

**《操作系统》lab5**

|  |  |
| --- | --- |
| 学 院： | 计算机与信息技术学院 |
| 专 业： | 计算机科学与技术 |
| 班 级： | 计科1602 |
| 姓 名： | 麻锦涛 |
| 学 号： | 16281262 |

**日期：2019.6.10**

1. **实验目的**

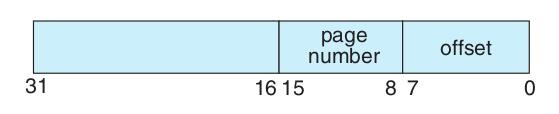
该项目的目标是实现虚拟内存管理的地址转换。

1. **实验要求**

该项目要求编写一个程序，该程序将逻辑地址转换为物理地址，用于大小为2 ^ 16 = 65,536字节的虚拟地址空间。您的程序将从包含逻辑地址的文件中读取，并使用TLB和页表将每个逻辑地址转换为其对应的物理地址，并输出存储在转换后的物理地址中的字节值。

您的学习目标是使用模拟来理解将逻辑地址转换为物理地址所涉及的步骤。这将包括使用**请求分页解决页面错误，管理TLB以及实现页面替换算法**。

具体要求程序可以读取包含几个表示逻辑地址的32位整数的文件。但是，我只需要关注16位地址，因此必须屏蔽每个逻辑地址的最右边16位。这16位分为（1）8位页码和（2）8位页偏移。因此，地址的结构如下所示：



其他细节包括以下内容：

页面表中有2 ^ 8个条目

页面大小为2 ^ 8字节

TLB中有16个条目

帧大小为2 ^ 8字节

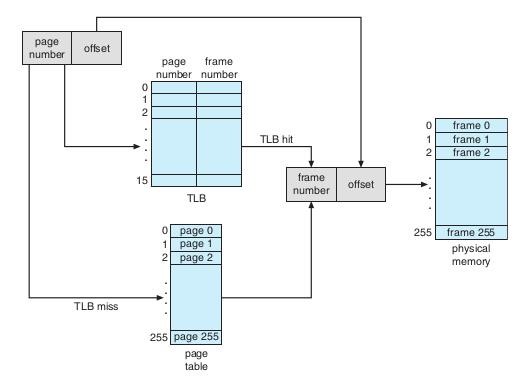
256帧

物理内存为65,536字节（256帧×256字节帧大小）

此外，您的程序只需要关注读取逻辑地址并将其转换为相应的物理地址。您不需要支持写入逻辑地址空间。

**地址翻译**

程序将使用教科书中概述的TLB和页表将逻辑地址转换为物理地址。首先，从逻辑地址中提取页码，并查阅TLB。在TLB命中的情况下，帧号从TLB获得。在TLB未命中的情况下，必须查阅页表。在后一种情况下，从页表获得帧号，或者发生页面错误。地址转换过程的直观表示是：



**处理页面错误**

您的程序将实现**请求分页策略**。后备存储由文件BACKING\_STORE.bin表示，该文件是大小为65,536字节的二进制文件。发生页面错误时，您将从文件BACKING\_STORE读取一个256字节的页面，并将其存储在物理内存中的可用页面框架中。例如，如果页码为15的逻辑地址导致页面错误，则程序将在第15页从BACKING\_STORE读取（请记住页面从0开始，大小为256字节）并将其存储在物理内存的页面框架中。一旦存储了该帧（并且更新了页表和TLB），TLB或页表将解析对页面15的后续访问。

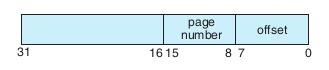
您需要将BACKING\_STORE.bin视为随机访问文件，以便您可以随机搜索文件的某些位置以供阅读。我们建议使用标准C库函数来执行I / O，包括fopen()，fread()，fseek()和fclose()。

物理内存的大小与虚拟地址空间的大小相同--65,536字节 - 因此您不必在页面错误期间关注页面替换。后来，我们使用较少量的物理内存描述了对该项目的修改;此时，将需要一个页面替换策略。

**测试文件**

根据提供的文件addresses.txt，其中包含表示逻辑地址的整数值，范围从0到65535（虚拟地址空间的大小）。您的程序将打开此文件，读取每个逻辑地址并将其转换为相应的物理地址，并在物理地址处输出有符号字节的值。

首先，编写一个简单的程序，根据以下内容提取页码和偏移量：



来自以下整数：

1,256,32768,32769,128,65534,33153

也许最简单的方法是使用运算符进行位屏蔽和位移。一旦您可以正确地建立页码和偏离整数，您就可以开始了。

最初，我们建议您绕过TLB并仅使用页表。一旦页表正常工作，您就可以集成TLB。请记住，地址转换可以在没有TLB的情况下工作; TLB只是让它变得更快。当您准备好实现TLB时，请记住它只有16个条目，因此在更新完整TLB时需要使用替换策略。

您可以使用FIFO或LRU策略来更新TLB。

**如何运行程序**

您的程序应运行如下：

./a.out addresses.txt

您的程序将读取文件addresses.txt，其中包含1,000个逻辑地址，范围从0到65535.您的程序是将每个逻辑地址转换为物理地址，并确定存储在正确物理地址的有符号字节的内容。 （回想一下，在C语言中，char数据类型占用一个字节的存储空间，因此我们建议使用char值。）您的程序将输出以下值：

1.正在转换的逻辑地址（从addresses.txt读取的整数值）。

2.相应的物理地址（程序将逻辑地址转换为）。

3.签名的字节值存储在翻译的物理地址的物理存储器中。我们还提供了文件correct.txt，其中包含文件addresses.txt的正确输出值。您应该使用此文件来确定您的程序是否正确地将逻辑转换为物理地址。

完成后，您的程序将报告以下统计信息：

1.页面错误率 - 导致页面错误的地址引用的百分比。

2. TLB命中率 - 在TLB中解析的地址引用的百分比。由于addresses.txt中的逻辑地址是随机生成的，并且不反映任何内存访问位置，因此不要期望具有高TLB命中率。

**提示：**

可以尝试生成具有局部性的逻辑地址序列，因此可以预期更高的命中率。

到目前为止，该项目假设物理内存与虚拟地址空间大小相同。实际上，物理内存通常比虚拟地址空间小得多。该项目的这一阶段现在假设使用一个较小的物理地址空间，其中包含128个页面帧而不是256个。此更改将需要修改程序，以便跟踪可用页面帧以及使用FIFO或FIFO实现页面替换策略LRU在没有空闲内存时解决页面错误。不要忘记将以前的程序保留为版本1，并将其作为另一个版本启动。

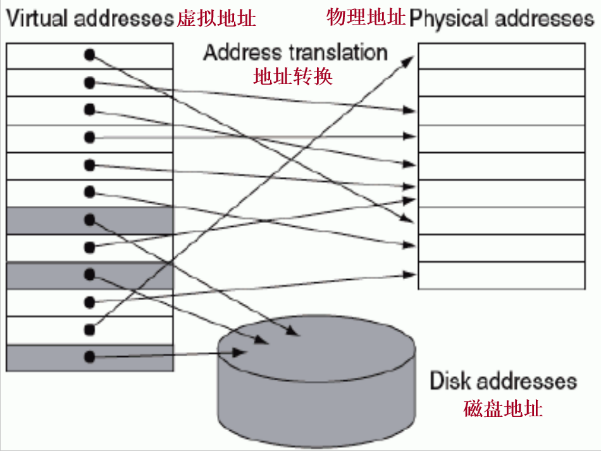
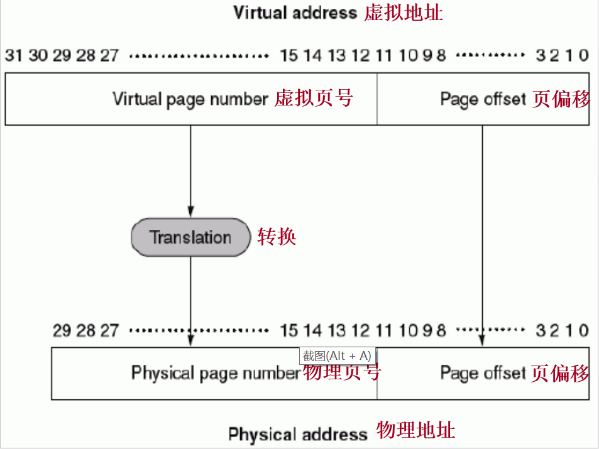
1. **实验原理以及流程图**
2. **物理地址和逻辑地址**

