|  |  |  |  |
| --- | --- | --- | --- |
| **武汉大学国家网络安全学院教学实验报告** | | | |
| 课程名称 | 操作系统及安全设计 | 实验日期 | 2019/11/1 |
| 实验名称 | 认识保护模式（二）：分页 | 实验序号 | 三 |
| 1. 实验目的及实验内容   （本次实验所涉及并要求掌握的知识；实验内容；必要的原理分析） | | | |
| **实验目的：**  掌握内存分页机制  对应章节：3.3  **实验内容：**  1.认真阅读章节资料，掌握什么是分页机制  2. 调试代码，掌握分页机制基本方法与思路  – 代码3.22中，212行---237行，设置断点调试这几个循环，分析究竟在这里做了什么？  3. 掌握PDE，PTE的计算方法  – 动手画一画这个映射图  4. 熟悉如何获取当前系统内存布局的方法  5. 掌握内存地址映射关系的切换  – 画出流程图  6. 基础题：依据实验的代码，  – 自定义一个函数，给定一个虚拟地址，能够返回该地址从虚拟地址到物理地址的计算  过程，如果该地址不存在，则返回一个错误提示。  – 完善分页管理功能，补充alloc\_pages, free\_pages两个函数功能  7. 进阶题（选做）  – 设计一个内存管理器，选择其一实现：首次适应算法、最佳适应算法、伙伴算法，要  求实现内存的分配与回收。（提示，均按照页为最小单位进行分配、对于空闲空间管  理可采用位图法或者双向链表法管理）  **完成本次实验要思考的问题：**  1. 分页和分段有何区别？在本次实验中，段页机制是怎么搭配工作  的？  2. PDE、PTE，是什么？例程中如何进行初始化？CPU是怎样访问  到PDE、PTE，从而计算出物理地址的？  3. 为什么PageTblBase初始值为2M+4K？  4. 怎么读取本机的实际物理内存信息？  5. 如何进行地址映射与切换？  6. 如何实现alloc\_pages,free\_pages  7. 首次适应/最佳适应/伙伴算法，在本实验中应该怎么来实现？  （进阶） | | | |
| 1. 实验环境及实验步骤   （本次实验所使用的器件、仪器设备等的情况；具体的实验步骤） | | | |
| **实验环境：**  VMwareWorkstationPro 15.5.0  Ubuntu 12.04.5 desktop i386 32位  bochs 2.6.9  **关键技术：**  内存分页机制  **实验步骤：**  **1. 分页和分段有何区别？在本次实验中，段页机制是怎么搭配工作**  **的？**  页是信息的物理单位，分页是为实现离散分配方式，以消减内存的外零头，提高内存的利用率。或者说，分页是出于系统管理的需要而不是用户需要。  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。  分页机制是 80x86 内存管理机制的第二部分。它在分段机制的基础上完成虚拟地址到物理地址的转换过程。分段机制把逻辑地址转换成线性地址，而分页机制则把线性地址转换成物理地址。  所谓“页”，就是一块内存，在80386中，页的大小是固定的4096字节（4KB）。    本次实验中在GDT定义了两个段descriptor，定义两个段，分别存放页目录表和页表。对于pmtest6.asm，页目录表4kB，页表4mB。PageDirBase和PageTblBase是两个宏，指定了页目录表和页表在内存中的位置。在段中先对PDE,PTE，cr0，cr3初始化，从而实现了分页机制。  **2. PDE、PTE，是什么？例程中如何进行初始化？CPU是怎样访问**  **到PDE、PTE，从而计算出物理地址的？**  页目录表的表项简称 PDE（Page Directory Entry），页表的表项简称PTE（Page Table Entry）。 PDE高20位为页表基址，PTE高20位为页基址。低12位为属性。    **pmtest6.asm中，206-216初始化页目录表。**  第207行和第208行将段寄存器es对应页目录表段，下面让edi等于0，于是es:edi就指向了页目录表的开始。  