个特定块地址为11011110000.

a. 如果块大小为4，它的伙伴的二进制地址为多少？

b. 如果块大小为16，它的伙伴的二进制地址为多少？

答:

a. 011011110100

b. 011011100000

7.9. 令 $\text{buddy}_k(x)$ 为大小为 2^k 、地址为 x 的块的伙伴的地址，写出 $\text{buddy}_k(x)$ 的通用表达式。

答:

$$\text{buddy}_k(x) = \begin{cases} x + 2^k & \text{if } x \bmod 2^{k+1} = 0 \\ x - 2^k & \text{if } x \bmod 2^{k+1} = 2^k \end{cases}$$

7.10. Fibonacci 序列定义如下:

$$F_0=0, F_1=1, F_{n+2}=F_{n+1}+F_n, n \geq 0$$

a. 这个序列可以用于建立伙伴系统吗？

b. 该伙伴系统与本章介绍的二叉伙伴系统相比，有什么优点？

答:

- a. 是。字区大小可以确定 $F_n = F_{n-1} + F_{n-2}$ 。
- b. 这种策略能够比二叉伙伴系统提供更多不同大小的块，因而具有减少内部碎片的可能性。但由于创建了许多没用的小块，会造成更多的外部碎片。

7. 11. 在程序执行期间，每次取指令后处理器把指令寄存器的内容（程序计数器）增加一个字，但如果遇到会导致在程序中其他地址继续执行的转跳或调用指令，处理器将修改这个寄存器的内容。现在考虑图 7.8。关于指令地址有两种选择：

- 在指令寄存器中保存相对地址，并把指令寄存器作为输入进行动态地址转换。当遇到一次成功的转跳或调用时，由这个转跳或调用产生的相对地址被装入到指令寄存器中。
- 在指令寄存器中保存绝对地址。当遇到一次成功的转跳或调用时，采用动态地址转换，其结果保存到指令寄存器中。

哪种方法更好？

答：使用绝对地址可以减少动态地址转换的次数。但是，我们希望程序能够被重定位。因此，在指令寄存器中保存相对地址似乎就更好一些。也可以选择在进程被换出主存时将指令寄存器中的地址转换为相对地址。

7. 12. 考虑一个简单分页系统，其物理存储器大小为 2^{32} 字节，页大小为 2^{10} 字节，逻辑地址空间为 2^{16} 个页。

- a. 逻辑地址空间包含多少位？
- b. 一个帧中包含多少字节？
- c. 在物理地址中指定帧需要多少位？
- d. 在页表中包含多少个页表项？
- e. 在每个页表项中包含多少位？（假设每个页表项中包含一个有效/无效位）

答:

- a. 物理地址空间的比特数是 $2^{16} * 2^{10} = 2^{26}$
- b. 一个帧包含的字节跟一个页是一样的 2^{10} 比特。
- c. 主存中帧的数量是 $2^{32} / 2^{10} = 2^{22}$ ，所以每个帧的定位要 22 个比特
- d. 在物理地址空间，每个页都有一个页表项，所以有 2^{16} 项
- e. 加上有效/无效位，每个页表项包含 23 位。

7. 13. 分页系统中的虚地址 a 相当于一对 (p, w) ，其中 p 是页号， w 是页中的字节号。令 z 是一页中的字节总数，请给出 p 和 w 关于 z 和 a 的函数。

答：关系是： $a = pz + w$ 其中 $p = \lfloor a/z \rfloor$ ， a/z 的整数部分。 $w = Rz(a)$ ， a 除以 z 的余数

7. 14. 在一个简单分段系统中，包含如下段表：

起始地址	长度（字节）
660	248
1752	442
222	198
996	604

对如下的每一个逻辑地址，确定其对应的物理地址或者说明段错误是否会发生：

- a. 0, 198
- b. 2, 256
- c. 1, 530
- d. 3, 444
- e. 0, 222

答:

- a. 段 0 定位在 660，所以我们有物理地址 $660 + 190 = 858$ 。
- b. $222 + 156 = 378$
- c. 段 1 长度为 422，所以会发生错误

d. $996+444=1440$

e. $660+222=882$.

7. 15. 在内存中，存在连续的段 S_1, S_2, \dots, S_n 按其创建顺序一次从一端放置到另一端，如下图所示：

当段 S_{n+1} 被创建时，尽管 S_1, S_2, \dots, S_n 中的某些段可能已经被删除，段 S_{n+1} 仍被立即放置在段 S_n 之后。

当段（正在使用或已被删除）和洞之间的边界到达内存的另一端时，压缩正在使用的段。

a. 说明花费在压缩上的时间 F 遵循以下的不等式：

$$F \geq (1-f)/(1+kf), k=t/2s-1$$

其中， s 表示段的平均长度（以字为单位）， l 标识段的平均生命周期，按存储器访问； f 表示在平衡条件下，未使用的内存部分。提示：计算边界在内存中移动的平均速度，并假设复制一个字至少需要两次存储器访问。

b. 当 $f=0.2, t=1000, s=50$ 时，计算 F 。

答：

a. 很明显，在一个周期 t 内一些段会产生而一些段会被删除。因为系统是公平的，一个新的段会在 t 内被插入，此外，边界会以 s/t 的速度移动。假设 t_0 是边界到达空洞的时间， $t_0=fmr/s$ ， m 内存的长度，在对段进行压缩时会有 $(1-f)m$ 个数被移动，压缩时间 t_c 至少是 $2(1-f)m$ 。则花在压缩上的时间 F 为 $F=1-t_0/(t_0+t_c)$ 。

b. $K=(t/2s)-1=9; F \geq (1-0.2)/(1+1.8)=0.29$

第 8 章 虚拟内存

8.1 假设在处理器上执行的进程的页表如下所示。所有数字均为十进制数，每一项都是从 0 开始记数的，并且所有的地址都是内存字节地址。页尺寸为 1024 个字节。

虚拟页号	有效位	访问位	修改位	页帧号
0	1	1	0	4
1	1	1	1	7
2	0	0	0	—
3	1	0	0	2
4	0	0	0	—
5	1	0	1	0

a. 描述 CPU 产生的虚拟地址通常是如何转化成一个物理主存地址的。

b. 下列虚拟地址对应于哪个物理地址（不用考虑页错误）？

(i) 1052

(ii) 2221

(iii) 5499

解答

a: 由虚拟地址求得页号和偏移量，用虚拟页号作为索引页表，得到页帧号，联系偏移量得到物理地址

b: (i) $1052 = 1024 + 28$ 查表对应的页帧号是 7，因此物理地址为 $7 * 1024 + 28 = 7196$

(ii) $2221 = 2 * 1024 + 173$ 此时出现页错误

(iii) $5499 = 5 * 1024 + 379$ 对应的页帧号为 0 因此物理地址是 379

8.2 考虑一个使用 32 位的地址和 1KB 大小的页的分页虚拟内存系统。每个页表项需要 32 位。需要限制页表的大小为一个页。

a. 页表一共需要使用几级？

b. 每一级页表的大小是多少？提示：一个页表的大小比较小。

c. 在第一级使用的页较小与在最底下一级使用的页较小相比，那种策略使用最小个数的页？

解答

a: 虚拟内存可以分为 $2^{32}/2^{10} = 2^{22}$ 页，所以需要 22 个 bit 来区别虚拟内存中的一页，每一个页表可以包含 $2^{10}/4=2^8$ 项，因此每个页表可以包含 22bit 中的 8 个 bit，所以需要三级索引。

b: 第二级页表有 2^8 个页表项，第一级页表有 2^6 个页表项。

c: 如果顶层有 2^6 个页表项将会减少使用空间，在这种情况下，中间层页表有 2^6 个并且每个都有 2^8 个页表项，底层有 2^{14} 个页并且每个都有 2^8 个页表项，因此共有 $1+2^6+2^{14}$ 页 = 16,449 页。如果中间层有 2^6 个页表项，那么总的页数有 $1+2^8+2^{14}$ 页 = 16,641 页。如果底层有 2^6 个页表项，那么总的页表数是 $1+2^8+2^{16}$ 页 = 65,973 页。

8.3 a: 图 8.4 中的用户表需要多少内存空间？

b: 假设需要设计一个哈希反向页表来实现与图 8.4 中相同的寻址机制，使用一个哈希函数来将 20 位页号映射到 6 位哈希表。表项包含页号帧号和链指针。如果页表可以给每个哈希表项分配最多 3 个溢出项的空间，则哈希反向页表需要占用多大的内存空间？

解答

a: 4Mbyte

b: 行数: $2^{6+2}=128$ 项。每项包含: 20 (页号) + 20 (帧号) + 8bits (链索引) = 48bits = 6bytes
总共: $128*6=768$ bytes

8.4 一个进程分配给 4 个页帧(下面的所有数字均为十进制数,每一项都是从 0 开始计数的)。上一次把一页装入到一个页帧的时间, 上一次访问页帧中的页的时间, 每个页帧中的虚拟页号以及每个页帧的访问位 (R) 和修改位 (M) 如下表所示 (时间均为从进程开始到该事件之间的时钟时间, 而不是从事件发生到当前的时钟值)。

虚拟页号	页帧	加载时间	访问时间	R 位	M 位
2	0	60	161	0	1
1	1	130	160	1	0
0	2	26	162	1	0
3	3	20	163	1	1

当虚拟页 4 发生错误时, 使用下列内存管理策略, 哪一个页帧将用于置换? 解释原因。

a.FIFO (先进先出) 算法

b.LRU (最近最少使用) 算法

c.Clock 算法

d.最佳 (使用下面的访问串) 算法

e.在页错误之前给定上述内存状态, 考虑下面的虚拟页访问序列:

4,0,0,2,4,2,1,0,3,2

如果使用窗口大小为 4 的工作集策略来代替固定分配, 会发生多少页错误? 每个页错误何时发生?