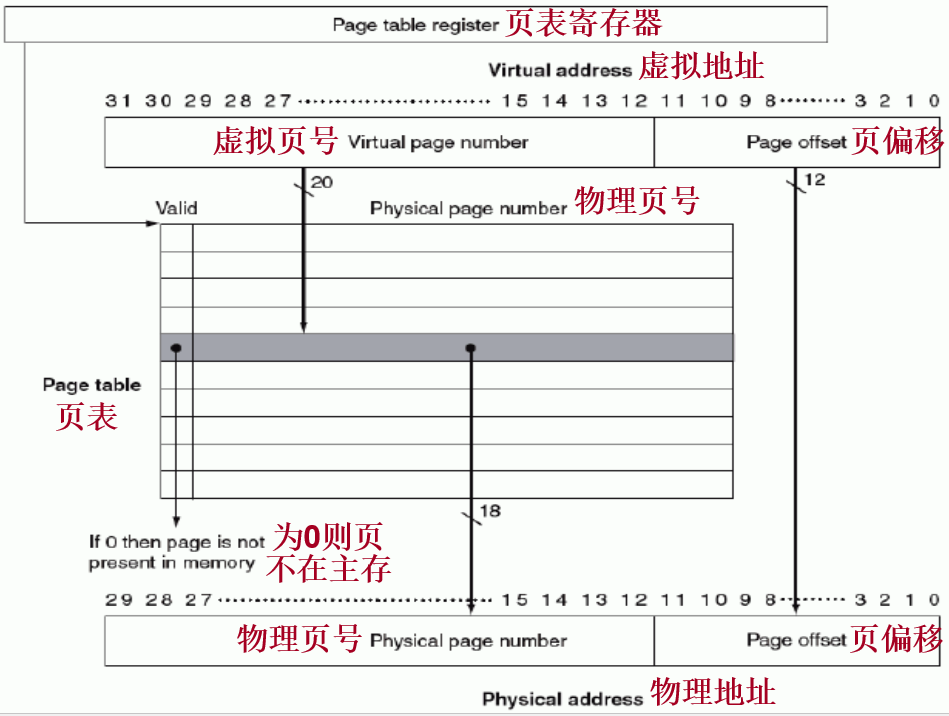
**物理地址**：加载到内存地址寄存器中的地址，内存单元的真正地址。在前端总线上传输的内存地址都是物理内存地址，编号从0开始一直到可用物理内存的最高端。这些数字被北桥(Nortbridge chip)映射到实际的内存条上。物理地址是明确的、最终用在总线上的编号，不必转换，不必分页，也没有特权级检查(no translation, no paging, no privilege checks)。

**逻辑地址**：CPU所生成的地址。逻辑地址是内部和编程使用的、并不唯一。例如，你在进行C语言指针编程中，可以读取指针变量本身值(&操作)，实际上这个值就是逻辑地址，它是相对于你当前进程数据段的地址（偏移地址），不和绝对物理地址相干。

个人认为在于逻辑地址分配更加灵活，可以允许不唯一，看起来也较为直观，例如，一段代码中分配数组，逻辑地址上是连续的，然而在物理地址上，这个数组所占用的页可能分散开来，物理地址上就是不连续的，这样对程序的可理解性上有影响。另外，有了逻辑地址这个概念，才能使用虚拟内存技术。

二者之间的关系如下面几张图：



1. **请求分页存储管理**

**页和帧**

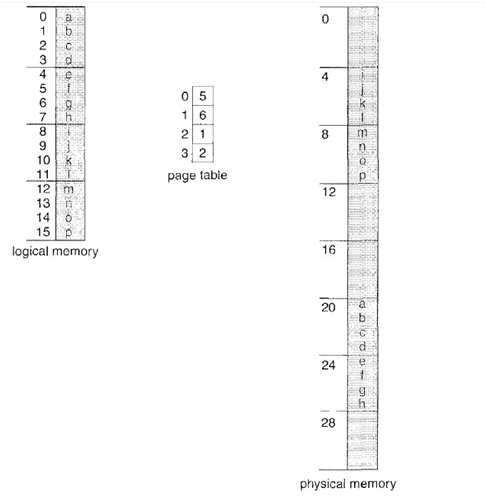
而在这里，分页的最大作用就在于：使得进程的物理地址空间可以是非连续的。

物理内存被划分为一小块一小块，每块被称为帧(Frame)。分配内存时，帧是分配时的最小单位，最少也要给一帧。在逻辑内存中，与帧对应的概念就是页(Page)。

逻辑地址的表示方式是：前部分是页码后部分是页偏移。

例如，已知逻辑空间地址为2^m个字节（也就是说逻辑地址的长度是m位），已知页大小是2^n字节。那么一共可以有2^(m-n)个页。因此页码部分会占m-n位，之后的n位，用来存储页偏移。

举个例子， 页大小为4B，而逻辑内存为32B（8页），逻辑地址0的页号为0，页号0对应帧5，因此逻辑地址映射为物理地址5\*4+0=20。逻辑地址3映射物理地址5\*4+3=23。逻辑地址13(4\*3+1，页号为3，偏移为1，因此帧号为2)，映射到物理地址9。



采用分页技术不会产生外部碎片(内存都被划分为帧)，但可能产生内部碎片(帧已经是最小单元，因此帧内部可能有空间没有用到)。

按概率计算下来，每个进程平均可有半个帧大小的内部碎片。

页表的硬件实现

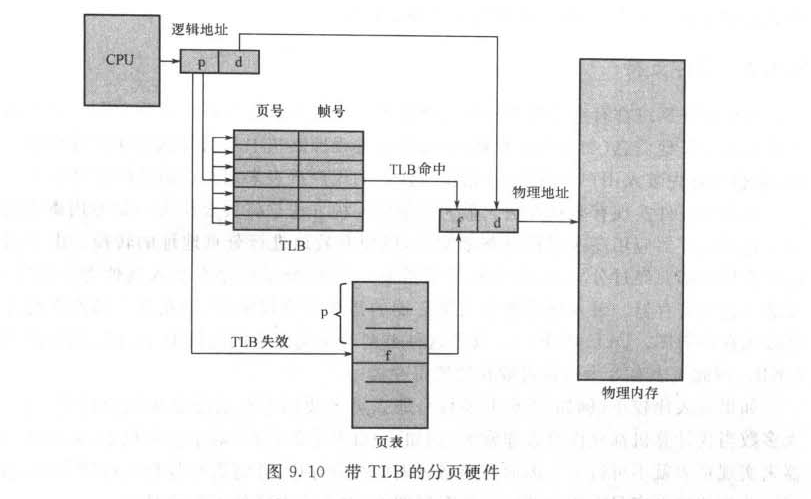
上一小节中写到页表是逻辑地址转化到物理地址的关键所在。那么页表如何存储？

每个操作系统都有自己的方法来保存页表。绝大多数都会为每个进程分配一个页表。现在由于页表都比较大，所以放在内存中(以往是放在一组专用寄存器里)，其指针存在进程控制块(PCB)里，当进程被调度程序选中投入运行时，系统将其页表指针从进程控制块中取出并送入用户寄存器中。随后可以根据此首地址访问页表。