第214行的 指令stosd第一次执行时就把eax中的PageTblBase|PG\_P|PG\_USU|PG\_RWW存入了页目录表的第一个PDE。 然后edi+4  .1循环，每次eax+4096然后写入edi当前指向的位置(PDE),然后dei+4指向下一个PDE    **然后初始化PTE**  与PDE类似    **然后初始化cr3指向页目录表。然后设置cr0的PG(开启分页机制)**      cr3又叫做PDBR（Page-Directory Base Register）。它的高20位将是页目录表首地址的高20位，页目录表首地址的低12位会  是零，也就是说，页目录表会是4KB对齐的。类似地，PDE中的页表基址（PageTable Base Address）以及PTE中的页基址（Page  Base Address）也是用高20位来表示4KB对齐的页表和页。  **CPU访问PDE,PTE,计算物理地址:**  先是从由寄存器cr3指定的页目录中根据线性地址的高10位得到页表地址，然后在页表中根据线性地址的第12到 21位得到物理页首地址，将这个首地址加上线性地址低12位便得到了物理地址。    使用magic break可以对pmtest6.asm调试      –stosd  •将eax的内容存储到es:edi指向的内存单元中，同时edi的值根据方向标志的  值增加或者减少（4）  •相应的还有stosb,stosw  **3.为什么PageTblBase初始值为2M+4K？**  因为设置页目录表起始位置为2M，然后页目录表占4K，然后页目录表与页表在内存中相邻，所以是2M+4K  **4.怎么读取本机的实际物理内存信息？**  利用中断15h。  **先填充如下寄存器：**  eax int 15h可完成许多工作，主要由ax的值决定，我们想要获取内存信息，需要将ax赋值为0E820h。  ebx 放置着“后续值（continuation value）”，第一次调用时ebx必须为0。  es: di 指向一个地址范围描述符结构ARDS（Address Range Descriptor Structure），BIOS将会填充此结构。  ecx es:di所指向的地址范围描述符结构的大小，以字节为单位。无论es:di所指向的结构如何设置，BIOS最多将会填  充ecx个字节。不过，通常情况下无论ecx为多大，BIOS只填充20字节，有些BIOS忽略ecx的值，总是填充20字节。  edx 0534D4150h（'SMAP'）──BIOS将会使用此标志，对调用者将要请求的系统映像信息进行校验，这些信息会被 BIOS放置到es:di所指向的结构中。  **中断调用之后，结果存放于下列寄存器之中。**  CF CF=0表示没有错误，否则存在错误。  eax 0534D4150h（'SMAP'）。  es: di 返回的地址范围描述符结构指针，和输入值相同。  ecx BIOS填充在地址范围描述符中的字节数量，被BIOS所返回的最小值是20字节。  ebx 这里放置着为等到下一个地址描述符所需要的后续值，这个值的实际形势依赖于具体的BIOS的实现，调用者不必 关心它的具体形式，只需在下次迭代时将其原封不动地放置到ebx中，就可以通过它获取下一个地址范围描述符。如果 它的值为0，并且CF没有进位，表示它是最后一个地址范围描述符。  上面提到的地址范围描述符结构（Address Range Descriptor Structure）如表3.5所示。    由上面的说明，ax=0E820h时调用int 15h得到的不仅仅是内存的大小，还包括对不同内存段的一些描述。而且，这些描述都被保存在一个缓冲区中。所以，在我们调用int 15h之前，必须先有缓冲区。我们可以在每得到一次内存描述时都使用同一个缓冲区，然后对缓冲区里的数据进行处理，也可以将每次得到的数据放进不同的位置，比如一块连续的内存，然后在想要处理它们时再读取。  **pmtest7.asm：**  定义了一块256字节的缓冲区（pmtest7.asm第65行），它最多可以存放12个20 字节大小的结构体。我们现在还不知道它到底够不够用，这个大小仅仅是凭猜测设定。我们将把每次得到的内存信息连续写入这块 缓冲区，形成一个结构体数组。