JOS 设计文档（lab2范围内）
丁卓成 5120379064

一、概述

本文档为Lab1提交的设计文档之续，继续分析JOS的架构，本次Lab的主要内容是内存管理系统。

二、物理内存管理

（一）、检测物理内存的状态

首先要明确一个概念，此处的物理内存指的是我们的kernel能使用的Memory，即Physical Address中映射到RAM的部分，物理地址空间中的空洞即使到OS启动完毕后仍是无法利用的（因为会有硬件需要用到该部分空间）。物理地址空间被0xA0000-0xFFFFF的空洞分割成Low Memory和Upper Memory，该空洞是标准化了的，目前所有的PC上都有这个空洞。而在Upper Memory中也存在空洞，例如上课提到过在接近4G的地方有另一个空洞（其大小与具体的主板等有关，最大不超过1G，需检测才能得知），不过Upper Memory的布局并未标准化，各个厂商的主板其空洞的分布可能不同，空洞可能用于Memory Mapped Hardware或者ACPI Area。此外，历史上曾在15-16M的位置存在一个标准化的空洞(ISA Memory Hole)，但是ISA已被PCI和PCIe取代，理论上该空洞已不需要，不过仍有些主板允许该空洞存在，这也成了OS为了兼容所需要考虑的问题之一。

检测物理内存，推荐的方式是通过BIOS Service，而本Lab中使用的是从CMOS存储中获取内存信息。对于本次Lab而言，其区别在于，BIOS Service必须在实模式下调用，而CMOS存储可以通过端口操作，在保护模式亦可进行，因此在已经进入保护模式的情况下，为简便起见没有特地回到实模式去通过BIOS获取内存信息。这样做不是没有代价的，BIOS可以提供更准确、更详细的信息，而CMOS方式会忽略15M处的空洞、ACPI Table，并且当物理内存大于4G时无法检测到高于4G的内存区域（实际上目前只有通过INT 0x15,EAX=0xE820对应的BIOS功能才能获取关于高于4G内存区域的信息，这也是GRUB的首选方式，即使失败GRUB也会选择调用更旧一些的BIOS调用而不去选择CMOS方式）。当然，理论上还可以手动探测内存，就像Lab1中探测MDA Buffer可用还是CGA Buffer可用一样，但这样做可能会破坏一些重要的数据结构或者忽略了一些应避免使用的硬件映射区域，从实践的角度看这种做法是应极力避免的。

回到本次Lab，现在来介绍CMOS相关的知识。CMOS与RTC(Real-Time Clock)位于同一块芯片，在早期是摩托罗拉的MC 146818芯片，因此kclock.c中函数名以mc146818为前缀，CMOS中存储的BIOS配置信息以及RTC都要靠这块芯片上的电池供电以保证非易失性。CMOS通过端口0x70和0x71访问，不幸的是由于早期计算机资源有限，使用端口时非常节俭，0x70端口同时还被赋予了配置NMI(Non Maskable Interrupt)的功能。

NMI是一种中断，不过与通常的中断连接到CPU的方式不同（直接连接或通过不同的芯片控制），因而中断屏蔽指令并不能屏蔽NMI（故名Non Maskable），该中断作用是在硬件发生错误（内存错误或总线错误）时产生中断，或也可作为一个时钟中断。为了屏蔽该中断，必须向0x70端口输出一个字节，该字节的最高位表示是否屏蔽NMI（1为屏蔽0为允许）。同时，CMOS中的值可以逐字节存取，每个字节通常被称为register（因此kclock.c中函数参数名为reg），为了指定读写哪个register，需要向0x70端口输出一个字节来进行选择，随后可以对0x71端口读写一次（能且只能一次，试图第二次操作时该端口对应的register会复位到0xD）来操作所选择的register。因此，综合以上两点，用户在使用CMOS时，必须在每次选择register的同时指定NMI是否要屏蔽，在Lab的代码中，采取了允许NMI的策略。

这些Register中前14个用于RTC，其余的可归为NVRAM，其中0x15和0x16表示Low Memory的大小，0x17和0x18表示1M到16M或到64M的内存大小（以上两者均以K为单位），0x34和0x35（有时）表示16M到4G之间的内存大小（以64K为单位），且后两项数据是不够可靠的（至少于BIOS Service相比，因为不能识别内存空洞），尤其是当实际物理内存大小超过64M/4G时更是如此，我们不能因为Lab中使用了这种方式就想当然地以为这种方式是适宜的。0x17,0x18现在一般能识别到64M，而在早期只能识别到16M，在0x30,0x31处有其副本（即kclock.h中NVRAM_PEXTLO，其实那并不是第三项）。早期0x34及以后的Register是保留的，后来各厂商做了不同的扩展导致互不兼容，其中一种做法是在0x34,0x35存放16M以上内存大小，另一种做法则是存放若干Flag位，根据我的测试在QEMU中0x34,0x35是用于存放内存大小的。注意Lab中的i386_detect_memory函数，并未读取第三项0x34和0x35位置的数据，这导致JOS只能识别出64M左右的内存，比Lab1说明中所说的能利用前256M还小了好几倍。这可能是出于兼容性的考虑，因为并不是所有厂商的主板其CMOS的0x34,0x35都提供内存大小数据，但这导致其无法利用QEMU所模拟的BIOS的这项功能（经测试在QEMU 1.5版本中可以识别出最多3.5G内存，较64M翻了很多倍）。