页表的存储方式是**TBL(Translation look-aside buffer, 翻译后备缓冲器)**+内存。TBL实际上是一组硬件缓冲所关联的快速内存。若没有TBL，操作系统需要两次内存访问来完成逻辑地址到物理地址的转换，访问页表算一次，在页表中查找算一次。TBL中存储页表中的一小部分条目，条目以键值对方式存储。

常规存储器管理方式（基本分页、基本分段）的特征：

一次性.都要求将作业所有装入内存后方能执行。很多作业在每次执行时，并不是其所有程序和数据都要用到。假设一次性地装入其所有程序，造成内存空间的浪费。

驻留性作业装入内存后，便一直驻留在内存中，直至作业执行结束。虽然执行中的进程会因I/O而长期等待，或有的程序模块在执行过一次后就不再须要(执行)了，但它们都仍将继续占用宝贵的内存资源。  


**虚拟存储器的定义**

应用程序在执行之前，没有必要所有装入内存，仅须将那些当前要执行的少数页面或段先装入内存便可执行，其余部分暂留在盘上。程序在执行时，假设它所要訪问的页(段)已调入内存，便可继续执行下去；但假设程序所要訪问的页(段)尚未调入内存(称为缺页或缺段)，此时程序应利用OS所提供的**请求调页(段)功能**，将它们调入内存，以使进程能继续执行下去。假设此时内存已满，无法再装入新的页(段)，则还须再利用页(段)的**置换功能**，将内存中临时不用的页(段)调至盘上，腾出足够的内存空间后，再将要訪问的页(段)调入内存，使程序继续执行下去。

虚拟存储器是指具有请求调入功能和置换功能，能从逻辑上对内存容量加以扩充的一种存储器系统。其逻辑容量由内存容量和外存容量之和所决定，其执行速度接近于内存速度，而每位的成本却又接近于外存。可见，虚拟存储技术是一种性能很优越的存储器管理技术，故被广泛地应用于大、中、小型机器和微型机中。

**请求分页存储管理方式**

**定义：**

请求分页系统是建立在基本分页系统的基础上，为了能支持虚拟存储器功能而添加了请求调页功能和页面置换功能。

**页表机制**

在请求分页系统中所须要的主要数据结构是**页表**。其基本作用仍然是**将用户地址空间中的逻辑地址变换为内存空间中的物理地址**。因为仅仅将应用程序的一部分调入内存，另一部分仍在盘上，故须在页表中再添加若干项，供程序(数据)在换进、换出时參考。在请求分页系统中的每个页表项例如以下所看到的：

http://img.blog.csdn.net/20140506010220625?watermark/2/text/aHR0cDovL2Jsb2cuY3Nkbi5uZXQvY291dF9zZXY=/font/5a6L5L2T/fontsize/400/fill/I0JBQkFCMA==/dissolve/70/gravity/SouthEast

现对当中各字段说明例如以下：  
状态位P：用于指示该页**是否已调入内存**，供程序訪问时參考。  
访问字段A：用于记录本页在一段时间内**被访问的次数**，或记录本页近期已有多长时间未被访问，供选择换出页面时參考。

改动位M：表示该页在调入内存后**是否被改动过**。因为内存中的每一页都在外存上保留一份副本，因此，若未被改动，在置换该页时就不需再将该页写回到外存上，以降低系统的开销和启动磁盘的次数；若已被改动，则必须将该页重写到外存上，以保证外存中所保留的始终是最新副本。

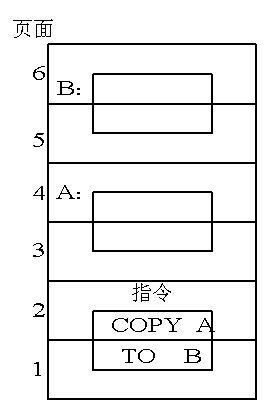
简言之，M位供置换页面时參考。

外存地址：用于指出该页在外存上的地址，一般是**物理块号**，供调入该页时參考。

**缺页中断机构**

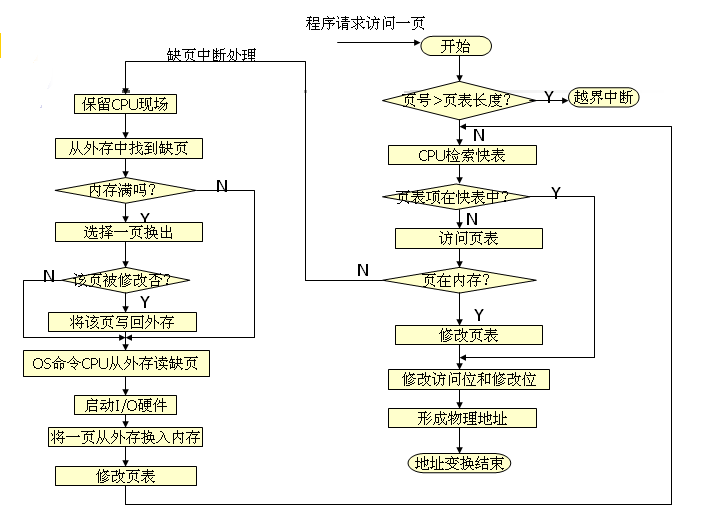
在请求分页系统中，每当所要訪问的页面不在内存时，**便产生一缺页中断，请求OS将所缺之页调入内存**。缺页中断作为中断，它们相同须要经历诸如保护CPU环境、分析中断原因、转入缺页中断处理程序进行处理、恢复CPU环境等几个步骤。但缺页中断又是一种特殊的中断，它与一般的中断相比，有着明显的差别，主要表如今以下两个方面：  
在指令运行期间产生和处理中断信号。通常，**CPU都是在一条指令运行完后，才检查是否有中断请求到达**。若有，便去响应，否则，继续运行下一条指令。然而，缺页中断是在指令运行期间，发现所要訪问的指令或数据不在内存时所产生和处理的。

一条指令在运行期间，可能产生多次缺页中断。在下图中示出了一个样例。如在运行一条指令COPY A TO B时，可能要产生6次缺页中断，当中指令本身跨了两个页面，A和B又分别各是一个数据块，也都跨了两个页面。基于这些特征，系统中的硬件机构应能保存多次中断时的状态，并保证最后能返回到中断前产生缺页中断的指令处继续运行。



**地址变换机构**

请求分页系统中的地址变换机构，是在分页系统地址变换机构的基础上，再为实现虚拟存储器而添加了某些功能而形成的，如产生和处理缺页中断，以及从内存中换出一页的功能等等。下图表示出了请求分页系统中的地址变换过程。在进行地址变换时，首先去检索快表，试图从中找出所要訪问的页。若找到，便改动页表项中的訪问位。对于写指令，还须将改动位置成“1”，然后利用页表项中给出的物理块号和页内地址形成物理地址。地址变换过程到此结束。请求分页系统的地址变换机构。是在分页系统地址变换机构的基础上，又增加了一些功能。



例：某虚拟存储器的用户空间共有32个页面，每页1KB，主存16KB。假定某时刻系统为用户的第0、1、2、3页分别分配的物理块号为5、10、4、7，试将虚拟地址0A5C和093C变换为物理地址。