然后在保护模式下把它们读出来，显示在屏幕上，并且凭借它们得到内存的容量。  **得到内存信息并写入缓冲区：**      **添加显示：**  pmtest7.asm 305-347  一个循环，循环的次数为地址范围描述符结构（下文用ARDStruct代替）的个数，每次循环将会读取一个ARDStruct。首先打印其中每一个成员的各项，然后根据当前结构的类型，得到可以被操作系统使用的内存的上限。结果会被存放在变量 dwMemSize中，并在此模块的最后打印到屏幕。  其中新添加了DispInt和DispStr等函数。它们用来方便地显 示整形数字和字符串。而且，为了读起来方便，它们连同函数DispAL、DispReturn被放在了lib.inc中，并且通过如下语句包含进 pmtest7.asm中：  %include "lib.inc"  238 **push** szMemChkTitle  239 **call** DispStr  240 **add** esp, 4  241  242 **call** DispMemSize ; 显示内存信息  在调用它之前，我们还显示了一个字符串作为将要打印的内存信息的表格头。  之后pmtest7.com运行如图    其中内存段意义    这里RAMSIZE是01FF000H，31.9375MB。  我们除了得到了内存的大小，还得到了可用内存的分布信息。 由于历史原因，系统可用内存分布得并不连续。  得到内存是为了节约使用，不再初始化所有PDE和所有页表。现在，我们已经可以根据内存大小计算应初始化多少PDE以及多少页表。  **修改setuppaging**  在函数的开头，用内存大小除以4MB来得到应初始化的PDE的个数（同时也是页表的个数）。（4096B/4=1024,1024\*1024\*4KB=4MB。一个PDE的空间）  在初始化页表的时候，通过 刚刚计算出的页表个数乘以1024（每个页表含1024个PTE）得出要填充的PTE个数，然后通过循环完成对它的初始化。 这样一来，页表所占的空间就小得多，在本例中，32MB的内存实际上只要32KB的页表就够了（书中是32MB，实际测试31.9375MB，但向上取整，/4还是8.）  所以在GDT中，这样初始化页表段：  LABEL\_DESC\_PAGE\_TBL: Descriptor PageTblBase, 4096\*8-1,DA\_DRW  这样，程序所需的内存空间就小了许多。     1. **如何进行地址映射与切换？**   通过改变cr3来转换地址映射。改变cr3从而切换页目录表，从而切换页表，从而使得同一个线性地址映射到不同的物理地址。  pmtest8.asm：  先执行某个线性地址处的模块，然后通过改变cr3来转换地址映射关系，再执行同一个线性地址处的模块，由于地址映射已经改变，所以两次得到的应该是不同的输出。  映射关系转换前的情形如图3.34所示。 开始，我们让ProcPagingDemo中的代码实现向LinearAddrDemo这个线性地址的转移，而LinearAddrDemo映射到物理地址空间中 的ProcFoo处。我们让ProcFoo打印出红色的字符串Foo，所以执行时我们应该可以看到红色的Foo。    随后我们改变地址映射关系，变化成如图3.35所示的情形。 页目录表和页表的切换让LinearAddrDemo映射到ProcBar（物理地址空间）处，所以当我们再一次调用过程ProcPagingDemo 时，程序将转移到ProcBar处执行，我们将看到红色的字符串Bar。    **在pmtest7.asm的基础上修改：**  将页目录表和页表放到一个段，同时在此段中增加一套页表页目录表。  两组页目录表和页表分别由SetUppaging和PSwitch初始化。    为了操作方便，新增加一个段flat，其线性地址空间为0～4GB。由于分页机制启动之前线性地址等同于物理地址，所以通过这个段可以方便地存取特定的物理地址。两组页目录表和页表都存在Flat段。  段flat有两个描述符SelectorFlatC和SelectorFlatRW。  因为不仅仅要读写这段内存，而且要执行其中的代码，而这对描述符的属性要求是不一样的。这两个段的段基址都是0，长度都是4GB。  