解答

a: 页帧 3, 在时间 20 加载, 时间最长。

b: 页帧 1, 在时间 160 访问距现在时间最长。

c: 清除页帧 3 的 R 位 (最早加载), 清除页帧 2 的 R 位, (次最早加载), 换出的是页帧 0 因为它的 R 位为 0。

d: 换出的是页帧 3 中的虚拟页 3, 因为它将最晚被访问到。

e: 一共有 6 个错误, 如下

	*	4	0	0	0	2	*	4	2	*	1	*	0	*	3	*	2
VPN of pages in memory in LRU order	3 0 2 1	4 3 0 2	0 4 3 0	0 4 3 0	0 4 3 0	2 0 3 0	*	4 2 0 0	2 4 0 0	*	1 4 2 4	*	0 2 4 2	*	3 1 4 2	*	2 1 2 2

8.5 一个进程访问 5 页：A,B,C,D 和 E，访问顺序如下：

A;B;C;D;A;B;E;A;B;C;D;E

假设置换算法为先进后出，该进程在主存中有三个页帧，开始时为空，请查找在这个访问顺序中传送的页号。对于 4 个页帧的情况，请重复上面的过程。

解答

分别有 9 次和 10 次页错误，这被称之为“Belady's 现象” ("An Anomaly in Space-Time Characteristics of Certain Programs Running in a Paging Machine," by Belady et al, *Communications of the ACM*, June 1969)

8.6 一个进程在磁盘上包含 8 个虚拟页，在主存中固定分配给 4 个页帧。发生如下顺序的页访问：

1,0,2,2,1,7,0,1,2,0,3,0,4,5,1,5,2,4,5,6,7,6,7,2,4,2,7,3,3,2,3

a. 如果使用 LRU 替换策略，给出相继驻留在这 4 个页帧中的页。计算主存的命中率。假设这些帧最初是空的。

b. 如果使用 FIFO 策略，重复问题 (a)。

c. 比较使用这两种策略的命中率。解释为什么这个特殊的访问顺序，使用 FIFO 的效率接近于 LRU。

解答

a:LRU: 命中率=16/33

1	0	2	2	1	7	6	7	0	1	2	0	3	0	4	5	1	5	2	4	5	6	7	6	7	2	4	2	7	3	3	2	3	
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	4	4	4	4	4	4	4	4	4	4	4	4	2	2	2	2	2	2	
-	0	0	0	0	6	6	6	6	2	2	2	2	2	2	5	5	5	5	5	5	5	5	5	5	5	5	4	4	4	4	4	4	
-	-	2	2	2	2	2	2	0	0	0	0	0	0	0	0	0	0	0	2	2	2	2	7	7	7	7	7	7	7	7	7	7	7
-	-	-	-	-	7	7	7	7	7	7	7	3	3	3	3	1	1	1	1	1	1	6	6	6	6	6	6	6	3	3	3	3	3
F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F

b:FIFO: 命中率=16/33

1	0	2	2	1	7	6	7	0	1	2	0	3	0	4	5	1	5	2	4	5	6	7	6	7	2	4	2	7	3	3	2	3	
1	1	1	1	1	6	6	6	6	6	6	4	4	4	4	4	4	4	4	4	4	6	6	6	6	6	6	6	6	6	2	2		
-	0	0	0	0	0	0	1	1	1	1	1	5	5	5	5	5	5	5	5	5	7	7	7	7	7	7	7	7	7	7	7	7	
-	-	2	2	2	2	2	2	2	2	2	2	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	4	4	4	4	4	4
-	-	-	-	-	7	7	7	7	7	7	7	3	3	3	3	3	3	3	3	2	2	2	2	2	2	2	2	2	3	3	3	3	3
F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F

c:这两种策略对这个特殊的页轨迹（执行顺序）是等效的。

8.7 在 VAX 中，用户页表以系统空间的虚拟地址进行定位。让用户页表位于虚存而不是主存中有什么好处？有什么缺点？

解答

最主要的优点是在物理内存空间上的节省。这主要是两方面的原因：(1) 一个用户页表可以仅当使用时才取入内存。(2) 操作系统可以动态的分配用户页表，产生一个页表仅当程序被创建时。

当然，也有一个缺点：地址转换需要多余的工作。

8.8 假设在主存中执行下列程序语句：

for(i=1;i≤n;i++)

$$a[i] = b[i] + c[i];$$

页尺寸为 1000 个字。令 $n=1000$ 。使用一台具有所有寄存器指令并使用了索引寄存器的机器，写出实现上述语句的一个假想程序，然后给出在执行过程中的页访问顺序。

解答

由机器语言编写的程序，在主存中地址 4000 处开始运行。运行情况如下：

```

4000 (R1) ← 1 建立索引记录 i
4001 (R1) ← n 在 R2 中建立 n
4002 比较 R2, R1 检查 i > n
4003 如果大于则跳转到 4009
4004 (R3) ← B(R1) 使用索引记录 R1 到达 B[i]
4005 (R3) ← (R3) + C(R1) 使用索引记录 R1 加上 C[i]
4006 A(R1) ← (R3) 使用索引记录 R1 将总和存入 A[i] 中
4007 (R1) ← (R1) + 1 i 加一
4008 跳到 4002
6000—6999 存储 A
7000—7999 存储 B
8000—8999 存储 C
9000 存储 1
9001 存储 n

```

由这个循环产生的参考串为

494944 (4748464944)₁₀₀₀

包括 11,000 个参考，但仅包括 5 个不寻常的页

8.9 IBM System/370 体系结构使用两级存储器结构，并且分别把这两级称为段和页，这里的分段方法缺少本章所描述的关于段的许多特征。对于这个基本的 370 体系结构，页尺寸可以是 2KB 或 4KB，段大小固定为 64KB 或 1MB。这种方案缺少一般分段系统的那些优点？370 的分段方法有什么好处？

解答

S/370 分段系统是固定的且对程序员是不可见的，因此没有一个分段系统的优点在 S/370 中实现（无保护情况下）每一个段表项的 P 位提供整个段表的保护。

8.10 假设页尺寸为 4KB，页表项大小为 4 字节。如果要映射一个 64 位地址空间，并且最顶层的页表对应于一页，则需要几级页表？

解答

因为每个页表项有 4bytes，每个页表有 4Kbytes，所以每个页表可以映射 $1024 = 2^{10}$ 个页，标识出 $2^{10} \times 2^{12} = 2^{22}$ bytes 的地址空间。然而，地址空间是 2^{64} bytes。增加一个二层页表，顶层页表指向 2^{10} 个页表，标识出 2^{32} 个页表，将这个过程继续下去就得到：

深度	地址空间
1	2^{22} bytes
2	2^{32} bytes
3	2^{42} bytes
4	2^{52} bytes
5	2^{62} bytes
6	2^{72} bytes ($> 2^{64}$ bytes)

我们可以看到 5 层是不够表示 64 位的地址空间，但是 6 层达到了要求。但是 6 层中只有 2 位被使用，而不是全部的 10 位。所以不是使用 72 位的虚拟地址空间，而是将除了最低两位外的其他位全部屏蔽忽略。这样将会得到一个 64 位地址空间，顶层页只有 4 个页表项。另外一种方法是修改规则将顶层页做成一个单独的物理页并且让它适合 4 个页。这样将会节省一个

页。

8.11 考虑一个系统该系统采用基于页的内存映射，并使用一级页表。假设页表总是在主存中。

a.如果一次存储器访问需要 200ns，那么一次需要调页的存储器访问要多长时间？

b.现在增加一个 MMU，在命中或未命中时有 20ns 的开销。如果假设有 85% 的存储器访问命中都在 MMU TLB 中，那么哪些是有效的存储器访问时间？

c.解释 TLB 命中率如何影响有效的存储器访问时间。

解答

a.400ns。200ns 用来得到页表项，200ns 用到达存储位置

b.这是一个常见的有效时间计算公式：

$$(220 \times 0.85) + (420 \times 0.15) = 250$$

两种情况：第一种，TLB 中包含所需的页表项。在这种情况下在 200ns 外多了 20ns 的存储访问时间。第二种，TLB 中不包含所需的页表项。这时我们会再多花 200ns 来把所需的页表项取入 TLB。

c.TLB 命中率越高有效存储器访问时间就越短，因为额外的 200ns 来得到页表项的时间被节省了。

8.12 考虑一个进程的页访问序列，工作集为 M 帧，最初都是空的。页访问串的长度为 P ，包含 N 个不同的页号。对任何一种页替换算法，

a.页错误次数的下限是多少？

b.页错误次数的上限是多少？

解答

a.N

b.P

8.13 在论述一种页替换算法时，一位作者用一个在循环轨道上来回移动的雪犁机来模拟说明：雪均匀地落在轨道上，雪犁机一恒定的速度在轨道上不断的循环，轨道上被扫落的雪从系统中消失。

a.8.2 节讨论的哪一种页替换算法可以用它来模拟？

b.这个模拟说明了页替换算法的那些行为？

解答

a.这是一个很好的时钟算法的类似。雪落在轨道上类似于页到达循环页缓存中。时钟算法时钟算法指针的移动类似于雪犁机的移动。

b.注意到在时钟指针最近的前面可替换页的密度是最高的，就好像在雪犁机最近的前面的雪是最厚的一样。因此我们可以认为时钟算法在寻找替换页时是非常有效的。事实上可以看到雪犁机前雪的厚度是轨道上雪平均厚度的两倍。通过这种类似，在单循环中被时钟策略替换的页的号码是被随机替换的页的号码的两倍。这个近似不是最完美的，因为时钟指针并不是以一个确定的速率移动，但是直观意义还是有的。

8.14 在 S/370 体系结构中，存储关键字是与实存中每个页帧相关联的控制字段。这个关键字中与页替换有关的有两位：访问位和修改位。当在帧中的任何单元执行写操作时，修改位被置为 1；当一个新页被装入到该帧中时，访问位被置为 1。请给出一种方法，仅仅使用访问位来确定哪个页帧是最近最少使用的。

解答

处理器硬件置访问位为 0 当一个新页被加入到帧时，置为 1 当这个页帧的位置被访问到时。操作系统可以维护几个页帧表队列，一个页帧表项从一个队列移动到另一个队列取决于这个页帧的访问位被值为零的时间长短。当必须有页被替换时，被替换的页将从具有最长未被访问时间的页帧队列中选取。

8.15 考虑如下的页访问序列（序列中的每一个元素都是页号）：

12345213323454511325

定义经过 k 次访问后平均工作集大小为 $S_k(\Delta) = 1/k \sum W(t, \Delta) (t=1-k)$, 并且定义经过 k 次访问后错过页的概率为 $M_k(\Delta) = 1/k \sum F(t, \Delta) (t=1-k)$, 其中如果页错误发生在虚拟时间 t , 则 $F(t, \Delta) = 1$, 否则 $F(t, \Delta) = 0$ 。

a. 当 $\Delta=1, 2, 3, 4, 5, 6$ 时, 绘制与图 8.19 类似的图标来说明刚定义的页访问序列的工作集。

b. 写出 $S_{20}(\Delta)$ 关于 Δ 的表达式。

c. 写出 $M_{20}(\Delta)$ 关于 Δ 的表达式。

解答

a.