解：虚拟地址为：页号（2^5=32）5位 页内位移（1K =2^10=1024）10位

物理地址为 物理块号（2^4=16）4位 因为页内是10 位， 块内位移（1K =2^10=1024）10位

虚拟地址OA5C对应的二进制为：

00010 1001011100

即虚拟地址OA5C的页号为2，页内位移为1001011100，由题意知对应的物理地址为：0100 1001011100即125C

同理求093C。

**内存分配策略和分配算法**

在请求分页系统中，为进程分配内存时，将涉及以下三个问题：

最小物理块数的确定；物理块的分配策略；物理块的分配算法。

**最小物理块数的确定**

概念：最小物理块数：是指能保证进程正常运行所需的最小物理块数。

确定方法：与计算机的硬件结构有关，取决于指令的格式、功能和寻址方式。

**物理块的分配策略**

内存分配策略：固定和可变分配策略

置换策略：全局置换和局部置换

三种合适的策略如下：

**固定分配局部置换（Fixecd Allocation,Local replacement）**：为每个进程分配固定数目n的物理块，在整个运行中都不改变。如出现缺页，则从中置换一页。

**可变分配全局置换(VariableAllocatio,Global Repalcement)**：分配固定数目的物理块，但OS自留一空闲块队列，若发现缺页，则从空闲块队列中分配一空闲块与该进程，并调入缺面于其中。当空闲块队列用完时，OS才从内存中任选择一页置换。

**可变分配局部置换(VariableAllocatio,Local Repalcement)**：分配一定数目的物理块，若发现缺页，则从该进程的页面中置换一页，根据该进程缺页率高低，则可增加或减少物理块。也就是若某进程频繁的发生缺页中断，则系统再为该进程分配物理块，知道缺页率减少到一定程度。

**物理块的分配算法**

在采用固定分配策略时，将系统中可供分配的所有物理块分配给各个进程，可采用以下几种算法：

**平均分配算法**：将系统中所有可供分配的物理块，平均分配给每个进程。缺点：未考虑各进程本身的大小。

**按比例分配算法**：这是根据进程的大小按比例分配物理块的算法。如果系统中共有n个进程，每个进程的页面数为Si，则系统中各进程页面数的总和为：



又假定系统中可用的物理块总数为m，则每个进程所能分到的物理块数为bi，将有：



b应该取整，它必须大于最小物理块数。

**考虑优先权的分配算法**：将系统提供的物理块一部分根据进程大小先按比例分配给各个进程，另一部分再根据各进程的优先权适当增加物理块数。

**调页策略和页面调入过程**

什么时候将一个页面由外存调入内存？从何处将页面调入内存？这就是调页策略所要解决的问题。

**何时调入页面？**

* + - 预调页策略：将那些预计在不久便被访问的页预先调入内存。这种调入策略提高了调页的效率，减少了I/O次数。但由于这是一种基于局部性原理的预测，若预调入的页面在以后很少被访问，则造成浪费，故这种方式常用于程序的首次调入。
    - 请求调页策略：当进程运行中访问的页出现缺页时，则发出缺页中断，提出请求调页，由OS将所需页调入内存。这种策略实现简单，应用于目前的虚拟存储器中，但易产生较多的缺页中断，且每次调一页，系统开销较大，容易产生抖动现象。

**注意：首次：预调页；运行时：请求调页。**

**从何处调入页面？**

在请求分页系统中，通常将外存分成了**文件区**和**对换区**，文件区按离散分配方式存放文件，对换区按连续分配方式存放对换页。

* + - 系统有足够的对换区空间情况：运行前可将与进程相关的文件从文件区复制至对换区，以后缺页时，全部从对换区调页。**只从对换区调页。**
    - 系统没有足够的对换区空间情况：

页面不会被修改：凡是不会被修改的文件，每次都直接从文件区调页，换出时不必换出。正因为没有被修改，因此不用换出，因为文件区存放的页面没改变。**只从文件区调页。**

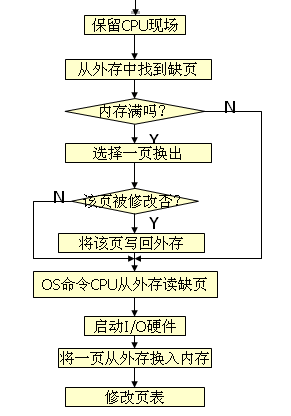
页面可能被修改：若对可能会修改的文件第一次调页直接从文件区，换出时换至对换区，以后从对换区调页。第一次从文件区调入以后从对换区。**从文件区/对换区调页**

* + - UNIX方式：凡未运行过的页面均从文件区调页，运行过的页面和换出的页面均从对换区调页。

页面调入的具体过程如下：每当程序所要访问的页面未在内存时，便向CPU发出一缺页中断，中断处理程序首先保留CPU环境，分析中断原因后，转入缺页中断处理程序。该程序通过查找页表，得到该页在外存上的物理块后，如果此时内存能容纳新页，则启动磁盘I/O将所缺之页调入内存，然后修改页表。如果内存已满，则需按照某种置换算法从内存中选出一页准备换出；如果该页未被修改过，可不必写回磁盘；但如果此页已被修改，则必须将它写回磁盘，然后把所缺的页调入内存，并修改页表中的相应表项，置其存在位为“1”，并将此页表项写入快表。在缺页调入内存后，利用修改后的页表，形成所要访问的物理地址，再去访问内存数据。

其实就是下面四步：缺页中断；保留CPU环境；缺页中断处理；访问内存数据

流程图如下：



1. **FIFO算法**

一种最早出现的置换算法。该算法总是淘汰最先进入内存的页面，即选择在内存中驻留时间最久的页面予以淘汰。该算法实现简单，只需把一个进程已调入内存的页面，按先后次序链接成一个队列，并设置一个指针，称为替换指针，使它总是指向最老的页面。但该算法与进程实际运行的规律不相适应，因为在进程中，有些页面经常被访问，比如，含有全局变量、常用函数、例程等的页面，FIFO 算法并不能保证这些页面不被淘汰。

将内存块数组视为队列，在实现的过程中，初始化时，队头指针指向第一个内存块位置，队尾指针指向最后一个内存块位置。每次进行页面置换时，算法总置换当前队头，并且，队头指针循环加1，队尾指针循环加1.

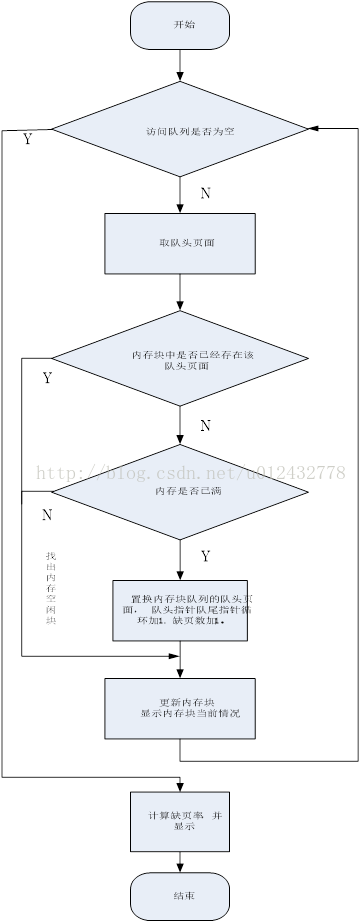
在内存块初始化后，取出页面访问序列队列的队头。首先判断内存块中是否已经存在该队头页面，如果存在则直接显示内存块当前情况；否则，判断此时内存是否已满。如果内存未满，循环遍历找出空闲内存块，进行页面置换；若内存已满，置换内存块队列的队头页面，缺页数加1，队头指针队尾指针循环加1. 如此循环迭代，直到页面访问序列队列为空时，整个算法执行完毕。最后计算并显示缺页率。

这里假定系统为某进程分配了三个物理块，并考虑有以下页面号引用串：7, 0, 1, 2, 0, 3, 0,4,2,3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1。釆用FIFO算法进行页面置换，进程访问页面2时，把最早进入内存的页面7换出。然后访问页面3时，再把2, 0, 1中最先进入内存的页换出。由下图可以看出，利用FIFO算法时进行了12次页面置换。



