**OS第三次实验报告**

——vmm实验

目录

[一、 实验成员及分工 3](#_Toc450252536)

[二、 实验目的 3](#_Toc450252537)

[三、 需求说明 3](#_Toc450252538)

[四、 设计详情 5](#_Toc450252539)

[五、 实验感想 11](#_Toc450252540)

## 实验成员及分工

实验组长：张诚——完成基本要求，多级页表以及程序整合

实验组员：何玥——部分提高要求

柴舜——部分提高要求

周啸辰——实验会议记录以及实验报告撰写

## 实验目的

1、了解Linux的内存管理机制

2、掌握页式虚拟存储技术，理解虚地址到实地址的定位过程

3、掌握最不频繁使用淘汰算法，即LFU页面淘汰算

## 需求说明

#### 3·1基本要求

通过本实验，要求学生能够了解Linux系统下页式存储管理机制，并实现一个简单的虚存管理模拟程序具体要求如下：

1、设计并实现一个虚存管理模拟程序，模拟一个单道程序的页式存储管理，用一个一维数组模拟实存空间，用一个文本文件模拟辅存空间

2、建立一个一级页表

3、 程序中使用一个函数do\_request()随机产生访存请求，访存操作包括读取、 写入、执行三种类型

4、实现一个函数do\_response()响应访存请求，完成虚地址到实地址的定位及 读/写/执行操作，同时判断并处理缺页中断

5、实现LFU页面淘汰算法

#### 3·2提高要求

1. 建立一个多级页表

2. 实现多道程序的存储控制

3. 将do\_request()和do\_response()实现在不同进程中，通过进程间通信 （如FIFO）完成访存控制的模拟

4. 实现其它页面淘汰算法：如页面老化算法、最近最久未使用淘汰算法 （LRU）、最优算法（OPT）等

### 设计详情

#### 4·1基本实验要求实现情况

对于这次实验而言，代码量虽然较多，但是相比之前的工作而言，源代码中的BUG相对较少。小组成员在第一次会议的时候对程序进行了较为完整的讨论，因此程序理解的时间并不是很长，只是在一开始理解程序的时候发现在vmm.c文件中有一处辅存的代码那里有一个小细节：

pageTable[i].auxAddr = i \* PAGE\_SIZE \* 2;

这句代码在最初讨论的时候觉得比较费解，因为在学习代码的过程中我们发现这个地方并不需要乘2。最开始我们认为是因为无符号数的问题需要将数字翻倍，但后来仔细研究发现并不需要这样操作，最终我们认定这个乘2是一个不需要的部分，遂将其删去，这是我们在实验开始时遇到的第一个比较棘手的问题。

由于在本次实验中，基本要求在源代码中基本都实现了，因此我们组将绝大部分的时间用于研究提高要求的实现和优化上。



对于提高要求，将请求输入过程从主程序中分离出来，作为独立的程序。这一程序可以开多个，每一个都代表一个不同的进程，这样就实现了第二个提高要求。首先开主程序，主程序打印一遍二级页表后就会进入等待状态，一旦有请求进程启动，就会向主进程发送信号，主进程会给该请求进程一个编号，这个编号就是指导书中提到的processNum。获得编号后，请求进程才可以工作。然后可以在请求进程中输入请求，主进程会相应。如果请求的目标页未被使用，则指定该页属于当前请求进程（即设置页表项的proccessNum属性值为请求进程的编号），以后除非该请求进程退出，否则其他进程都无法访问该页。

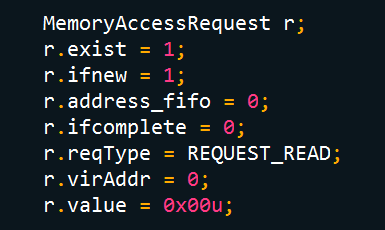
实现多级页表后，原页表成为二级页表，添加一个新页表作为一级页表，该页表共有8页，每一页对应二级页表中的8页。一级页表的页表项只有页号和装入两个属性，页号即0~7，装入代表每一页对应的8页二级页表是否装入。因为设置多级页表的目的在于节省空间，故二级页表并不是一开始全部处于装入状态，对于没有被装入的二级页表项，只有收到对应的请求之后才会将其装入，由于是模拟程序，各页表和实存是独立的数组存储，故装入过程并没有模拟二级页表添加到实存的过程，只是把一级页表项的装入位置为TRUE。

在实现第三个附加实验中，为了实现进程间的通信和多道任务同时进行，我们考虑了如下一种更加真实的虚存访问模型：一个分配的主进程（服务端）+n个虚存访问的进程（客户端），将do\_request()函数单独写一个方法，并通过进程间通信完成多个客户端和服务端的信息通信。

基于此，我们实现并设计了更加有效的真实的访存模拟：新的进程访问虚存，会首先申请一个设备号，服务端会在未分配的设备号中给出一个最小的设备号，分配给进程。进程获得设备号后才能发送请求（请求中标明设备号）；如果服务端达到服务上线，没有能够使用的设备号，则告知设备，设备会等待一段时间后，再次尝试访问设备，直到可以占用此设备号。进程结束后，会告知服务端，服务端会回写该进程涉及的内存内容，回收设备号和相关的页表项。

进程之间的通信是通过FIFO实现。服务端先建立两个管道：一个用来读，一个用来告知进程是否有设备号，设备号是多少。使用经典的非阻塞读和阻塞写的方式，客户端先发送标明新进程的信息，服务端读取后，查找是否有可用的设备号并返回设备号（或者没有可用的信息），客户端如果没有收到可用的设备号，反复询问服务端，否则记录自己的设备号，并发送虚存的访问请求，或者结束信息。