Seq of page refs	Window Size, Δ					
	1	2	3	4	5	6
1	1	1	1	1	1	1
2	2	12	12	12	12	12
3	3	23	123	123	123	123
4	4	34	234	1234	1234	1234
5	5	45	345	2345	12345	12345
2	2	52	452	3452	3452	13452
1	1	21	521	4521	34521	34521
3	3	13	213	5213	45213	45213
3	3	3	13	213	5213	45213
2	2	32	32	132	132	5132
3	3	23	23	23	123	123
4	4	34	234	234	234	1234
5	5	45	345	2345	2345	2345
4	4	54	54	354	2354	2354
5	5	45	45	45	345	2345
1	1	51	451	451	451	3451
1	1	1	51	451	451	451
3	3	13	13	513	4513	4513
2	2	32	132	132	5132	45132
5	5	25	325	1325	1325	1325

b.c.

Δ	1	2	3	4	5	6
$s_{20}(\Delta)$	1	1.85	2.5	3.1	3.55	3.9
$m_{20}(\Delta)$	0.9	0.75	0.75	0.65	0.55	0.5

S_{20} 是一个 Δ 的增函数, M_{20} 是一个 Δ 的非增函数。

8.16 VSWS 驻留集管理策略的性能关键是 Q 的值。经验表明, 如果对于一个进程使用固定的 Q 值, 则在不同的执行阶段, 页错误发生的频率有很大的差别。此外对不同的进程使用相同的 Q 值, 则发生页错误的频率会完全不同。这些差别表明, 如果有一种机制可以在一个进程的生命周期中动态的调整 Q 得知, 则会提高算法的性能。请基于这种目标设计一种简单的机制。

解答

[PIZZ89] 推荐使用如下策略。在窗口中使用一个机构在窗口时间调整 Q 的值作为实际页错误率的函数, 页错误率被计算出并且同作为系统值的作业理想页错误率比较。 Q 的值被上调(下调)当实际的页错误率比理想值高(低)。使用这种调整机制的实验证明可以动态调整 Q 值的测试作业在每次运行时产生较少的页错误和减小的驻留集, 相比于固定 Q 值的作业的运行

(在很大程度上)。存储时间在相对于 Q 值使用可调整机制时也会产生一个固定且可观的改进，比较于使用固定的 Q 值。

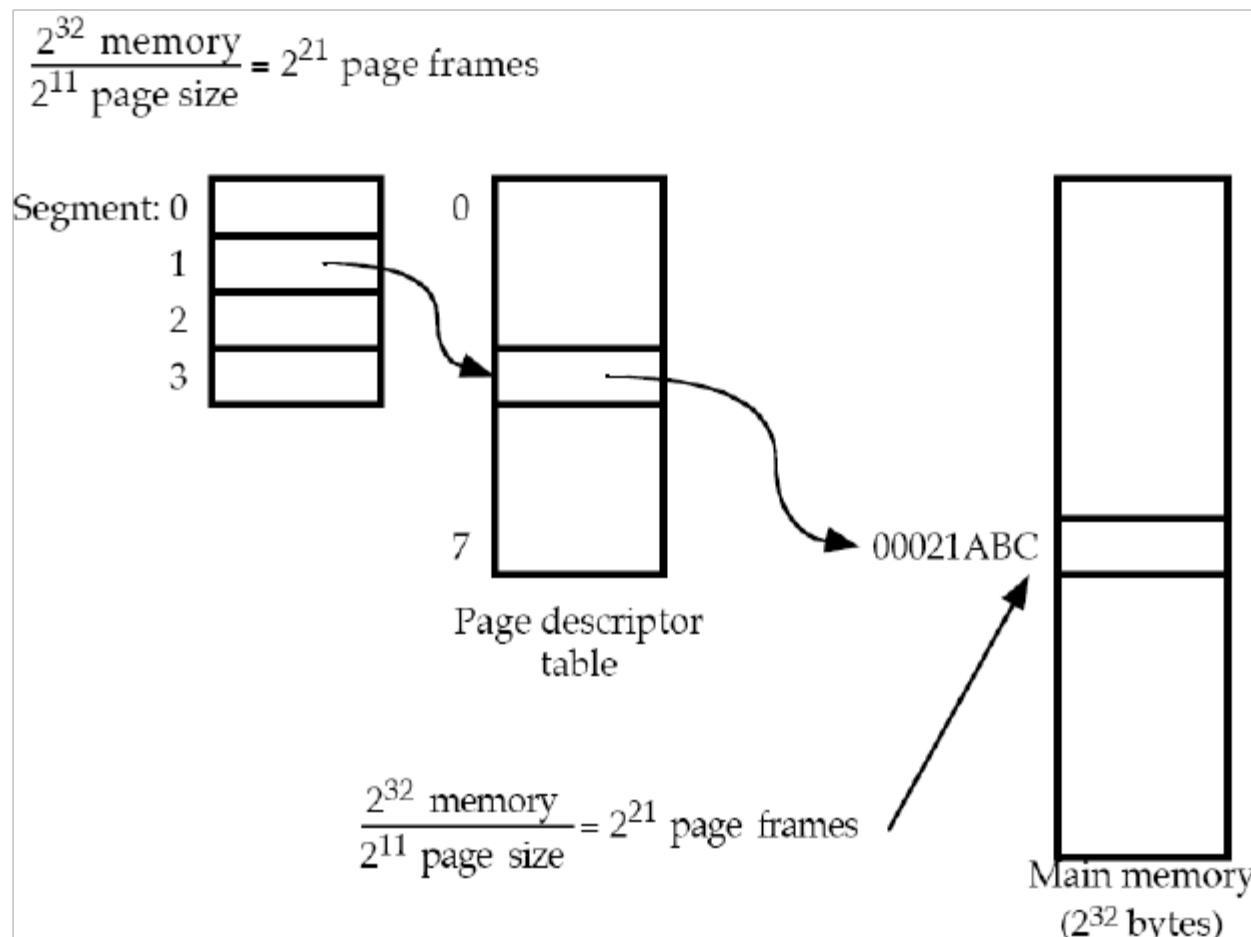
8.17 假设一个任务被划分为 4 个大小相等的段，并且系统为每个段建立了一个有 8 项的页描述符表。因此，该系统是分段与分页的组合。假设页尺寸为 2KB。

a. 每段的最大尺寸为多少？

b. 该任务的逻辑地址空间最大为多少？

c. 假设该任务访问到物理单元 00021ABC 中的一个元素，那么为它产生的逻辑地址的格式是什么？该系统的物理地址最大为多少？

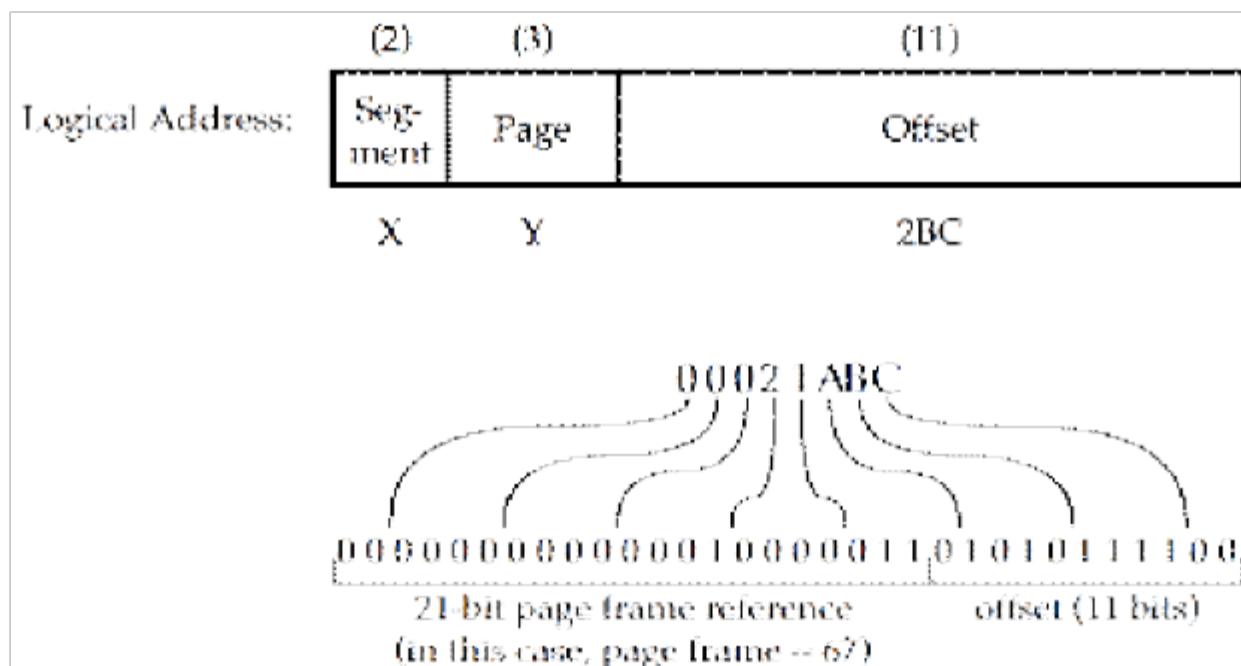
解答



$$a. 8 \times 2K = 16k$$

$$b. 16K \times 4 = 64K$$

$$c. 2^{32} = 4G\text{Bytes}$$



8.18 考虑一个分页式的逻辑地址空间（由 32 个 2KB 的页组成），将它映射到一个 1MB 的物理内存空间。

- 该处理器的逻辑地址空间格式是什么？
- 页表的长度和宽度是多少（忽略“访问权限”位）？
- 如果物理内存空间减少了一半，则会对页表有什么影响？

解答

a.