缺点：FIFO算法还会产生当所分配的物理块数增大而页故障数不减反增的异常现象，这是由Belady于1969年发现，故称为Belady异常，如下图所示。只有FIFO算法可能出现Belady异常，而LRU和OPT算法永远不会出现Belady异常。

其流程图如下：



1. **实现过程**

代码如下：

#include <iostream>

#include <vector>

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#define MAX\_PAGE (256)

using namespace std;

typedef struct

{

int status;//页有没有分配，0表示没有分配，1表示分配

int framenumber;//这个页对应的帧号

}Page;

typedef struct

{

int pagenumber;

int framenumber;

}TLB;

Page pageTable[MAX\_PAGE];//虚拟内存和物理内存一样大的页表、256个页表

vector<TLB> tlb;//定义TLB

int nextframenumber=0;

char mem[65536];

int totalcnt=0;

int pagecnt=0;

int tlbcnt=0;

void virtualToPhysical(int vir)

{

int pagenumber,offset;

int tlbindex;

int physicaladdr=-1;

pagenumber=(vir>>8)&0xff;//获取页的数字

offset=vir&0xff;//获取页上面的偏移地址

for(int i=0;i<tlb.size();i++)//加快内存访问，相当于缓存，页码和帧

{

if(tlb[i].pagenumber==pagenumber)

{

physicaladdr=(tlb[i].framenumber<<8)|offset;//找到物理地址

     tlbcnt++;

break;

}

}

if(physicaladdr==-1)//没有找到物理地址时进入，TLB里面没有

{

if(pageTable[pagenumber].status==0)//0的时候没有分配页，去从BACKING\_STORE.bin里面加载内容到分配页

{

FILE\* fp=fopen("BACKING\_STORE.bin","rb");

fseek(fp, pagenumber\*256,0);

fread(mem+(256\*nextframenumber),1,256,fp);

fclose(fp);

pageTable[pagenumber].status=1;//pagenumber就是分配的页

pageTable[pagenumber].framenumber=nextframenumber++;//分配的帧号，物理内存分段，++是为了指向下一个空闲位置（没有分配的物理地址

}

     else

     pagecnt++;//页表命中的次数计数

TLB t;

t.pagenumber=pagenumber;

t.framenumber=pageTable[pagenumber].framenumber;

     tlb.push\_back(t);//当前访问的页表记录到TLB里面

     if(tlb.size()>16)//控制TLB不大于16控制大小

     tlb.erase(tlb.begin());

physicaladdr=(pageTable[pagenumber].framenumber<<8)|offset;//从页表里面访问的时候计算物理地址

}

printf("Virtual address: %d Physical address: %d Value: %d\n",vir,physicaladdr,mem[physicaladdr]);//输出

}

int main(int argc,char \*argv[])

{

FILE\* fp;

char buf[20];

if(argc!=2)

return -1;

memset(pageTable,0,sizeof(Page)\*MAX\_PAGE);//初始化页表

fp=fopen(argv[1],"rb");

if(fp==NULL)

return -1;

while(!feof(fp))//判断有没有读到文件末尾

{

memset(buf,0,sizeof(buf));

fgets(buf,20,fp);//每次读一行

if(strlen(buf)<=0)

continue;

virtualToPhysical(atoi(buf));

totalcnt++;

}

fclose(fp);

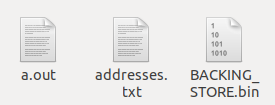
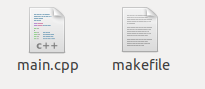
if(totalcnt>0)

printf("page rate:%d%%,tlb rate:%d%%\n",100\*pagecnt/totalcnt,100\*tlbcnt/totalcnt);//统计页表和TLB命中的概率

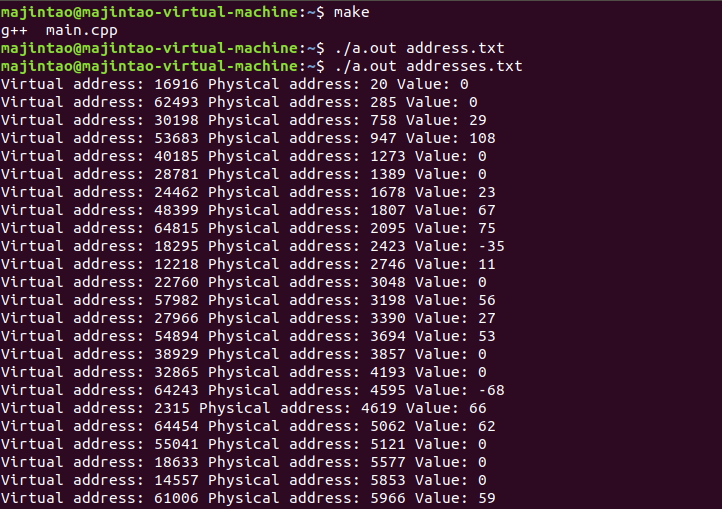
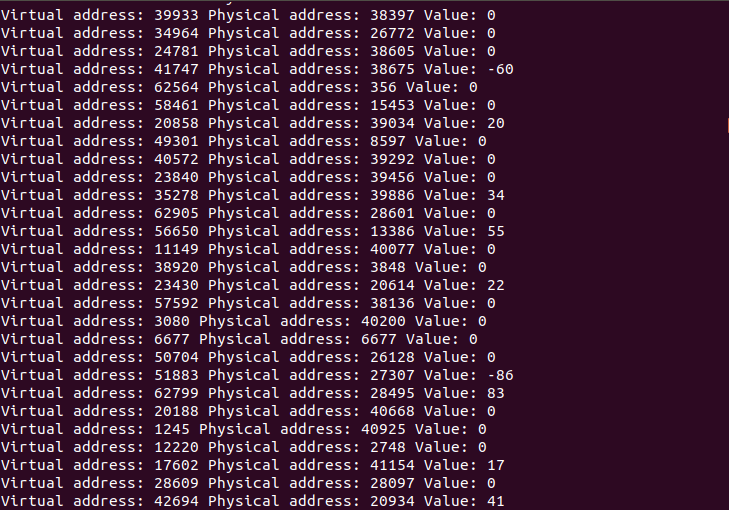
return 0;