修改启动分页的代码（SetupPaging），存储页表个数。然后PSwitch再次初始化页表时就按照PageTableNumber的个数初始化相同数目的PDE。    在整个初始化页目录和页表的过程中，es始终为SelectorFlatRW。存取物理地址的时候，将PDE或PTE地址赋值给edi，那么es:edi指向的PDE和PTE指向的就是相应物理地址。  es为基址，edi为偏移。Stosd将eax赋值给es:edi指向的地址。这样es:edi存储了物理地址。Flat段基址+PageDirBase0，就是页目录表的起始位置。  初始化页表也是同样的道理。    **增加函数PagingDemo，调用各个和分页有关的函数。同时填充代码至F4（见下文）**    程序的实现中有4个要关注的要素，分别是ProcPagingDemo、LinearAddrDemo、ProcFoo和ProcBar，称为F4。  ProcPagingDemo调用LinearAddrDemo，然后地址映射到ProcFoo和ProcBar，执行ProcFoo和ProcBar所在处的代码，显示Foo和Bar。  F4虽然都是当做函数来使用，但实际上却都是内存中指定的地址。我们把它们定义为常量。（然后把代码复制到四个地址，执行，相当于函数）代码填充进这些内存地址的代码就在PagingDemo中。其中用到了名为MemCpy的函数，它复制三个过程到指定的内存地址，类似于C语言中的memcpy。它假设源数据放在ds段中，而目的在es段中。所以在函数的开头，需要分别为ds和es赋值。函数MemCpy也放进文件lib.inc。    程序开始时LinearAddrDemo指向ProcFoo并且线性地址和物理地址是对等的，所以 LinearAddrDemo应该等于ProcFoo。而ProcFoo和ProcBar应该是指定的物理地址，所以LinearAddrDemo也应该是指定的物理地址。 因此，我们使用它们时应该确保使用的是FLAT段，即段选择子应该SelectorFlatC或者SelectorFlatRW。  我们先写两个函数foo和bar，在程序运行时将这两个函数的执行码复制到ProcFoo和ProcBar所在的地址。（PagingDemo完成）  ProcPagingDemo要调用FLAT段中的LinearAddrDemo，因为不想使用段间转移，我们需要把ProcPagingDemo也放进FLAT段中。写一个函数PagingDemoProc，然后把代码复制到ProcPagingDemo处。  代码PagingDemo大部分语句是内存复制工作。    代码最后的4个call指令。它们首先启动分页机制（SetupPaging），然后调用ProcPagingDemo()，再切换页目录（PSwitch），最后又调用一遍ProcPagingDemo。  由于LinearAddrDemo和ProcFoo相等，并且函数 SetupPaging建立起来的是对等的映射关系（线性地址=物理地址），所以第一次对ProcPagingDemo的调用地址映射到ProFoo。然后PSwitch后修改了LinearAddrDemo，指向Procbar。  PSwitch前面初始化页目录表和页表的过程与SetupPaging差不多，程序增加了改变线性地址LinearAddrDemo对应的物理地址的语句。改变后，LinearAddrDemo将不再对应ProcFoo，而是对应ProcBar。  同时把cr3的值改成了PageDirBase1，映射切换过程宣告完成。  修改后的线性地址高十位为页表在页目录表中的位置，中间十位为ProcBar页对应页表项在页表中的位置，偏移不变仍为000H。并非修改线性地址而是修改对应表项。  389 行Dword就把对应页表项修改为ProcBar的地址  ‘    运行看到红色的Foo和Bar，说明页表切换成功     1. **如何实现alloc\_pages,free\_pages**   **先实现计算线性地址到物理地址:**  设计函数Checkadd，检查切换页目录后地址映射过程：  见注释。