页码 (5)	偏移量 (11)
--------	----------

b. 32 个页表项长，每个页表项 9 个 bits 宽

c. 如果页表项还是 32 个且页大小不变，那么每个页表项变为 8bits 宽

8.19 UNIX 内核可以在需要时动态地在虚存中增加一个进程的栈，但却从不缩小这个栈。考虑下面的例子：一个程序调用一个 C 语言程序，这个子程序在栈中分配一个本地数组，一共需要 10KB 大小，内核扩展这个栈段来适应它。当这个子程序返回时，内核应该调整栈指针并释放空间，但它却未被释放。解释为什么可以缩小栈以及 UNIX 内核为什么没有缩小栈。

解答

可以通过释放再分配不使用的页来缩减栈的大小。按照惯例，在超出当前栈顶范围内内存中的值没有定义。在全部的体系结构中，标志栈顶部的指针在一个定义完好的记录中。因此可以读取栈的内容且按要求释放不使用的页。不缩小栈的原因是这样做的效果很小。如果用户程序重复调用子程序需要附加的空间来分配给局部变量（一个很可能的情况），然后许多时间会被浪费在释放重分配栈空间

第 9 章 单处理器调度

9.1 考虑下面的进程集合：

进程名	到达时间	处理时间
A	0	3
B	1	5
C	3	2
D	9	5
E	12	5

对这个集合，给出类似于表 9.5 和图 9.5 的分析。

每格代表一个时间单位，方框中的数表示当前运行的进程

A	A	A	B	B	B	B	C	C	D	D	D	D	E	E	E	E
A	B	A	B	C	A	B	C	B	D	B	D	E	D	E	D	E
A	A	A	B	B	B	C	C	B	D	D	E	D	E	E	E	D
A	A	A	C	C	B	B	B	B	D	D	D	D	E	E	E	E
A	A	A	C	C	B	B	B	B	D	D	D	D	E	E	E	E
A	A	A	B	B	B	B	C	C	D	D	D	D	E	E	E	E
A	B	A	C	B	A	B	B	D	B	D	E	D	E	D	E	E
A	B	A	A	C	B	B	C	B	D	D	D	D	E	E	D	E

第一到第八行依次是FCFS RR, q=1 RR, q=4 SPN SRT HRRN

Feedback, q=1 Feedback, q=2 (i)

		A	B	C	D	E
FCFS	Ta	0	1	3	9	12
	Ts	3	5	2	5	5
	Tf	3	8	10	15	20
	Tr	3.00	7.00	7.00	6.00	8.00 6.20
RR q = 1	Tr/Ts	1.00	1.40	3.50	1.20	1.60 1.74
	Tf	6.00	11.00	8.00	18.00	20.00
	Tr	6.00	10.00	5.00 9.00	8.00	7.60
RR q = 4	Tr/Ts	2.00	2.00	2.50	1.80	1.60 1.98
	Tf	3.00	10.00	9.00	19.00	20.00
	Tr	3.00	9.00	6.00	10.00	8.00 7.20
SPN	Tr/Ts	1.00	1.80	3.00	2.00	1.60 1.88
	Tf	3.00	10.00	5.00	15.00	20.00
	Tr	3.00	9.00	2.00	6.00	8.00 5.60
SRT	Tr/Ts	1.00	1.80	1.00	1.20	1.60 1.32
	Tf	3.00	10.00	5.00	15.00	20.00
	Tr	3.00	9.00	2.00	6.00	8.00 5.60
HRRN	Tr/Ts	1.00	1.80	1.00	1.20	1.60 1.32
	Tf	3.00	8.00	10.00	15.00	20.00
	Tr	3.00	7.00	7.00	6.00	8.00 6.20
FB q = 1	Tr/Ts	1.00	1.40	3.50	1.20	1.60 1.74
	Tf	7.00	11.00	6.00	18.00	20.00
	Tr	7.00	10.00	3.00	9.00	8.00 7.40
FB q = 2i	Tr/Ts	2.33	2.00	1.50	1.80	1.60 1.85
	Tf	4.00	10.00	8.00	18.00	20.00
	Tr	4.00	9.00	5.00	9.00	8.00 7.00
	Tr/Ts	1.33	1.80	2.50	1.80	1.60 1.81

9.2 对下面的集合重复习题9.1 的要求

进程	到达时	处理时间
A	0	1
B	1	9
C	2	1
D	3	9

A	B	B	B	B	B	B	B	B	C	D	D	D	D	E	E	E
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

A	B	C	B	D	B	D	B	D	B	D	B	D	B	D	D	E	D	E	E
A	B	B	B	B	C	B	B	B	B	D	D	D	D	B	E	E	E	D	E
A	B	B	B	B	B	B	B	B	B	C	D	D	D	D	E	E	E	E	E
A	B	C	B	B	B	B	B	B	B	B	D	D	D	D	E	E	E	E	E
A	B	B	B	B	B	B	B	B	B	C	D	D	D	D	E	E	E	E	E
A	B	C	D	B	D	B	D	B	D	B	D	B	D	B	D	E	D	E	E
A	B	C	D	B	B	D	D	B	B	B	B	D	D	D	E	E	D	E	E

		A	B	C	D
FCFS	Ta	0	1	2	3
	Ts	1	9	1	9
	Tf	1.00	10.00	11.00	20.00
RRq=1	Tr	1.00	9.00	9.00	17.00 9.00
	Tr/Ts	1.00	1.00	9.00	1.89 3.22
	Tf	1.00	18.00	3.00	20.00
RRq=4	Tr	1.00	17.00	1.00	17.00 9.00
	Tr/Ts	1.00	1.89	1.00	1.89 1.44
	Tf	1.00	15.00	6.00	20.00
SPN	Tr	1.00	14.00	4.00	17.00 9.00
	Tr/Ts	1.00	1.56	4.00	1.89 2.11
	Tf	1.00	10.00	11.00	20.00
SRT	Tr	1.00	9.00	9.00	17.00 9.00
	Tr/Ts	1.00	1.00	9.00	1.89 3.22
	Tf	1.00	11.00	3.00	20.00
HRRN	Tr	1.00	10.00	1.00	17.00 9.00
	Tr/Ts	1.00	1.00	9.00	1.89 3.22
	Tf	1.00	10.00	11.00	20.00
FBq=1	Tr	1.00	18.00	1.00	17.00 9.25
	Tr/Ts	1.00	2.00	1.00	1.89 1.47
	Tf	1.00	19.00	3.00	20.00
FB q=2i	Tr	1.00	17.00	1.00	17.00 9.00
	Tr/Ts	1.00	1.89	1.00	1.89 1.44

9.3 证明在非抢占调度算法中，对于同时到达的批处理作业，SPN 提供了最小平均等待时间。假设调度器只要有任务就必须执行。

我们能够证明在一批 n 个作业同时到达且忽略以后到来的作业。试验会延伸包含以后的作业。假设各作业到来的顺序是： $t_1 \leq t_2 \leq \dots \leq t_n$ 。假设 n 个使用者必须等到工作 1 的完成： $n-1$ 个使用者必须等到工作 2 的完成，依次继续。那么，平均反应时间是 $[n*t_1 + (n-1)*t_2 + \dots + t_n]/n$ 。如果我们在这个时间表中做些改变，例如调换工作 j 和 k ($j < k$)，平均反应时间增加，因为 $(k-j) * (t_k - t_j)/n \geq 0$ 。换句话说，如果 SPN 运算法则没被采用，平均反应时间只会增加。

9.4 假设一个进程的每一次瞬间执行时间 (burst-time 模式) 为 6, 4, 6, 4, 13, 13, 13，假设最初的猜测值为 10。请画出类似于图 9.9 的图表。

数据图如下：

平均观察	观测值	平均值	alpha=0.8	alpha=0.5
------	-----	-----	-----------	-----------

1	6	0.00	0.00	0.00
2	4	3.00	4.80	3.00
3	6	3.33	4.16	3.50
4	4	4.00	5.63	4.75
5	13	4.00	4.33	4.38
6	13	5.50	11.27	8.69
7	13	6.57	12.65	10.84

9.5 考虑下面的公式，它可以替代 $S_{n+1} = \alpha T_n + (1-\alpha)S_n, X_{n+1} = \min[Ubound, \max[Lbound, (\beta S_n + 1)]]$ 其中，Ubound 和 Lbound 是预先选择的估计值 T 的上限和下限。X_{n+1} 的值用于最短进程优先的算法，并且可以代替 S_{n+1}。 α 和 β 有什么功能，它们每个取最大值和最小值会产生什么影响？

第一个等式等同于等式 9.3，所以参数使指数起平滑作用。参数 β 是延迟变化参数。一个较小的 β 值将会时在观察次数里更快的适应改变，但估计上会产生更多的波动。一个诡辩的这种估计类型的程序收录于 Applied Optimal Estimation

9.6 在图 9.5 下面的例子中，在控制权限转移到B之前，进程 A 运行 2 个时间单元，另外一个场景是在控制权限转移到B之前，进程 A 运行 3 个时间单元。在反馈调度算法中，这两种场景的策略有什么不同？

关键要看进程 A 是否一开始就被放置在队列中。如果是，它在被抢占前就被赋予 2 个额外的时间单元。

9.7 在一个非抢占的单处理器系统中，在刚刚完成一个作业后的时间 t，就绪队列中包含三个作业。这些作业分别在时间 t₁, t₂, t₃ 处到达，估计执行时间为 r₁, r₂, r₃。图 9.18 显示了它们的响应比随着时间线形增加。使用这个例子，设计一种响应比调度的变体，称为极小极大响应比调度算法，它可以使给定的一批作业的最大响应比最小。

首先，调度器计算 t+r₁+r₂+r₃ 时刻的响应比，如果三个进程都结束，此时，进程 3 就有最高响应比，因此调度器就执行该进程，然后在 t+r₁+r₂ 时刻继续执行进程 1 和 2 直到结束。因此，进程 1 的响应比最小，其次进程 2 在 t 时刻被选中执行。这种算法在每个进程结束的时候重复一次并且考虑新加入的进程。但注意这种算法和最高响应比算法不完全相同。后者在 t 时刻会选择进程 1。直觉上看，当前算法通过不断的推迟具有最少响应比增值的进程从而减小最高响应比。