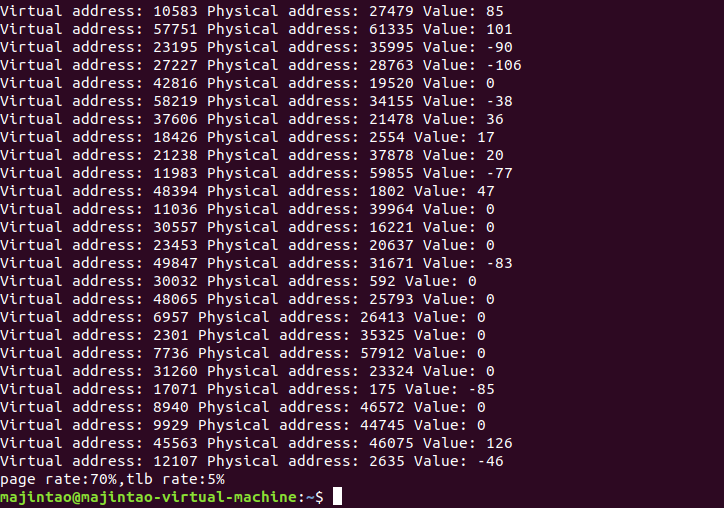
}

有以下文件：

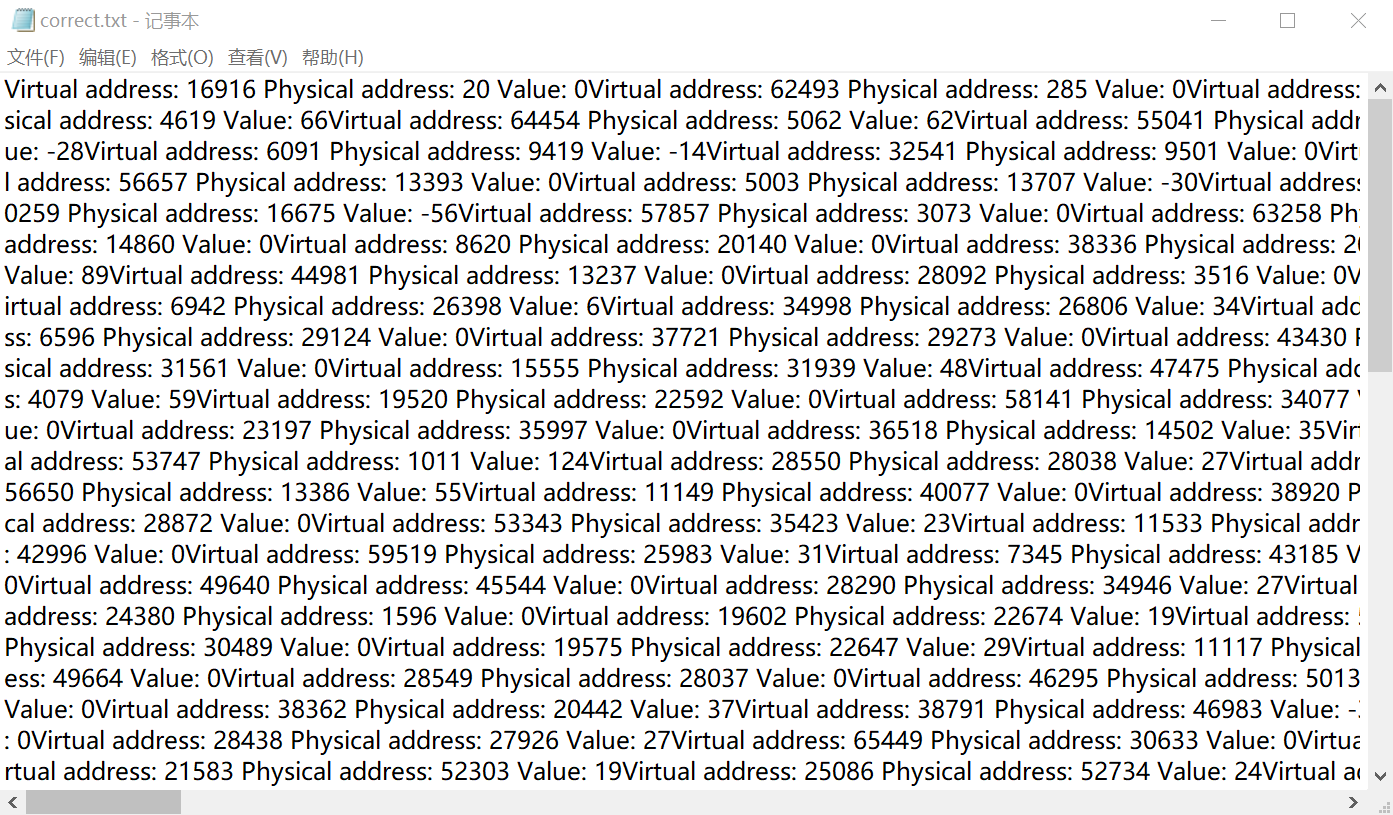


打开终端，输入make，然后输入“./a.out addresses.txt”，程序将打开此文件，读取每个逻辑地址并将其转换为相应的物理地址，并在物理地址处输出有符号字节的值：



打开correct.txt进行对比，发现没问题

****

**1.页面错误率 - 导致页面错误的地址引用的百分比，这里得出是30%**

**2. TLB命中率 - 在TLB中解析的地址引用的百分比，这里得出是5%**

1. **实验总结与收获**

**本次实验的目的是实现虚拟内存管理的地址转换。通过这次实验我也对请求分页解决页面错误，TLB以及页面替换算法进行了一次系统的实现和复习。**

**下面我总结了不同内存的管理方法：**

  内存管理主要包括虚地址、地址变换、内存分配和回收、内存扩充、内存共享和保护等功能。

**连续分配存储管理方式**

 连续分配是指为一个用户程序分配连续的内存空间。连续分配有单一连续存储管理和分区式储管理两种方式。

**单一连续存储管理**

在这种管理方式中，内存被分为两个区域：系统区和用户区。应用程序装入到用户区，可使用用户区全部空间。其特点是，最简单，适用于单用户、单任务的操作系统。CP／M和 DOS 2．0以下就是采用此种方式。这种方式的最大优点就是易于管理。但也存在着一些问题和不足之处，例如对要求内存空间少的程序，造成内存浪费；程序全部装入，使得很少使用的程序部分也占用—定数量的内存。

**分区式存储管理**

为了支持多道程序系统和分时系统，支持多个程序并发执行，引入了分区式存储管理。分区式存储管理是把内存分为一些大小相等或不等的分区，操作系统占用其中一个分区，其余的分区由应用程序使用，每个应用程序占用一个或几个分区。分区式存储管理虽然可以支持并发，但难以进行内存分区的共享。

分区式存储管理引人了两个新的问题：内碎片和外碎片。

  内碎片是占用分区内未被利用的空间，外碎片是占用分区之间难以利用的空闲分区(通常是小空闲分区)。

  为实现分区式存储管理，操作系统应维护的数据结构为分区表或分区链表。表中各表项一般包括每个分区的起始地址、大小及状态(是否已分配)。

分区式存储管理常采用的一项技术就是**内存紧缩(compaction)。**

**固定分区(nxedpartitioning)**。

 固定式分区的特点是把内存划分为若干个固定大小的连续分区。分区大小可以相等：这种作法只适合于多个相同程序的并发执行(处理多个类型相同的对象)。分区大小也可以不等：有多个小分区、适量的中等分区以及少量的大分区。根据程序的大小，分配当前空闲的、适当大小的分区。

**优点**：易于实现，开销小。

**缺点主要有两个**：内碎片造成浪费；分区总数固定，限制了并发执行的程序数目。

**动态分区(dynamic partitioning)。**

 动态分区的特点是动态创建分区：在装入程序时按其初始要求分配，或在其执行过程中通过系统调用进行分配或改变分区大小。与固定分区相比较其优点是：没有内碎片。但它却引入了另一种碎片——外碎片。动态分区的分区分配就是寻找某个空闲分区，其大小需大于或等于程序的要求。若是大于要求，则将该分区分割成两个分区，其中一个分区为要求的大小并标记为“占用”，而另一个分区为余下部分并标记为“空闲”。分区分配的先后次序通常是从内存低端到高端。动态分区的分区释放过程中有一个要注意的问题是，将相邻的空闲分区合并成一个大的空闲分区。

下面列出了几种常用的分区分配算法：

**最先适配法(nrst-fit)：**按分区在内存的先后次序从头查找，找到符合要求的第一个分区进行分配。该算法的分配和释放的时间性能较好，较大的空闲分区可以被保留在内存高端。但随着低端分区不断划分会产生较多小分区，每次分配时查找时间开销便会增大。

**下次适配法(循环首次适应算法 next fit)：**按分区在内存的先后次序，从上次分配的分区起查找(到最后{区时再从头开始}，找到符合要求的第一个分区进行分配。该算法的分配和释放的时间性能较好，使空闲分区分布得更均匀，但较大空闲分区不易保留。