计算的同时将中间结果（PDE,PTE等打印到屏幕）    在PSwitch后直接call Checkadd    效果如图：  检查错误：当最后一行，倒数第二个数据大于RAM size时，说明映射错误    实现alloc\_pages：  alloc\_pages用于连续物理内存的分配  struct page \*alloc\_pages(gft\_t gfp, unsigned int order)  alloc\_pages函数用于分配2^order个 ****连续**** 的物理页. 分配失败返回NULL。 **伙伴系统算法** 在实际应用中，经常需要分配一组连续的页，而频繁地申请和释放不同大小的连续页，必然导致在已分配页框的内存块中分散了许多小块的空闲页框。这样，即使这些页框是空闲的，其他需要分配连续页框的应用也很难得到满足。为了避免出现这种情况，Linux内核中引入了伙伴系统算法(buddy system)。把所有的空闲页框分组为11个块链表，每个块链表分别包含大小为1，2，4，8，16，32，64，128，256，512和1024个连续页框的页框块。最大可以申请1024个连续页框，对应4MB大小的连续内存。每个页框块的第一个页框的物理地址是该块大小的整数倍。  假设要申请一个256个页框的块，先从256个页框的链表中查找空闲块，如果没有，就去512个页框的链表中找，找到了则将页框块分为2个256个页框的块，一个分配给应用，另外一个移到256个页框的链表中。如果512个页框的链表中仍没有空闲块，继续向1024个页框的链表查找，如果仍然没有，则返回错误。页框块在释放时，会主动将两个连续的页框块合并为一个较大的页框块。  实现free\_pages  ****void free\_pages(unsigned long addr, unsigned int order)**** 功能:释放逻辑地址addr开始的页面2^order次方个 addr:页面开始的逻辑地址 order:释放页面的个数2^order个  **7. 首次适应/最佳适应/伙伴算法，在本实验中应该怎么来实现？**  **（进阶）**  首次适应算法从空闲[分区表](https://baike.baidu.com/item/%E5%88%86%E5%8C%BA%E8%A1%A8/215102" \t "https://baike.baidu.com/item/%E9%A6%96%E6%AC%A1%E9%80%82%E5%BA%94%E7%AE%97%E6%B3%95/_blank)的第一个表目起查找该表，把最先能够满足要求的空闲区分配给作业，这种方法目的在于减少查找时间。为适应这种算法，空闲分区表(空闲区链)中的空闲分区要按地址由低到高进行排序。  实现：定义数据结构，包含内存大小，首地址，当前状态（是否被占用），然后用双向链表链接结构。从低地址到高地址链接。  在分配内存时，从链首开始顺序查找，直到找到一个大小能满足要求的空闲分区为止，然后再按照作业的大小，从该分区中划出一块内存空间分给请求者，余下的空闲分区仍停留在空闲链中。  当进程运行完毕释放内存，系统根据回收区的首址，从空闲区链表中找到相应的插入点，此时可能出现以下4种情况之一  1回收区与插入点的前一个空闲分区F1相邻接，此时将两个分区合并  2回收区与插入点的后一个空闲分区F2相邻接，此时将两个分区合并  3回收区与插入点的前，后两个空闲分区相邻接，此时将三个分区合并  4回收区既不与F1相邻接，又不与F2相邻接，此时应为回收区单独建立一个新表项 | | | |
| 1. 实验过程分析   （实验分工，详细记录实验过程中发生的故障和问题，进行故障分析，说明故障排除的过程及方法。根据具体实验，记录、整理相应的数据表格等） | | | |
| **遇到的问题及解决：**   1. 不清楚如何在dos单步调试程序   通过magic\_break，加入断点，实现单步调试。  **实验结果记录：**   1. 调试代码pmtest6.asm，掌握分页机制基本方法与思路        1. 调试pmtest7.com，熟悉如何获取当前系统内存布局的方法     3.调试pmtest8.asm 掌握内存地址映射关系的切换    4.自定义一个函数，给定一个虚拟地址，能够返回该地址从虚拟地址到物理地址的计算  过程，如果该地址不存在，则返回一个错误提示。 | | | |