9.8 证明，对给定的一批作业，上一习题中的最大响应比调度算法能够使它们的最大响应时间最小

这个证明，来自于 P. Mondrup, is reported in [BRIN73]。考虑在时间 t 时的队列，仅仅在前一个进程结束且忽略以后工作的到来。等待执行的作业被编号从 1 到 n：

作业:	1	2	I	n
到达时间:	t ₁	t ₂	t _i	t _n
服务时间:	r ₁	r ₂	r _i	r _n

我们假设作业 i 在完成前能达到最高的响应时间比。当作业 1 到 n 都执行结束，总时间为

$$T_i = t + r_1 + r_2 + \dots + r_i, \text{ 作业 } i \text{ 的响应比为 } R_i (T_i) + (T_i - t_i) / r_i$$

执行作业 i 的原因是它的响应比将是以下作业在 T_i 时最小的：

$$R_i (T_i) = \min[R_1 (T_i), R_2 (T_i), \dots, R_i (T_i)]$$

考虑不同序列中同样的 n 个作业的排序结果：

作业:	a	b	j	z
到达时间:	t _a	t _b	t _j	t _z
服务时间:	r _a	r _b	r _j	r _z

在新的序列中，我们选择最小的后继服务作业，从 a 到 j，包括从 1 到 i 最初的后继。当作业 a 到 j 被执行结束，总时间为 T_j = t + r_a + r_b + ... + r_j

作业 j 的响应比是 R_j (T_j) + (T_j - t_j) / r_j

由于作业 1 到 j 是作业 a 到 j 的一个子集，那么总的服务时间 T_i - t 一定小于等于总的服务时间 T_j - t。又响应比随着时间增加而增加，T_i <= T_j，意味 R_j (T_j) >= R_i (T_i)

人们还知道，作业 j 是其中的之一，作业 j 在时间 T_i 有最小的响应比。上述不平等的，因此可以延长如下：R_j (T_j) >= R_j (T_i) >= R_i (T_i) 换言之，在调度算法改变后，会有作业 j 达到响应比 R_j (T_j)，它会

大于等于最高的响应比 R_i (T_i)。

这证明有效期一般为优先考虑的都是非递减函数的时间。例如，在一个FIFO制度，优先增加线性与等候时间，在同样的速度，为所有作业机会。因此，目前的证据表明，该FIFO的算法最大限度地减少候车时间，对于给定了一批作业。

9.9 驻留时间 T_r 被定义成一个进程花费在等待和被服务上的总平均时间。说明对FIFO，若平均服务时间为 T_s ，则有 $T_r = T_s / (1 - \rho)$ ，其中 ρ 为使用率。

在我们开始以前，有一个结果，这是必要的，具体情况如下。假设一个项目拥有服务时间，而实际服务了时间 h 。那么，预期的残留服务时间 $E[T/T > h] = T_s$ 那就是，无论多久，一个项目服务，预计剩余服务时间，只是平均服务时间为项目。这一结果，虽然反以直觉是正确的，正如我们现在查看，考虑到指数的概率分布函数： $F(x) = Pr[X \leq x] = 1 - e^{-\mu x}$ ，那么，我们有 $Pr[X > x] = e^{-\mu x}$ 现在让我们看看 x 多于 $x + h$ 的条件概率：

$$\Pr[X > x+h | X > x] = \frac{\Pr[(X > x+h) | (X > x)]}{\Pr[X > x]} = \frac{\Pr[X > x+h]}{\Pr[X > x]}$$

$$\Pr[X > x+h | X > x] = \frac{e^{-\mu(x+h)}}{e^{-\mu x}} = e^{-\mu h}$$

， 则 $\Pr[X \leq x+h | X > x] = 1 - e^{-\mu h} = \Pr[X \leq h]$

因此，概率分布为服务时间给予确已送达期限 x 是相同的概率分布，总服务时间。因此，预期价值的剩余服务时间是一样的原来预期值的服务时间。与这一结果，我们现在可以着手将原问题。当一个项目展开服务以来，总的响应时间，该项目将包括它自己的服务时间，再加上服务的时候，所有的物品摆在它面前的，在排队。预期总体响应时间由三部分组成。

- 预计到达进程的服务时间= T_s 。
- 预计所有目前正在等待服务的进程的服务时间。值为 $w \times T_s$ ， w 是平均等待服务时间。剩余等待服务时间，如果现在有一个进程正在执行。这个值可以表示成为 $\rho \times T_s$ ρ 是利用，因此一个进程正在服务的概率。 T_s 正如我们已证明是预计等待服务的时间。因此，我们有

$$R = T_s \times (1 + w + \rho) = T_s \times \left(1 + \frac{\rho^2}{1-\rho} + \rho\right) = \frac{T_s}{1-\rho}$$

9.10 一个处理器被就绪队列中的所有进程以无限的速度多路复用，且没有任何额外的开销（这是一个在就绪进程中使用轮转调度的理想模型，时间片相对于平均服务时间非常小）。说明对来自一个无限源的泊松输入和指数服务时间，一个进程的平均响应时间 R_x 和服务时间 x 由式 $R_x = x / (1 - \rho)$ 给出。

让我们把时间片在轮转调度中记为 δ 。在这个问题中， δ 相对于服务时间很小。现在，考虑一个新来的进程，它排在就绪队列的最后。我们假设这一特定的进程的服务时间为 x ，是 δ 的一些倍数： $x = m\delta$ 。首先，让我们回答一个问题，多少时间花费在进程得到服务前的等待队列中。它必须等待所有排在它之前的 q 个进程结束服务后，从而初步等待时间= $q \cdot \delta$ 。 Q 则是等待队列中的平均进程数目。现在，我们可以计算总时间花费在收到 x 秒钟的服务之前的等待时间。因为它必须经过队列 m 次，而每一次他们等待 $q \cdot \delta$ 秒，其总等待时间分列如下：

$$\begin{aligned} \text{Wait time} &= m(q\delta) \\ &= \frac{x}{\delta}(q\delta) \\ &= qx \end{aligned}$$

然后，响应时间是指等待时间和总服务时间 $R_x = \text{wait time} + \text{service time} = qx + x = (q + 1)x$ 。谈到排队的公式，在附录A，是指物品的数量，该系统的 Q 可表示为：

$$Q = \rho / (1 - \rho)$$

9.11 大多数轮转调度器使用大小固定的时间片。给定支持小时间片的参数。现在给定一个支持大的时间片的参数。比较并对比两个参数所适用的系统和作业的类型。存在两种参数都合理的情况吗？

一个论点主张一个小时片：用一个小时片会加强反应能力，多次运行的所有进程进行了短暂的时间片。当就绪队列有许多流程，这是互动性，反应能力是非常重要的如：一般用途的计算机系统。

一个论点主张一个大的时间片：利用大型公司将提高吞吐量和CPU利用率测量方面的实际工作，因为那里是不足的背景下切换，从而减少开销。

一个系统，其中既可能是适当的：有一些制度，即无论是中小企业还是大广，是合理的。例如：很长的行业，需要在短短数用户互动。虽然这种类型的工作可以被视为一个批处理工作，在某种意义上来说，它仍然需要与用户手中。因此，在时代的时候，有没有使用者之间的相互作用，时间片可能会增加优化，并在整个过程中的互动时间，时间片可能降至提供更好的应变能力。

9.12 在排队系统中，新作业在得到服务之前必须等待一会儿。当一个作业等待时，它的优先级从0开始以速度 α 线形增加。一个作业一直等待到它的优先级达到正在运行中的作业的优先级，然后它开始与其他正在运行的作业使用轮转法平等地共享处理器，与此同时，它的优先级继续以比较慢的速度 β 增长。这个算法称为自私轮转法，因为在运行中的作业试图通过不断地增加它的优先级来垄断处理器。

首先，我们需要厘清意义的参数 λ' 。速度在项目到达第一盒（"队列"专栏）是 λ 。两个相邻的来港人士，以第二个选项（"服务"的框），将得出一个稍微减慢，因为第二项是延迟其追逐的第一个项目。我们可以计算出垂直偏移 Y 的数字，在两种不同方式的基础上，几何形状的图表：

9.13 一个使用轮转调度和交换的交互式系统，试图按如下方式对普通的请求给出有保证的响应：在所有就绪进程中完成一次轮转循环后，系统通过用最大响应时间除以需要服务的进程数目，确定在下一个循环中分配给每个就绪进程的时间片。请问是否是合理的解释？

只是，只要有相对很少有用户在该系统内。当时间片变小，以满足更多用户迅速两件事情发生：（1）处理器利用率下降，以及（2）在某一个点，时间片变得太小，以满足最琐碎的请求。用户将经历一个突然增加的响应时间，因为他们的要求，必须经过轮转调度好几次。

9.14 多级反馈排队调度对哪种类型的进程有利，是受处理器限制的进程还是受I/O限制的进程？请简要说明原因。

如果一个进程占用过多处理器时间，它将会被移到一个低优先级的队列中。这会使受I/O限制的进程留在优先级高的队列中。

9.15 在基于优先级的进程调度中，如果当前没有其他优先级更高的进程处于就绪状态，调度器将把控制给一个特定的进程。假设在进行进程调度决策时没有使用其他信息，还假设进程的优先级是在进程被创建时建立的，并且不会改变。在这样的系统中，为什么使用Dekker方法解决互斥问题是“危险”的？通过说明会发生什么和如何发生来解释该问题。

Dekker的算法依靠的是对公平性的硬件和操作系统。使用的优先次序的风险饥饿如下。可能有这样的情况，如果出生是一种很快速的反复过程，因为它不断地发现Flag[1]=虚假的，不断进入其关键路段，而在P1，离开内部回路，它正在等待它反过来又可以不设置Flag[1]为真实，阻止这样做出生的读变量（记得进入该变发生下相互排斥）

9.16 5个批作业，从A到E，同时到达计算机中心。它们的估计运行时间为15, 9, 3, 6和12分钟，它们的优先级（外部定义）分别为6, 3, 7, 9和4（值越小，表示的优先级越高）。对下面的每种调度算法，确定每个进程的周转时间和所有作业的平均周转时间（忽略进程切换的开销），并解释是如何得到这个结果的。对于最后三种情况，假设一次只有一个作业运行直到它结束，并且所有作业都是受处理器限制的。

- a. 时间片为1分钟的轮转法。
- b. 优先级调度
- c. FCFS（按15, 9, 3, 6和12顺序运行）。
- d. 最短作业优先

a: 时间片为1分钟的轮转法：

	1	2	3	4	5	Elapsed time
A	B	C	D	E	5	
A	B	C	D	E	10	
A	B	C	D	E	15	
A	B		D	E	19	

A	B	D	E	23
A	B	D	E	27
A	B		E	30
A	B		E	33
A	B		E	36
A			E	38
A			E	40
A			E	42
A				43
A				45

每个进程的周转时间

$A=45 \text{ min}$, $B=35 \text{ min}$, $C=13 \text{ min}$, $D=26 \text{ min}$, $E=42 \text{ min}$

平均周转时间是 $(45+35+13+26+42)/5=32.2 \text{ min}$

b.