**最佳适配法(best-fit)：**按分区在内存的先后次序从头查找，找到其大小与要求相差最小的空闲分区进行分配。从个别来看，外碎片较小；但从整体来看，会形成较多外碎片优点是较大的空闲分区可以被保留。

**最坏适配法(worst- fit)：**按分区在内存的先后次序从头查找，找到最大的空闲分区进行分配。基本不留下小空闲分区，不易形成外碎片。但由于较大的空闲分区不被保留，当对内存需求较大的进程需要运行时，其要求不易被满足。

伙伴系统

固定分区和动态分区方式都有不足之处。固定分区方式限制了活动进程的数目，当进程大小与空闲分区大小不匹配时，内存空间利用率很低。动态分区方式算法复杂，回收空闲分区时需要进行分区合并等，系统开销较大。伙伴系统方式是对以上两种内存方式的一种折衷方案。  
        伙伴系统规定，无论已分配分区或空闲分区，其大小均为 2 的 k 次幂，k 为整数， l≤k≤m，其中：

        2^1 表示分配的最小分区的大小，

        2^m 表示分配的最大分区的大小，

        通常 2^m是整个可分配内存的大小。   
        假设系统的可利用空间容量为2^m个字， 则系统开始运行时， 整个内存区是一个大小为2^m的空闲分区。在系统运行过中， 由于不断的划分，可能会形成若干个不连续的空闲分区，将这些空闲分区根据分区的大小进行分类，对于每一类具有相同大小的所有空闲分区，单独设立一个空闲分区双向链表。这样，不同大小的空闲分区形成了k(0≤k≤m)个空闲分区链表。

**分配步骤：**

       当需要为进程分配一个长度为n 的存储空间时:

       首先计算一个i 值，使 2^(i－1) <n ≤ 2^i，

       然后在空闲分区大小为2^i的空闲分区链表中查找。

       若找到，即把该空闲分区分配给进程。

       否则，表明长度为2^i的空闲分区已经耗尽，则在分区大小为2^(i＋1)的空闲分区链表中寻找。

       若存在 2^(i＋1)的一个空闲分区，则把该空闲分区分为相等的两个分区，**这两个分区称为一对伙伴，**其中的一个分区用于配，   而把另一个加入分区大小为2^i的空闲分区链表中。

 若大小为2^(i＋1)的空闲分区也不存在，则需要查找大小为2^(i＋2)的空闲分区， 若找到则对其进行两次分割：

  第一次，将其分割为大小为 2^(i＋1)的两个分区，一个用于分配，一个加入到大小为 2^(i＋1)的空闲分区链表中；

   第二次，将第一次用于分配的空闲区分割为 2^i的两个分区，一个用于分配，一个加入到大小为 2^i的空闲分区链表中。

若仍然找不到，则继续查找大小为 2^(i＋3)的空闲分区，以此类推。

由此可见，在最坏的情况下，可能需要对 2^k的空闲分区进行 k 次分割才能得到所需分区。

与一次分配可能要进行多次分割一样，一次回收也可能要进行多次合并，如回收大小为2^i的空闲分区时，若事先已存在2^i的空闲分区时，则应将其与伙伴分区合并为大小为2^i＋1的空闲分区，若事先已存在2^i＋1的空闲分区时，又应继续与其伙伴分区合并为大小为2^i＋2的空闲分区，依此类推。  
        在伙伴系统中，其分配和回收的时间性能取决于查找空闲分区的位置和分割、合并空闲分区所花费的时间。与前面所述的多种方法相比较，由于该算法在回收空闲分区时，需要对空闲分区进行合并，所以其时间性能比前面所述的分类搜索算法差，但比顺序搜索算法好，而其空间性能则远优于前面所述的分类搜索法，比顺序搜索法略差。 需要指出的是，在当前的操作系统中，普遍采用的是下面将要讲述的基于分页和分段机制的虚拟内存机制，该机制较伙伴算法更为合理和高效，但在多处理机系统中，伙伴系统仍不失为一种有效的内存分配和释放的方法，得到了大量的应用。

 **内存紧缩**

**内存紧缩：**将各个占用分区向内存一端移动，然后将各个空闲分区合并成为一个空闲分区。

   这种技术在提供了某种程度上的灵活性的同时，也存在着一些弊端，例如：对占用分区进行内存数据搬移占用CPU时间；如果对占用分区中的程序进行“浮动”，则其重定位需要硬件支持。

  紧缩时机：每个分区释放后，或内存分配找不到满足条件的空闲分区时。

**堆结构的存储管理的分配算法：**

 在动态存储过程中，不管哪个时刻，可利用空间都是-一个地址连续的存储区，在编译程序中称之为"堆"，每次分配都是从这个可利用空间中划出一块。其实现办法是：设立一个指針，称之为堆指针，始终指向堆的最低（或锻联）地址。当用户申请N个单位的存储块时，堆指针向高地址（或 低地址）称动N个存储单位，而移动之前的堆指针的值就是分配给用户的占用块的初始地址。例如，某个串处理系统中有A、B、C、D这4个串，其串值长度分别為12,6,10和8. 假设堆指针free的初值为零，则分配给这4个串值的存储空间的初始地址分别为0.12.18和 28,如图8.12(a)和（b)所示，分配后的堆指针的值为36。 因此，**这种堆结构的存储管理的分配算法非常简单**，

**释放内存空间执行内存紧缩：**

   回收用户释放的空闲块就比较麻烦.由于系统的可利用空间始终是一个绝址连续的存储块，因此回收时必须将所释放的空间块合并到整个堆上去才 能重新使用，这就是"存储策缩"的任务.通常，有两种做法：

 一种是一旦有用户释放存储块即进行回收紧缩，例始，图8.12 (a)的堆，在c串释放存储块时即回收紧缩，例如图8.12 (c)的堆，同时修改串的存储映像成图8.12(d)的状态；

  另一种是在程序执行过程中不回收用户随时释放的存储块，直到可利用空同不够分配或堆指针指向最高地址时才进行存储紧缩。此时紧缩的目的是将堆中所有的空间块连成一块，即将所有的占用块部集中到 可利用空间的低地地区，而剩余的高地址区成为一整个地继连续的空闲块.

 和无用单元收集类似，为实现存储紫编,首先要对占用块进行“标志”，标志算法和无用单元收集类同(存储块的结构可能不同）,其次需进行下列4步雄作：

（1）计算占用块的新地址。从最低地址开始巡査整个存储空间，对每一个占用块找到它在紧缩后的新地址。 为此,需设立两个指针随巡查向前移动,这两个指针分别指示占用 块在紧缩之前和之后的原地址和新地址。因此,在每个占用块的第-·个存储单位中，除了 设立长度域(存储该占用换的大小）和标志域(存储区别该存储块是占用块或空闲块的标 志）之外，还需设立一个新地址城，以存储占用块在紧缩后应有的新地址，即建立一张新， 旧地址的对照表m

 (2)修改用户触初始变量表，以便在存储紧缩后用户程序能继续正常运行\*。

 (3)检查每个占用块中存储的数据， 若有指向其他存储换的指针，则需作相应修改.