（二）、物理页数据结构及其管理

为了控制页表的分配，设立了一个pages数组，成员均为struct Page类型，该类型的定义为一个Page*指针和一个uint16_t的引用计数。其具备指针域所以可以串成链表，我们使用指针page_free_list指向空闲链表的头，每次调用page_free(pp)就将pp加入空闲链表，每次调用page_alloc()就从空闲链表中取出一个Page*指针pp返回，提供了对于Page数据结构的基本操作。至于对其引用计数的修改以及真实页表的设置，则由其调用者完成，这两个函数只负责底层的数据结构操作。类似的，还有page_free_npages,page_alloc_npages,page_realloc_npages三个函数，实现连续物理页的分配释放（即从空闲链表取出放回），也是只负责管理数据结构，不负责实际操作页表。

（三）、页的映射和新页表的建立

为了进行内存管理，不止需要建立每个物理页对应的数据结构（主要是为了分配时方便地找到空闲的物理页），还要实现从虚拟地址到物理地址的映射。要建立一个映射，就要修改PTE，为此由pgdir_walk函数负责对给定的va找到相对应的PTE，若create参数不为零，则必要时可以创建新的Page Table以容纳PTE。注意如果Page Table不存在需要创建时，需要page_alloc申请一个物理页，并使用其物理地址设置对应的PDE，设置时的权限采用PTE_U | PTE_W | PTE_P。这是因为x86的页表翻译机制中，只有每级页表都具有U这个flag时，才允许用户态程序使用该内存（可读以及具有写权限时可写），也只有每级页表都具有W这个flag时，才允许写入（当具有内存使用权限时）。PDE的权限位均设为1，就使得我们可以只通过PTE上的权限位来管理权限，而忽略PDE上的权限设置。

有了PTE，page_insert函数即可创建一个新的映射。若要申请一块新的虚拟内存，通过先调用page_alloc找到空闲的物理页，再调用page_insert建立va到pa的映射即可。为了删除已有的页映射，需要使用page_remove函数，它首先通过page_lookup函数找到va对应的PTE以及struct Page，然后将PTE置零，Page的ref count减一，若减到零则使用page_free释放之，最后调用tlb_invalidate刷新TLB。其中tlb_invalidate函数是通过一条invlpg指令实现的，可以只flush掉指定的va对应的cache行而不影响其他cache行，无需把整个TLB都清空，提高了效率。

在系统启动期间，只能使用0-4M的物理内存，我们新建的页表也位于这4M之内。在mem_init函数中，首先是通过boot_alloc函数分配了一个页作为Page Directory，并设置了虚拟地址UVPT开始的4M为Virtual Page Table（即通过将UVPT对应的PDE指向Page Directory起到巧妙地将该地址开始的4M空间对应到全体页表的总和，每一项PTE可以直接通过vpt[PGNUM]的形式得到）。随后又通过boot_alloc分配了一块空间给pages数组。boot_alloc函数所分配的空间，是从kernel的ELF文件的内存中的映像的末尾开始，向高地址增长的，此外这个函数还记录了从映像末尾算起，当前最低的空闲地址，以便后面初始化Page结构。

到目前为止，剩余的空间已经可以全部用来分配给Page Table。我们在page_init函数里初始化了Page结构，将空闲的Page加入空闲链表，随后通过检查函数check_page_free_list(1)将0-4M的Page移到空闲链表的开头以便alloc时得到当前有效的Page。在这里有

趣的是用于检查Lab正确性的代码，同时也参与到了建立新页表的过程中来，而不是仅仅负责检查代码正确性。

接着，通过boot_map_region函数，我们绕过page_alloc和page_insert的方式，直接设置页表，这样保持了Page结构不变。用该函数映射了三部分，其一，是UPAGES开始的虚拟地址，映射到pages所在的页；其二，是KSTACKTOP开始向下KSTKSIZE大小的虚拟地址空间，映射到kern/entry.S中设置的bootstack位置；其三，将所有物理内存都映射到KERNBASE开始的虚拟地址空间。前两项由于其物理页已在使用中，且永远不会释放，故没有必要增加其引用计数；第三项由于这个映射只是为了管理物理内存方便，并不代表由任何进程使用了这些物理页，因此仍保持物理页的引用计数不变，否则将会使得空闲页不存在了。这样一来，boot_map_region直接设置页表的设计就显得十分合理了。

至此，新页表已经建立完毕，通过lcr3指令载入page directory基址后新页表即加载完成，随后设置了CR0中一些位的flag，mem_init就到此结束了，整个Lab 2的工作都体现在了mem_init函数中，此后进程的实现等等都要留待Lab 3及以后的Lab再做分析。整个过程中实际上并未用到此前实现的page_alloc_npages,page_insert等函数，因此需要一些check_xx函数来检测其功能是否已正确实现，穿插在正常设置页表的代码间。这些现在并未用到的函数，在Lab 3中应该会用到，并起到很关键的作用。

此外，值得一提的是，原本位于0x7c00-0x7dff的引导扇区内存放有GDT，在page_init里将0x7000-0x7fff这一页设置为了free，在建立新页表的过程中原先的GDT可能已被破坏。不过，由于cs、ds等寄存器有不可见的一部分用来cache段描述符，不需要每次内存读写时都读取GDT，这并不会造成系统的崩溃。但有一点是肯定的，不论原先的GDT是否被破坏，到Lab 3时，必须要建立新的GDT，区分内核态和用户态的段选择子，从而和这里新建的页表中的权限组合起来，提供完整的内存保护机制。