Priority	Job	Turnaround Time
3	B	9
4	E	$9 + 12 = 21$
6	A	$21 + 15 = 36$
7	C	$36 + 3 = 39$
9	D	$39 + 6 = 45$

平均周转时间是 $(9+21+36+39+45)/5=30 \text{ min}$

c.

Job	Turnaround Time
A	15
B	$15 + 9 = 24$
C	$24 + 3 = 27$
D	$27 + 6 = 33$
E	$33 + 12 = 45$

平均周转时间是 $(15+24+27+33+45) / 5 = 28.8 \text{ min}$

d.

Running Time	Job	Turnaround Time
3	C	3
6	D	$3 + 6 = 9$
9	B	$9 + 9 = 18$
12	E	$18 + 12 = 30$
15	A	$30 + 15 = 45$

平均周转时间是: $(3+9+18+30+45) / 5 = 21 \text{ min}$

第 10 章 多处理器和实时调度

10.1 考虑一组周期任务(3个),表10.5给了它们的执行简表。按照类似与图10.5的形式,给出关于这组任务的调度图。

表 10.5 习题 10.1 的执行简表

进程	到达时间	执行时间	完成最后期限
----	------	------	--------

A(1)	0	10	20
A(2)	20	10	40
.	.	.	.
B(1)	0	10	50
B(2)	50	10	100
.	.	.	.
C(1)	0	15	50
C(2)	50	15	100
.	.	.	.

答：对于固定的优先级来说，我们以优先级是ABC来考虑这道题。每一方格代表五个时钟单元，方格里的字母是指现在正在运行的进程。第一行是固定的优先级；第二行表示的是使用完成最后期限的最早最后期限调度。表格如下：

A	A	B	B	A	A	C	C	A	A	B	B	A	A	C	C	A	A		
A	A	B	B	A	C	C	A	C	A	A	B	B	A	A	C	C	A	A	

对于固定优先级调度来说，进程C总是错过它的最后期限。

10.2 考虑一组非周期性任务（5个），表10.6给出了它们的执行简表。按照类似于图10.6的形式给出关于这组任务的调度图。

表10.6 习题10.2的执行简表

进程	到达时间	执行时间	启动最后期限
A	10	20	100
B	20	20	30
C	40	20	60
D	50	20	80
E	60	20	70

答：每一方格代表10个时间单元。

最早期限		A	A		C	C	E	E	D	D		
有自愿空闲时间的最早期限			B	B	C	C	E	E	D	D	A	A
先来先服		A	A		C	C	D	D				

10.3 这个习题用于说明对于速率单调调度，式(10.2)是成功调度的充分条件，但它并不是必要条件也就是说，有些时候，尽管不满足式(10.2)也可能成功调度]。

a. 考虑一个任务集，它包括以下独立的周期任务：

任务P1: C1=20; T1=100

任务P2: C2=30; T2=145

使用速率单调调度，这些任务可以成功地调度吗？

b. 现在再往集合里增加以下任务：

任务P3: C3=68; T3=150

式(10.2)可以满足吗？

c. 假设前述的三个任务的第一个实例在t=0是到达，并假设每个任务的第一个最后期限如下：

D1=100; D2=145; D3=150

如果使用速率单调调度，请问这三个最后期限都能得到满足吗？每个任务循环的最后想、期限是多少？

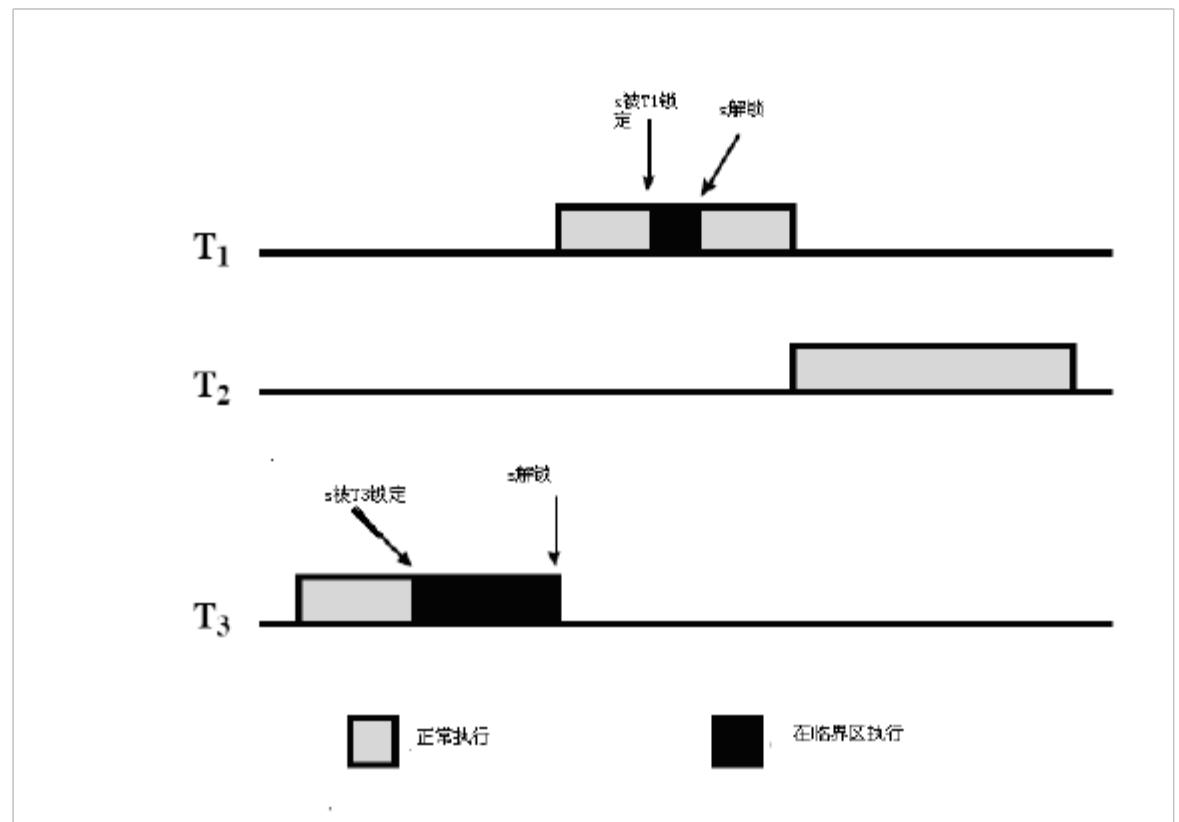
答：a. P1, P2的总使用率是0.41，小于由方程10.2给出的对于两个任务的界限0.828，因此这两个任务是

可以成功调度的。

b. 所有任务的使用率是 0.86，已经超过界限 0.779。

c. 可以观察到在 P3 执行前 P1, P2 必须至少执行一次。因此 P3 的第一次瞬间完成时间不低于 $20 + 30 + 68 = 118$ 。但是 P1 在一附加的时间区间 $(0, 118)$ 内初始化。因此 P3 直到 $118 + 20 = 138$ 才完成他的第一次执行。在 P3 的期限内。继续这个过程，我们可以知道，这三个任务的所有期限都能实现。

10.4 画一个与图 10.9 (b) 相似的图，用来显示队同一个例子使用优先级置顶的事件顺序。



一旦 T3 进入他的临界区，他的优先级就被指定为高于 T1。当 T3 离开他的临界区时，他被 T1 抢先。

第 11 章 I/O 管理和磁盘调度

11.1 考虑一个程序访问一个 I/O 设备，并比较无缓冲的 I/O 和使用缓冲区的 I/O。说明使用缓冲区最多可以减少 2 倍的运行时间。

如果计算的时间正好等于它的 I/O 时间（它是最佳环境），操作者和外围设备同时运行。如果单独运行，只要花费他们的一半时间，设 C 是整个程序的计算时间，T 为所要求总的 I/O 时间，因而寄存器最好的运行时间是 $\max(C, T)$ ，不需要寄存器的运行时间是 $C+T$ ，

显然 $(C+T)/2 \leq \max(C, T) \leq (C+T)$ 。

11.2 把习题 11.1 的结论推广到访问 n 个设备的程序中。

最佳比是 $(n+1) : n$

11.3 使用与表 11.2 类似的方式，分析下列磁道请求：27, 129, 110, 186, 147, 41, 10, 64, 120。假设磁头最初定位在磁道 100 处，并且沿着磁道号减小的方向移动。假设磁头沿着磁道增大的方向移动，请给出同样的分析。

FIFO	SSTF	SCAN	C-SCAN
------	------	------	--------

下一个被访问的磁道	横跨的磁道数	下一个被访问的磁道	横跨的磁道数	下一个被访问的磁道	横跨的磁道数	下一个被访问的磁道	横跨的磁道数
27	102	110	10	64	23	64	23
129	19	120	9	41	14	41	14
110	76	129	18	27	17	27	17
186	39	147	39	10	100	10	176
147	106	186	122	110	10	186	39
41	31	64	23	120	9	147	18
10	54	41	14	129	18	129	9
64	56	27	17	147	39	120	10
120	61.8	10	29.1	186	29.6	110	38
平均寻道长度		平均寻道长度		平均寻道长度		平均寻道长度	

如果磁头沿着增大的方向，只有SCAN 和 C-SCAN 的结果有变化

SCAN		C-SCAN	
下一个被访问的磁道	横跨的磁道数	下一个被访问的磁道	横跨的磁道数
10	10	10	10
110	10	110	10
120	9	120	9
129	18	129	18
147	39	147	39
186	122	186	176
64	23	10	17
41	14	27	14
27	17	41	23
10	29.1	64	35.1
平均寻道长度		平均寻道长度	

1.4 考虑一个磁盘，有N个磁道，磁道号从0到(N-1)，并且假设请求的扇区随机地均匀分布在磁盘上。现在要计算一次寻道平均跨越的磁道数。

- 首先，计算当磁头当前位于磁道t时，寻道长度为j的可能性。提示：这是一个关于确定所有组合数目问题，所有磁道位置作为寻道目标的可能性是相等的。
- 接下来计算寻道长度为K的可能性。提示：这包括所有移动了K个磁道的可能性之和。
- 使用下面计算期望值得公式，计算一次寻道平均跨越的磁道数目：

$$E[X] = \sum_{i=0}^{N-1} i \sum_{t=0}^{N-1} \Pr[x=i]$$

d. 说明当N比较大时，一次寻道平均跨越的磁道数接近N/3.