 (4)将所有占用块迁移到新地址走，这实质上是作传送数据的工作。

 至此，完成了存储紧缩的操作，最后,将堆指针赋以新值（即紧缩后的空闲存储区的最低地址）。

  可见,存储紧缩法比无用单元收集法更为复杂，前者不仅要传送数据（进行占用块迁移），而且还有需要修改所有占用块中的指针值。因此，存储紧缩也是个系统操作，且非不得已就不用。

**覆盖和交换技术**

 覆盖技术

**引入覆盖 (overlay)**技术的目标是在较小的可用内存中运行较大的程序。这种技术常用于多道程序系统之中，与分区式存储管理配合使用。

 覆盖技术的原理：一个程序的几个代码段或数据段，按照时间先后来占用公共的内存空间。将程序必要部分(常用功能)的代码和数据常驻内存；可选部分(不常用功能)平时存放在外存(覆盖文件)中，在需要时才装入内存。不存在调用关系的模块不必同时装入到内存，从而可以相互覆盖。

在任何时候只在内存中保留所需的指令和数据；当需要其它指令时，它们会装入到刚刚不再需要的指令所占用的内存空间；

 如在同一时刻，CPU只能执行B，C中某一条。B，C之间就可以做覆盖。

  覆盖技术的缺点是编程时必须划分程序模块和确定程序模块之间的覆盖关系，增加编程复杂度；从外存装入覆盖文件，以时间延长换取空间节省。

 覆盖的实现方式有两种：以函数库方式实现或操作系统支持。

**交换技术**

**交换 (swapping)**技术在多个程序并发执行时，可以将暂时不能执行的程序（进程）送到外存中，从而获得空闲内存空间来装入新程序（进程），或读人保存在外存中而处于就绪状态的程序。交换单位为整个进程的地址空间。交换技术常用于多道程序系统或小型分时系统中，因为这些系统大多采用分区存储管理方式。与分区式存储管理配合使用又称作“对换”或“滚进／滚出” (roll-in／roll-out)。

**原理：**暂停执行内存中的进程，将整个进程的地址空间保存到外存的交换区中（换出swap out），而将外存中由阻塞变为就绪的进程的地址空间读入到内存中，并将该进程送到就绪队列（换入swap in）。

**交换**技术优点之一是增加并发运行的程序数目，并给用户提供适当的响应时间；与覆盖技术相比交换技术另一个显著的优点是不影响程序结构。交换技术本身也存在着不足，例如：对换人和换出的控制增加处理器开销；程序整个地址空间都进行对换，没有考虑执行过程中地址访问的统计特性。

覆盖与交换比较

1）与覆盖技术相比，交换不要求程序员给出程序段之间的覆盖结构。

 2）交换主要是在进程与作业之间进行，而覆盖则主要在同一作业或进程内进行。 另外覆盖只能覆盖那些与覆盖程序段无关的程序段。

**页式和段式存储管理**

 在前面的几种存储管理方法中，为进程分配的空间是连续的，使用的地址都是物理地址。如果允许将一个进程分散到许多不连续的空间，就可以避免内存紧缩，减少碎片。基于这一思想，通过引入进程的逻辑地址，把进程地址空间与实际存储空间分离，增加存储管理的灵活性。地址空间和存储空间两个基本概念的定义如下：

地址空间：将源程序经过编译后得到的目标程序，存在于它所限定的地址范围内，这个范围称为地址空间。地址空间是逻辑地址的集合。

存储空间：指主存中一系列存储信息的物理单元的集合，这些单元的编号称为物理地址存储空间是物理地址的集合。

根据分配时所采用的基本单位不同，可将离散分配的管理方式分为以下三种：  
页式存储管理、段式存储管理和段页式存储管理。其中段页式存储管理是前两种结合的产物。

**页式存储管理**

基本原理: 将程序的逻辑地址空间划分为固定大小的页(page)，而物理内存划分为同样大小的页框(page frame)。程序加载时，可将任意一页放人内存中任意一个页框，这些页框不必连续，从而实现了离散分配。该方法需要CPU的硬件支持，来实现逻辑地址和物理地址之间的映射。在页式存储管理方式中地址结构由两部构成，前一部分是页号，后一部分为页内地址w（位移量）

**页式管理方式的优点是：**

**1）**没有外碎片，每个内碎片不超过页大比前面所讨论的几种管理方式的最大进步是，

  2）一个程序不必连续存放。

  3）便于改变程序占用空间的大小(主要指随着程序运行，动态生成的数据增多，所要求的地址空间相应增长)。

**缺点**是：要求程序全部装入内存，没有足够的内存，程序就不能执行。

页式和段式管理的区别

 页式和段式系统有许多相似之处。比如，两者都采用离散分配方式，且都通过地址映射机构来实现地址变换。但概念上两者也有很多区别，主要表现在：

  1)、需求：是信息的物理单位，分页是为了实现离散分配方式，以减少内存的碎片，提高内存的利用率。或者说，分页仅仅是由于系统管理的需要，而不是用户的需要。段是信息的逻辑单位，它含有一组其意义相对完整的信息。分段的目的是为了更好地满足用户的需要。

  一条指令或一个操作数可能会跨越两个页的分界处，而不会跨越两个段的分界处。

 2)、大小：页大小固定且由系统决定，把逻辑地址划分为页号和页内地址两部分，是由机器硬件实现的。段的长度不固定，且决定于用户所编写的程序，通常由编译系统在对源程序进行编译时根据信息的性质来划分。

3)、逻辑地址表示：页式系统地址空间是一维的，即单一的线性地址空间，程序员只需利用一个标识符，即可表示一个地址。分段的作业地址空间是二维的，程序员在标识一个地址时，既需给出段名，又需给出段内地址。

4)、比页大，因而段表比页表短，可以缩短查找时间，提高访问速度。

**学期末心得体会**

转眼间，学习了一个学期的计算机操作系统课程即将结束。在这个学期中，通过老师的悉心教导，让我深切地体会到了计算机操作系统的一些原理和具体操作过程。在学习操作系统之前，我只是很肤浅地认为操作系统只是单纯地讲一些关于计算机方面的操作应用，并不了解其中的具体操作过程和实用性。通过这一学期的学习，我知道了操作系统(Operating System，简称OS)是管理计算机系统的全部硬件资源包括软件资源及数据资源；控制程序运行；改善人机界面；为其它应用软件提供支持等，使计算机系统所有资源最大限度地发挥作用，为用户提供方便的、有效的、友善的服务界面。

黄老师的课程可以说是非常的硬核，英文版教材、课件和作业一度成为了我学习的障碍。但是我坚持阅读外文文献之后，也对课程平台材料的建设提出了一些建议。**比如第一个实验 fork 系统，**有微软工程师提出弃用它的观点，我在查阅了大量参考文献之后将之贡献给了课程平台。

经过一个学期的学习，我也知道了计算机操作系统是铺设在计算机硬件上的多层系统软件，不仅增强了系统的功能，而且还隐藏了对硬件操作的细节，由它实现了对计算机硬件操作的多层次的抽象。以下是我这一学期的学习，对操作系统的一些原理联系生活现象的一些理解和心得，这里我举两个小例子：

1. 进程的同步问题：

例如公交车司机和售票员，司机负责开车、进站、开车门、关车门与进站； 售票员负责售票、进出站后观察车门，给司机发开关门信息；正常状态下，各自活动，司机开车，售票员售票；在进出站时双方要进行进程同步。

这里的原理可以解释为：一个进程到达了确定的点后，除非另一些进程已经完成了某些操作，否则不得不停下来等待另一进程为它提供的消息，早未获得消息前，该进程处于等待状态，获得消息后被唤醒处于就绪状态，这就是进程同步。

1. 生产者-消费者问题：

例如你要寄信给家人，这里分四个步骤：你把信写完把信放入邮箱邮递员把信从邮箱取出邮递员把信送去邮局。这四个步骤正好对应生产者制造数据生产者把数据放入缓冲区消费者把数据取出缓冲区消费者处理数据。

单单抽象出生产者和消费者，够不上是生产者-消费者问题。该问题需要在生产者和消费者之间有一个缓冲区，生产者把数据放入缓冲区，消费者从缓冲区中取出数据。

1. **参考文献**

《Operating System Concepts》7th Edition, 高等教育出版社

《操作系统概念》第六版中文版，高等教育出版社，第九章，郑扣根译

《计算机操作系统（第三版）》（西安电子科技大学出版社）

http://blog.csdn.net/windowseight/article/details/8279863

http://bbs.chinaunix.net/thread-2083672-1-1.html

http://duartes.org/gustavo/blog/post/memory-translation-and-segmentation

http://blog.csdn.net/erazy0/article/details/6457626#comments

<http://blog.csdn.net/drshenlei/article/details/4261909>

<https://www.cnblogs.com/huangwentian/p/7487670.html>