实验报告



PA3-存储管理

学号 6319000359

姓名 张明君

|  |
| --- |
| 实验进度（任务自查表） |
| |  |  | | --- | --- | | 序号 | 是否已完成 | | 必做任务 1 | 是 | | 必做任务 2 | 是 | | 必做任务 3 | 是 | | 必做任务 4 | 是 | | 选做任务 1 | 是 | | 选做任务 2 | 是 | | 选做任务 3 | 否 |   **必做任务1:** **实现一级 Cache**  **选做任务1: 实现二级 Cache**  **必做任务2：在 NEMU 中实现分段机制**  **必做任务3：在 NEMU 中实现分页机制**  **必做任务4：实现 TLB**  **选做任务2：简易调试器**  **选做任务3: 为用户进程创建 video memory 映射** |
| 思考题（请注明题号，如思考题1，思考题2，. . .） |
| **思考题 1：GDT 能有多大**  你能根据段选择符的结构, 计算出 GDT 最大能容纳多少个段描述符吗？  **答案：**GDT是使用了数组的形式存储每一个段描述符的地址，又因为是32位基地址位数所以可以容纳2^32=4GB的段描述符。  **思考题 2：为什么是线性地址**  GDTR 中存放的 GDT 首地址可以是虚拟地址吗？为什么？  **答案：**GDTR 中存放的 GDT 首地址不可以是虚拟地址 因为GDTR全局唯一常驻内存中，所以一般放入到低地址常驻段内，使用绝对地址即可。  **思考题 3：如何提高寻找段描述符的效率?**  在上述4个步骤中, 如果段寄存器的内容没有改变, 前 3个步骤的结果都是一样的。注意到对 GDT 或 LDT 做索引是要访问内存的, 如果每次寻址都需要重复前 3 步, 就会产生很多不必要的内存访问。你能想到有什么办法来避免这些不必要的内存访问吗？请查阅 i386 手册, 对比一下你的想法和 80386 的实现是否一样。  **答案：**描述符的读取建立一个类似于高速缓存的数据结构来存储近期常使用的项，它能把少多次重复访问内存减小 及把时间开销降低。  **思考题 4：段式存储管理的缺点**  参考手册内容, 段式存储管理有什么缺点？(说不定考研会出这道题喔 ^\_^)  **答案：**段式存储 是直接将进程按照固定大小切成许多小部分存到了不同的页。它的缺点是大部分页要不存的全是代码，要不存的全是数据，但是总是会有几个页且在了数据和代码的交界处，这样这个页就会同时存储着数据和代码的混合体，这种页会可读性很差，不友好，导致用户编程不方便。  **思考题 5：页式存储管理的优点**  回忆课堂内容, 页式存储管理有什么优点？(说不定考研会出这道题喔 ^\_^)  **答案：**页式存储优点是没有外部碎片，可以将代码数据段切割成许多分区离散存储，而且不需要申请连续存储的空间，能提高了内存的利用率。这样的页式存储很适用于提高内存利用率方便数据的存储。  **思考题 6：一些问题**   * 80386 不是一个 32 位的世界吗, 为什么表项中的基地址信息只有 20 位, 而不是 32 位？ * 手册上提到表项(包括 CR3)中的基地址都是物理地址, 物理地址是必须的吗？能否使用虚拟地址或线性地址？ * 为什么不采用一级页表？或者说采用一级页表会有什么缺点？   **答案：**表项是要记录块号 它只是存储块号的。平常页内偏移量是要占12位的，而且地址是32为的， 所以块号只要20位。 那么个表项的基地址信息为20位。  手册上提到表项(包括 CR3)中的基地址都是物理地址, 物理地址是必须的。不能使用虚拟地址或线性地址 因为它会有变化。  不采用一级页表的原因是当数据块多的时候 一级页表会很大甚至装不下整个块号。这是它的缺点。 因此 采用二级页表甚至多级页表就可以经过层层块号的映射关系最终存储更多的块号。  **思考题7 ：空指针真的是“空”吗？**  程序设计课上老师告诉你, 当一个指针变量的值等于 NULL 时, 代表空, 不指向任何东西。仔细想想, 真的是这样吗？当程序对空指针解引用的时候, 计算 机内部具体都做了些什么？你对空指针的本质有什么新的认识？  **答案：**程序设计课上老师告诉你, 当一个指针变量的值等于 NULL 时, 代表空, 不指向任何东西。仔细想想, 不是真的这样的。空指针的定义是 和任何对象或函数的指针值都不相等，未分配或者 尚未指向任何地方的指针 那么并不是为NULL，只是一般使用上规定对于空指针需要赋值NULL来表示，其实空指针解引用时，因为空指针会指到VM的任何位置，碰到异常操作，比如对只读区写操作，就会引起硬件中断产生core，也就是通常的段错误，空指针解引用又称为空指针引用故障。  **思考题8：在扁平模式下如何进行保护**  现代操作系统一般使用扁平模式来"绕过"IA-32 分段机制。在扁平模式中, ring 0 和 ring 3 的段区间都是 [0, 4G) , 这意味着放在 ring 0 中的 GDT, 页表这些重要的数据结构对于处在 ring 3 的恶意程序来说竟是一览无余！扁平模式已经不能阻止恶意程序访问这些重要的数据结构了, 为何恶意程序仍然不能为所欲为？  **答案：**虽然恶意程序可以一览无余，但对于这其中的数据操作是需要在特权指令下完成的，即需要操作系统内核来完成，需要较高的指令权限才可以实现对齐的操作，对于一般的恶意程序，其为目态，无权使用特权指令，也就无法对这其中的数据进行恶意修改，当在目态下执行恶意修改的特权指令就会触发异常中断从而导致操作系统内核立刻阻止恶意行为对程序的破坏。  **思考题9：地址映射**  结合 kernel 的框架代码理解分页机制，阅读 init\_page()函数的代码, 它建立了一个从虚拟地址到物理地址的映射。请结合此处代码描述这个映射具体是怎么样的, 并尝试画出这个映射: 画两个矩形, 左边代表虚拟地址, 右边代表物理地址, 并标上 0 和 4G, 然后画出哪一段虚拟地址对应哪一段物理地址。你可以先用纸笔来画, 然后拍照片, 粘贴在实验报告中。  **答案：**我实现了 page 命令，截图如下：    我们可以看到虚拟地址的物理地址 0x00007fd8，0x0010015f等。  **思考题10：有本事就把我找出来！**  在 kernel/src/memory/kvm.c 中, 为了提高效率, 我们使用了内联汇编来填写页表项, 同时给出了相应的 C 代码作为参考。如果你曾经尝试用 C 代码替换内联汇编, 编译后重新运行, 你会看到发生了错误。事实上, 作为参考的 C 代码中隐藏着一个小小的 bug, 这个 bug 的藏身之术十分高超, 以至于几乎不影响你对 C 代码的理解。聪明的你能够让这个嚣张的 bug 原形毕露吗？  **答案：**这段代码的问题在于，pframe\_addr 是无符号类型，而 for 循环的循环条件是 pframe\_addr >= 0，这个式子显然成立（无符号数当然大于等于 0）。在循环的最后，在对 pframe\_addr 做减法运算时，会发生一次溢出，导致循环永远不会结束。  **思考题11：暗藏杀机的分页机制（这 5 个问题都有一些难度哦~）**  重新编译并运行, 你发现了什么问题？请解释为什么在开启分页之前不能使用全局变量, 但却可以使用局部变量(在 init\_page() 函数中使用了局部变量)。 细心的你会发现, 在开启分页机制之前, init\_page() 中仍然使用了一些全局变量, 但却没有造成错误, 这又是为什么?   * 在刚刚调用 init\_page()的时候, 分页机制并没有开启。但通过 objdump 查看 kernel 的代码, 你会发现 init\_page()函数在 0xc0000000 以上的地址, 为什么在没有开启分页机制的情况下调用位于高地址的 init\_page()却不会发生错误?   **答案：**   1. **为什么在开启分页之前不能使用全局变量, 但却可以使用局部变量?**   不能使用全局变量是因为全局变量被分配在 bss 或 data 节中，而在建立页表、开启分页前，bss 或 data 节对应地址出根本没有内存与它对应。可以使用局部变量是因为，局部变量存储在栈上，是根据栈指针来寻址的，只要设置好栈指针（在 start.S 中设置好了），让栈指针指向的位置向下有一段空间，就可以正常使用局部变量了。   1. **仍然使用了一些全局变量, 但却没有造成错误, 这又是为什么?**   不能直接使用全局变量是因为，此时访问全局变量是使用建立映射后的地址去访问，而此时映射还没有建立起来，而如果仔细小心地用 va\_to\_pa()宏把映射后的地址转换为真正的物理地址，就可以使用全局变量了。   1. **为什么调用位于高地址的 init\_page()却不会发生错误?**   因为一般的函数调用都是通过“call rel32”指令完成的，而此指令是“相  对跳转”，由于建立映射前（即内核运行在低地址）和映射后（即内核运行在高地址）这两种状态下，各个函数之间相对的位置并不会发生变化。“相对跳转”并不会造成任何问题。   1. **在 init\_page() 的循环中有这样两行代码, 尝试注释掉其中一行, 重新编译 kernel 并运行, 你会看到发生了错误. 请解释这个错误具体是怎么发生的.**   **注释掉第一行：**  结合 NEMU 的错误信息和 log 可以看到，在刚开启分页后（即刚刚执行完mov %eax, %cr0（即置 CR0 的 PG 位），由于低地址部分没有建立映射，当前 EIP 指向的位置的内容消失掉了，NEMU 找不到下一条指令于是崩溃。  **注释掉第二行：**  结合 NEMU 的错误信息和 log 可以看到，在 kernel/src/main.c 文件的init()函数中，最后有个 jmp 指令用来让内核跳到高地址上去执行。但是由于高地址处根本没有建立映射，于是 NEMU 找不到下一条指令就崩溃了。   1. **在 init\_page() 函数中, 有一处代码用于拷贝 0xc0000000 以上的内核映射, 尝试注释这处代码, 重新编译 kernel 并运行, 你会看到发生了错误. 请解释这个错误具体是怎么发生的.**   init\_page()函数的作用是，把内核的页目录的高地址部分，拷贝到用户程序的页目录的高地址部分，即在用户程序的页表中的高地址部分建立与内核页表相同的映射。这样做，一是为了内核在切换到用户页表后，内核可以继续运行，不至于因为找不到内核代码而崩溃；二是为了在用户程序中，能够让用户程序陷入内核代码中（如果整个地址空间没有内核代码，CPU 在陷入内核的时候就找不到内核代码了）。如果注释掉这处代码，用户页表的高地址部分将是一片未映射的区域，会导致在 loader()函数中切换页表时，CPU 由于找不到下一条指令而崩溃。 |
| 实验遇到的问题、思考、解决办法（可以不填写） |
|  |
| 实验心得（可以不填写） |
|  |