- 设P[j/t]表示位于磁道t，寻道长度为j的概率，知随机访问一个任何一个磁道的可能性为相等为1/N，因此我们有P[j/t]=1/N, t<=j-1 或者 t>=N-j; P[j/t]=2/N, j-1 前一种情况下，当前磁道接近于磁盘的两端。因此只有一个相距j长度的磁道，故为2/N。
- 令P[k]= $\sum P[k/t]*P[t]=1/N \sum P[k/t]$, 由(a)可知，取值1/N的有2k个磁道，取值为2/N有

(N-k) 个,

所以有

$$P[k] = (2k/N + 2(N-k)/N) / N = 2(N-k)/N \times N$$

$$(c) E[k] = \sum k * P[k] = \sum 2k(N-k) / N \times N$$

$$= (N \times N - 1) / 3N$$

(d) 当 N 比较大时, 从上文可以看出一次寻道平均跨越磁道数接近 N/3

11.5 下面的公式适用于高速缓冲存储器和磁盘高速缓存:

$$T_s = T_c + M \times T_d$$

请把这个公式推广到 N 级存储器结构, 而不是仅仅 2 级。

定义:

A_i=从 i 级存储器找到信息的时间;

H_i=消息在第 i 级存储器并且没有在更高级存储器的概率;

B_i=从第 (i+1) 级向第 i 级传送一块数据的时间。

假设缓存在 1 级存储上, 主存在 2 级存储上, 如此下去, 形成一个 N 级存储结构, 因此有

$$T_s = \sum A_i H_i$$

若消息在 M₁ 层, 可以立即被读, 如果在 M₂ 中, 不在 M₁ 中, 那么这块数据从 M₂ 传到 M₁ 中再读。

因此 A₂=B₁+A₁

进而有 A₃=B₂+A₂=B₁+B₂+A₁

即有 A_i=A₁+ $\sum B_j$

所以 T_s=T₁ $\sum H_i + \sum \sum B_j H_i$

因为 $\sum H_i = 1$

最后可得 T_s=T₁+ $\sum \sum B_j H_i$

11.6 对基于频率的替换算法 (见图 11.12), 定义 F_{new}, F_{middle} 和 F_{old} 分别为包含新区, 中间区和的高速缓存片段, 显然 F_{new}+F_{middle}+F_{old}=1 如果有

a. F_{old}=1—F_{new}

b. F_{old}=1/(高速缓存大小)

请分别描述该策略。

图 11.11 的中间区是空的, 因此这种策略退化为图 11.11a 的策略。

老区由一块组成, 并且我们有 LRU 替换策略。

11.7 对于一个有 9 个磁道的磁带, 磁带速度为 120 英寸每秒, 磁带密度为 1600 线位/英寸, 请问它的传送率为多少?

密度可表示为 1600 线位每英寸, 因此传送速率为 $1600 \times 1200 = 192000$ 线位每秒。

11.8 假设有一个 2400 英寸的磁带盘, 记录间的间隙为 0.6 英寸, 这个间隙是磁带在读操作之间的停止; 在间隙期间磁带速度成线性增加或减小, 磁带的其他与习题 11.7 相同。磁带上的数据按物理记录组织, 每个物理记录包含固定数目的由用户定义的单元, 称为逻辑记录。

a. 在磁带上读取分装在 10 个物理记录中的 120 个逻辑记录需要多少时间?

b. 同样。如果是分装在 30 个物理记录中, 则需要多少时间?

c. 对于上述每种分块方案, 整个磁带分别可以保存多少个逻辑记录?

d. 对于上述每种分块方案, 有效的总速率分别是多少?

e. 磁带的容量是多少?

假设每个记录由 30 块组成。

b. 我们先定义从一个物理块加间隙到了另一块的读取时间

物理块的大小 = (30 个逻辑记录每物理记录) \times (120 比特每逻辑记录)

=3600 字节

物理块的长度 = 3600 字节 / (1600 比特/英寸) = 2.35 英寸

间隙的长度 = 0.6 英寸

传输一个块加间隙的传输时间= $2.25/120 + 0.6/60 = 0.02875$ 秒

磁带上块的数目= $(2400 \times 12) / (2.25 + 0.6) = 10105$ 物理块

因此，读取时间为 $10105 \times 0.02875 = 291$ 秒

c. 如果分装在 30 个物理记录中，磁带包含 10105 个物理记录和 $30 \times 10105 = 303150$ 个逻辑记录。

d. 分装在 30 个物理记录中的有效传输速率：

$$R = (303150 \times 120) / 291 = 125010 \text{ 字节/秒}$$

e. 容量= $303150 \times 120 = 36378000$ 字节

11.9 如果磁盘中扇区大小固定为每扇区为 512 字节，并且每磁道 96 个磁区，每面 110 个磁道，一共有 8 个可用的面，对于习题 11.8 (b)，计算存储这些逻辑记录需要多少磁盘空间（扇区、磁道和面）。忽略文件头记录和磁道索引，并假设记录不能跨越两个扇区。

每个扇区能容纳 4 个记录，所需扇区数= $303150 / 4 = 75788$

所需磁道数= $75788 / 96 = 790$

所需面数= $790 / 110 = 8$

11.10 考虑习题 11.9 所描述的磁盘系统，假设该磁盘的旋转速度为 360r/m。一个处理器使用中断驱动 I/O 从磁盘中读取一个扇区，每个字节一个中断。如果处理每个中断需要 2.5us，处理器花费在处理 I/O 上的时间占多少百分比（忽略寻道时间）？

每扇区 512 字节，每字节一个中断，所以每扇区 512 个中断。

中断总时间= $2.5 \times 512 = 1280$ us。

每个扇区读取时间= $60 \text{ s/m} \times 360 \text{ r/m} \times 96 \text{ 扇区/磁道} = 1736$ us

处理器花费在处理 I/O 上的时间百分比= $100 \times 1280 / 1736 = 74\%$

11.11 如果使用 DMA 策略并假设每个扇区一个中断，重做习题 11.10。

使用 DMA 策略，中断总时间= 2.5us

处理器花费在处理 I/O 上的时间百分比= $100 \times 2.5 / 1736 = 0.14\%$

11.12 一个 32 位计算机有两个选择通道和一个多路通道，每个选择通道支持两个磁盘和两个磁带部件。多路通道有两个行式打印机、两个卡片阅读机，并连接着 10 个 VDT 终端。假设有以下的传送率：

磁盘驱动器	800KB/s
磁带驱动器	200KB/s
行式打印机	6.6KB/s
卡片阅读机	1.2KB/s
VDT	1KB/s

系统中的最大合计传送率为多少？

每次只有一个驱动设备能在选择通道上得到服务，

因此，最大速率= $800 + 800 + 2 \times 6.6 + 2 \times 1.2 + 10 \times 1 = 1625.6$ KB/s

11.13 当条带大小比 I/O 大小时，磁盘条带化显然可以提高数据传送率。同样，相对于单个的大磁盘，由于 RAID 0 可以并行处理多个 I/O 请求，显然它可以提高性能。但是，相对于后一种情况，磁盘条带化还有必要存在吗？也就是说，相对于没有条带化的磁盘阵列，磁盘条带化可以提高 I/O 请求速度的性能吗？

这取决于 I/O 请求类型。对于一种极端情况，如每次只有一个进程有一个大 I/O 请求时，磁盘条带化可以提高性能。但如果有很多进程有许多小的 I/O 请求时，相对于 RAID 0 没有条带化的磁盘阵列可以提高性能。

第12章 文件管理

12.1、定义：B=块大小 R=记录大小 P=块指针大小 F=组块因子，即一个块中期望的记录数。对图 12.6 中描述的三种组块方法分别给出关于 F 的公式。

答案：

$$\text{固 定 组 块 : } F = \text{最大整数} \leq \frac{B}{R}$$

当一个可变长度记录被保存到组块中的时候，组块中会增加一个标记着记录边界的数据，用来标识记录。当跨越式记录桥联块边界的时候，需要用到一些关联着后继组块的结构。一种可能情况是在每个记录前加一个长度标识。另一种可能情况是在两个记录之间加一个特殊的区分标识。因此，我们假设每一个记录需要一个标识，并且标识大小约等于块指针大小。对于跨越式组块，指向它下一个组块的大小为的块指针被包含在每一个组块中，所以跨越式记录可以很容易地被重定位。由此可知：

$$\text{可变组块跨越式: } F = \frac{B - P}{R + P}$$

由于不采用跨越的方式，可变长度非跨越式组块会导致平均 $R/2$ 的空间浪费，但不需要指向后继组块的指针：

$$F = \frac{B - \frac{R}{2}}{R + P}$$

12.2、一种避免预分配中的浪费和缺乏邻近性问题的方案是，分配区的大小随着文件的增长而增加。例如，开始时，分区的大小为一块，在以后每次分配时，分区的大小翻倍。考虑一个有 n 条记录的文件，组块因子为 F ，假设一个简单的一级索引用做一个文件分配表。

- a. 给出文件分配表中入口数的上限（用关于 F 和 n 的函数表示）。
- b. 在任何时候，已分配的文件空间中，未被使用的空间的最大量是多少？

答案：a.