在实现过程中修改了请求的数据结构：增加了是否为请求、是否为新进程、设备号、进程是否结束表征上述的属性。



通过单独编写do\_request()方法并用这种方式对实验进行模拟可以更加真实的模拟多用户系统进行进程管理时虚拟内存如何进行管理，并且这样的模拟方式效率更高，同时在和源代码vmm.c的配合使用中更加方便维护代码，提高了代码的可移植性和灵活性，实现了系统中进程上线的模拟。

在淘汰算法方面，我们选择了LRU算法，这个算法要求记录每个页表项最后被访问的时间，所以在页表项的数据结构中，加入一个参数，相对访问时间，来表示每个页表项最后被访问的时间，具体的实现方式如下：

1、某一物理块载入，然后该页表项的载入标记置1，其相对访问时间为1，其他未被使用的页表项相对访问时间为0，若此时有别的页表项载入，则将原先载入的页表项相对访问时间加1，新增页表项相对访问时间置1，以此类推。

2、在某页表项被访问时，将其相对访问时间记录并将其置1，然后将其他所有页表项相对访问时间+1。

3、产生缺页中断但仍有空闲的物理块时，将新载入的页表项相对访问时间置1，其他页表项相对访问时间+1。

4、产生缺页中断但没有空闲的物理块时，将相对访问时间最大的页表项替换，然后将其相对访问时间置1，其他页表项相对访问时间+1。

LRU算法采用类似栈的方法来实现，设置了一个栈，来存放使用过的页表项，若某一页表被使用过，则将其从栈中取出，然后放在栈顶。当产生缺页中断且物理块满时，将栈底的页表项的内容替换，从而达到实现LRU算法的目的。

在本次试验中把一个一个的页表项放在一个专门的栈中比较麻烦，所以决定在页表项的数据结构中加入了栈标记，表示这个页表项在栈中的位置，来模拟栈，大致的过程是这样的: 将某一物理块载入，然后该页表项的载入标记置1，其栈标记为1，其他未被使用的页表项栈标记为0，若此时有别的页表项载入，则将原先载入的页表项栈标记加1，新增页表项栈标记置1，以此类推。在某页表项被访问时，将其栈标记记录并将其置1，然后将所有栈标记小于该栈标记的页表项栈标记+1。产生缺页中断但仍有空闲的物理块时，将新载入的页表项栈标记置1，其他页表项栈标记+1。产生缺页中断但没有空闲的物理块时，将栈标记最大的页表项替换，然后将其栈标记置1，其他页表项栈标记+1。

通过这种方式完全模仿栈，有多少个已载入页表项，栈标记最大就会这么大，然而后来发现这样的设计不是很轻松，写起来像是带着镣铐，尤其是在某页表项被访问时，要将被访问的页表项栈标记记录并将其置1，然后将所有栈标记小于该栈标记的页表项栈标记+1这样其实只是为了保证有多少个已载入页表项，栈标记最大就会这么大，并且防止栈标记大小相同的情况或顺序中断的情况出现。比如共有23个页表项被使用，14号页表被访问，则前13个页表项的顺序提前，14号置1，如果将所有的除14号之外的所有页表项全部置1 ，则会出现1，2…14，16，….24，这样的顺序，这明显是不符合“栈”的情况的。所以要将所有栈标记小于它的页表项栈标记+1，总是要维护栈的顺序合理。

其实如果回归LRU算法的本质，就是记录每个页表项最后被访问的时间，其实如果直接将栈标记理解为访问的相对时间，这个镣铐其实就被打开了，这样想其实就可以不用考虑栈的逻辑内容，还拿刚才的情况举例，如果将所有的除14号之外的所有页表项全部置1 ，则出现1，2…14，16，….24，这样的顺序也无所谓，把这个顺序理解为相对访问时间的顺序，也能实现LRU算法，这样一来不需要考虑具体的顺序如何，栈中有多少个逻辑块，直接在进行操作之后，把所有的没有被访问的页表向相对时间+1。这样之前的顾虑就完全没有了，而且实现起来也比较简单，设计就变成了现在的设计。其实兜了个圈子，还是回到了这个算法的本质上。

### 实验感想

本次实验，我们模拟了虚拟内存的管理。经过了前面两次实验的锻炼，我们对实验的学习态度越来越主动，从开始的尽量达到实验的全部要求，到现在的希望能够以实验为基础，积极地从更加广泛的视角发现实验中的问题，解决问题，达到更好的锻炼效果。这也增强了我们团队的凝聚力和协作能力。以本次试验为例，我们小组通过多次正式会议和非正式会议，大家发散讨论，最终能够解决实验中发现的问题，也加入了我们自己的想法，提高实验效果。实验结果是成功的，过程是不平坦的。实验中，我们会发现问题，解决问题中又发现新的问题。有些问题，更是深深地困扰了我们。不过，通过查阅资料和小组学习，我们积累了基础知识，加上不断地实践，最终解决了问题。

这次实验内容并不是非常难，但不同要求之间存在较高重合度，导致整合多个人的代码时产生各种意想不到的冲突。此外，有些后来实现的要求可能会导致原先已经实现的要求出问题。总之，在团队合作中，任务的人员分配和时间顺序都很重要，对原程序改动较大的任务应该放在较后执行，重合度较高的多个任务最好由一个人承担，或者多人协商。

在对知识进行深化理解的同时，我们在合作学习和分工工作中获得了更多的经验，尤其在这次工作连贯性较强的流程下组员都能按时保质保量的完成工作，这种踏实认真的工作态度也是我们这次实验的巨大收获，希望能在日后的学习中将这种工作精神发扬光大，推动我们继续进步。