$$\log_2 \frac{N}{F}$$

b. 未被使用的空间总是小于已分配文件空间。

12.3、当数据

- a. 很少修改并且以随机顺序频繁地访问时，
- b. 频繁地修改并且相对频繁地访问文件整体时，
- c. 频繁地修改并以随机顺序频繁地访问时，

从访问速度、存储空间的使用和易于更新（添加 / 删除 / 修改）这几方面考虑，为了达到最大效率，你将选择哪种文件组织？

答案：a. 索引文件

b. 索引顺序文件

c. 索引文件或散列文件

12.4、目录可以当做一种只能通过受限方式访问的“特殊文件”实现，也可以当做普通文件实现。这两种方式分别有哪些优点和缺点？

答案：很明显地，如果操作系统把目录当做一种通过受限方式访问的“特殊文件”实现，安全性更容易被加强。把目录当做一种通过受限方式访问的普通文件实现使得操作系统更统一地管理对象，更容易地创建和管理用户目录。

12.5、一些操作系统具有一个树结构的文件系统，但是把树的深度限制到某个比较小的级数上。这种限制对用户有什么影响？它是如何简化文件系统的设计的（如果能简化）？

答案：这是一个少见的专题。如果操作系统构造一个文件系统以便子目录被允许包含在一个主目录底下，那么就很少或没有额外的逻辑被要求允许包含任意深度的子目录。限制子目录树的深度造成对用户组织文件空间不必要的限制。

12.6、考虑一个层次文件系统，空闲的磁盘空间保留在一个空闲空间表中。

a. 假设指向空闲空间的指针丢失了。该系统可以重构空闲空间表吗？

b. 给出一种方案，确保即使出现了一次存储失败，指针也不会丢失。

答案：a. 可以重构。使用的方法与许多 LISP 的垃圾收集系统用的方法非常相似。首先，我们将建立一种数据结构，代表磁盘的每一块，并且这个磁盘支持一种文件系统。在这里某种映射是比较合适的。然后，我们从这个文件系统的根目录开始，通过文件系统的递归下降寻找，我们标记每一块已被文件使用的磁盘块。当完成的时候，我们将为没有被使用的磁盘块建立一个空闲列表。这实质上就是NIX 命令 fsck 的功能。

b. 在磁盘上一个或多个地方备份空闲空间列表指针。无论何时列表的开端发生变化，备份指针也同样更新。这样将会保证即使发生了存储器或者磁盘块错误，你也总是能找到一个有效的指针值。

12.7、考虑由一个索引节点所表示的UNIX 文件的组织（见图12.13）。假设有 12 个直接块指针，在每个索引节点中有一个一级、二级和三级间接指针。此外，假设系统块大小和磁盘扇面大小都是 8K。如果磁盘块指针是 32 位，其中 8 位用于标识物理磁盘，24 位用于标识物理块，那么

a. 该系统支持的最大文件大小是多少？

b. 该系统支持的最大文件系统分区是多少？

c. 假设主存中除了文件索引节点外没有其他信息，访问在位置 12,423,956 中的字节需要多少次磁盘访问？

答案：a. 找出每一个块中根据指针大小来划分块大小的磁盘块指针的数目：

$$8K/4 = 2K \text{ pointers per block}$$

I-Node 所支持的最大文件大小是：

12	+	2K	+	(2K × 2K)	+	(2K × 2K × 2K)
直接寻址	一级间接寻址	二级间接寻址		三级间接寻址		
12	+	2K	+	4M	+	8G blocks

将以上数据乘以块大小 (8K)，得到：

$$96KB + 16MB + 32GB + 64TB$$

这就是该系统支持的最大文件大小。

b. 每一个分区中都有 24 位用于识别物理块，由此可知：

$$2^{24} \times 8K = 16M \times 8K = 128GB$$

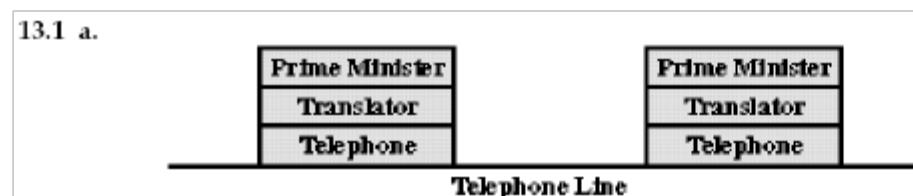
c. 由问题 (a) 中所得的信息可知，直接块只覆盖了第一个 96KB 区域，而一级间接块覆盖了接下来的 16MB 区域。被请求文件的位置是 13MB 而其偏移很明显地随机落在了一级间接块中。因此会有 2 次磁盘存储访问。一次是为了一级间接块，一次是为了包含被请求数据的块。

第13章 网络

13.1、a. 假设法国和中国总理需要通过电话达成一项协定。但是他们都不会说对方的语言，而且身边也没有翻译能将对方的语言翻译过来。但是，他们的工作人员中都有英语翻译人员。请画出与图13.8 类似的图来描述这种情况，并说明每一层之间的交互过程。

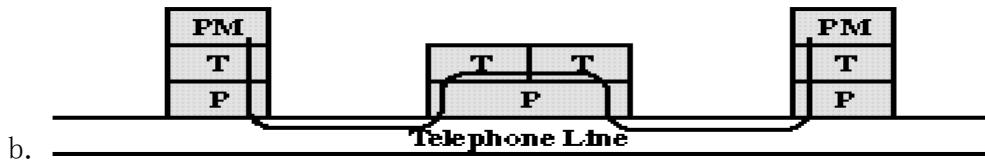
b. 现在假设中国总理只有日语翻译人员，法国总理有德语翻译人员。在德国有德语和日语翻译人员。请画图来说明协定过程并描述电话交谈过程。

答案：



两个总理的讲话就好像他们在直接同对方讲话，例如：当法国总理讲话时，他定位他的话是直接对中国总

理说的。尽管如此，事实上这些信息还是经由电话系统经过两个翻译。法国总理的翻译把他的话翻译成英文并告诉中国总理的翻译，中国总理的翻译收到后将之翻译成中文。



提供一个中间结点服务，在信息通过时将信息翻译。

13.2、请列举协议分层方法的主要缺点。

答案：这种方法的主要缺点可能是处理器处理和数据开销。造成处理器处理开销的原因是7个模块（OSI模块）被调用用来把数据从应用程序移入通信软件。数据开销的主要原因是附加在数据中的各种头部。另一个可能的缺点是至少每一层有一个标准协议。由于分层很多，因此它需要花很长的一段时间发展和发布协议标准。

13.3、一个TCP分段由1500位的数据和160位的头组成。当传送到IP层的时候又增加了另一个160位的头。传输时通过了两个网络，每一个网络都使用一个24位的包头。目的端网络的最大数据包大小为800位。当传送到目的端网络时，网络层包含协议头在内有多少位？

答案：数据+TCP头+IP头=1820bits，这些数据以数据包序列的反向传输，每个包含24bits的网络头和776bits的较高级头或数据，总计需3个包，总传输字节=1820+3x24=1892bits。

13.4、为什么TCP头包含长度域而UDP头没有包含长度域？

答案：UDP（用户数据报）拥有固定的头，而在TCP中这个头是不定长的。

13.5、在以前的版本的TFTP协议规范RFC783中，有如下表述：除了那些用来终止连接的数据包，所有的数据包都会收到单独的确认信息，除非发生超时。新的规范修改如下：除了重复的ACK及用于终止连接的数据包之外，所有的数据包都需要收到确认，除非发生超时。这个变化修正了一个所谓的“魔术师的学徒”问题。请推论出并解释这个问题。

答案：假设A发送一个一个数据包K到B，而自B的响应接收数据包(ACK)被延迟但没丢失。A重发数据包K，被B在此确认收到，至此A收到两个关于K的ACK，而其中一个都会激发数据包K+1的发送，而B将对到来的两个K+1做出响应，从而引起A再次发送两个包K+2，至此，每个数据包和响应包都将被发送两次。

13.6、使用TFTP传送文件时，影响传送所需时间的因素是什么？

答案：TFTP每一轮能传输的最大数值为512字节（数据发送，ACK接收）。因此，最大的传输量是由每一轮的传输时间来划定的。

第14章 分布式处理、客户服务器和集群

14.1、假设 α 是在集群中的n台计算机上同时执行的程序代码的百分率，每台计算机使用一组不同的参数或初始条件。假设其余代码必须由一台处理机串行执行；每台处理机的执行速率是MIPS。

- 当使用互斥执行这个程序的系统时，根据n, α 和 x 给出有效的MIPS率表达式。
- 如果 n=16 且 x=4 MIPS，确定能够产生40 MIPS系统性能的 α 值。

答案：a.MIPS率表达式： $[n\alpha + (1-\alpha)]x = (n\alpha - \alpha + 1)x$

b. $\alpha = 0.6$

14.2、一个应用程序在由9台计算机组成了集群上执行。一个基准程序在该集群上占用了时间T，而且还发现T的25%是应用程序同时在所有9台计算机上运行的时间。在其余的时间，应用程序只能运行在一台独立的计算机上。

a.计算在上述条件下与在一台单独计算机上执行程序相比的有效加速比。也计算 α ，它是上一题程序中已并行化（通过编程或编译手段使得能够使用集群模式）的代码的百分率。



版权说明：本文档由用户提供并上传，收益归属内容提供方，若内容存在侵权，请进行举报或认领

相关推荐

- 操作系统精髓与设计原理第五版 课后题答案
- 操作系统精髓与设计原理第五版中文答案
- 操作系统 精髓与设计原理(第五版))
- 操作系统——精髓与设计原理(第五章答案)
- 操作系统_精髓与设计原理(第五版)课后答案(中文版).doc (1)

猜你想看

- 操作系统第五版--精髓与设计概要第7章课后习题答案2
- 《操作系统精髓与设计原理·第五版》习题答案
- 操作系统第五版所有课后复习题中文答案
- 操作系统精髓与设计原理课后答案
- 操作系统一精髓与设计原理第五版选择题与答案英文版第一到第十一章...

相关好店

维普资讯网

「其它」

知识海洋2020梦想启航

「其它」

趋势小屋

「其它」

知识情报官

「教育」

好大一棵树和远方

「教育」

彼岸开花736

「教育」

工具

收藏

送VIP



领福利

下载文档