操作系统原理实验报告

|  |
| --- |
| **实验报告**  学院：数据科学与计算机学院 专业：计算机科学与技术 年级：2016级  实验人姓名（学号）：王锡淮（16337236）  日期： 2018 年 3月 24日 |
| **实验3 开发独立内核的操作系统**  **[实验目的]**   1. 学会引导程序、操作系统内核、用户程序三者的管理与安排。 2. 学会汇编程序与C程序编译器的组合。 3. 学会编写操作系统内核的基本功能。 4. 了解基本的编译原理。   **[实验要求]**   1. 编写引导程序和操作系统内核。 2. 该操作系统内核可以加载多个用户程序，并且和汇编模块相互调用。 3. 内核应具有以下功能： 4. 在磁盘上建立一个表，记录用户程序的存储安排。 5. 可以在控制台查到用户程序的信息，如程序名、字节数、在磁盘映像文件中的位置等。 6. 设计一种命令，并能在控制台发出命令，执行用户程序。   **[实验内容]**   1. 编写了一个命令行系统雏形，其具有以下指令： 2. Ls指令：查询程序的信息，包括程序名、字节数大小、在磁盘中所处的起始位置、文件类型（文档、可执行程序、文件夹）。 3. Date指令：显示当前的系统日期。 4. Clear指令：刷新屏幕。 5. Man指令：查看用户帮助文件。 6. Exit指令：退出操作系统. 7. reboot指令：重启操作系统。 8. ./+程序：运行用户程序。 9. 处理了内存的分布问题。 10. 了解磁盘上扇区的分布情况。 11. 带有屏幕自动滚动功能。 12. 带有回显、退格功能的键盘输入。   **[实验方案]**  实验工具以及环境  工具链：sublime text 3 编写代码，nasm汇编.asm文件，gcc将编译进行到汇编这一步，ld完成最后的链接，以实现c和汇编语言混合编译，然后是bochs调试，以及最后的vmware进行展示。此外，安装了dd for windows，以及ld连接器等linux下经过修改能在windows下运行的小工具。  实验环境：windows10系统。  方案思想  该操作系统雏形的设计参考的是windows下的powershell和linux下的bash，实现的是最简单的命令行指令，界面设计力求方便简洁。而进程设计则仍是采用的最简单的未运行态和运行态两态模型，而系统内也于实验2中保持一致，使用的是基于进程的系统内核，文件信息既可以采用放在磁盘中待程序读取并进行序列化，也可以硬编码在程序中，因为在编写man指令的时候就已经尝试过了前者序列化和反序列化的过程，于是采用的是在程序中硬编码的方法。用户程序也是基本使用的实验2中的用户程序。  本次实验的重点在于c语言和汇编语言的混合编译以及独立系统内核的编写，以及了解基本的编译原理、程序运行时的内存分布和磁盘的扇区分布。  实验原理  **如何将C语言和汇编语言混合编译**  在本次实验中使用的GCC编译器和NASM汇编器以及ld链接器。  想要使用gcc和nasm混合编译，首先了解二者的输出文件，nasm的默认输出是16位的程序，而gcc的默认输出是32位的程序，而我们知道编写32位的程序需要进入保护模式，但是本次实验要在实模式下完成，于是需要让最终的程序输出为16位程序。  于是需要了解让gcc编译产生16位代码的方法，查看gcc的用户手册[gcc]发现了一些指令，  [-march]指令，可以指定机器类型，使用-march=i386 可以指定使用的是intel i386CPU指令体系。  [-m32]指令，该指令指示int，long还有指针类型的长度为32位，并且生成的程序在i386机器上运行。  [-mpreferred-stack-boundary=num] 该指令使栈4字节对齐，这样方便16位和32位的数据共存于栈。  [-ffreestanding]指令，由于我们编写出来的程序是要在只有bios的裸机上运行的，所以是没有任何库能够使用的，于是用该指令显示地指明这一点。  [-c]指令，该指令使得gcc在编译过程“预处理-编译-汇编-链接”中执行到汇编这一步，输出.o文件。  除了在使用gcc时的选项设置之外，还要在.c文件开头加\_\_asm\_\_(".code16gcc\n");语句，该语句中的\_\_asm\_\_是gcc中的基本汇编使用方法，而.code16gcc指示gcc在汇编这一步生成16位代码。（其实以上-m32选项和\_\_asm\_\_(".code16gcc\n");语句的功能等于-m16选项）。  那么nasm的选项如何设置呢？因为gcc的默认输出文件是elf格式的，所以nasm文件也应输出elf格式的32位文件。使用的是[-f elf32]选项。  除此之外，还要在每个汇编文件首部加上[BITS 16]指示nasm该汇编文件是16位的。在该模式下，使用32位数据指令会被加上0x66前缀，指向32位的地址也被加上0x67前缀。  而至于链接这一步，使用ld工具，查看用户手册可知:  [-m i386pe]指示ld目标机器是i386pe  [-N] 是让.data和.text段不对齐  [-Ttext]和[-Tdata]的功能相当于org指令  以上内容参考自一个github项目[5]。  在看这段描述的时候可能产生疑惑，就是为什么要在命令行里指示生成32位，然后又在代码文本里面设置生成16位代码呢？其原因是我们在编写汇编代码的时候习惯于使用16位的写法，而在编写c语言时许多变量又是32位的，为了兼容，以上操作使得代码变为在nasm中使用16位变量而在gcc中使用32位变量的16位程序。  详细用法可以看我的run.bat批处理文件，展示如下：  屏幕剪辑  **如何编写汇编函数以便C语言调用（X86函数调用约定）**  CC遵循cdecl（C declaration，即C声明）是C语言的一种调用约定，在X86架构上，其内容包括：   1. 函数实参在线程栈上按照从右至左的顺序依次压栈。 2. 函数结果保存在寄存器EAX/AX/AL中 3. 浮点型结果存放在寄存器ST0中 4. 编译后的函数名前缀以一个下划线字符 5. 调用者负责从线程栈中弹出实参（即清栈） 6. 8比特或者16比特长的整形实参提升为32比特长。 7. 受到函数调用影响的寄存器（volatile registers）：EAX, ECX, EDX, ST0 - ST7, ES, GS 8. 不受函数调用影响的寄存器： EBX, EBP, ESP, EDI, ESI, CS, DS 9. RET指令从函数被调用者返回到调用者（实质上是读取寄存器EBP所指的线程栈之处保存的函数返回 地址并加载到IP寄存器）   除了以上内容，通过观察反汇编结果可以发现，gcc编译出来的汇编代码在调用C语言编写的程序时的确遵循了上述约定。会在函数被调用者其流程为：  1. 将ebp压入栈中。  2. 将esp的值赋给ebp。  3. 减小esp，申请栈空间。  4. 将参数从右往左压入栈中。  5. 执行函数流程。  6. 将ebp的值弹出。  7. 将ebp的值赋给esp。  8. 返回调用者。  对于ebp和esp的值的操作的目的在于分开在函数内部使用的局部参数和调用者申请的参数。  但是在调用汇编语言编写的程序时，gcc在调用者部分所做工作为：  1. 然后将函数从右向左压入栈中。  2. 随后用call指令调用函数。  并无保存ebp寄存器的值和恢复ebp寄存器的值的过程。所以在被调用的函数中自己注意维护栈，自己注意寻找返回地址的位置。  于是，如果要使用汇编编写C语言可以调用的函数，需要注意  1. 函数名添加下划线。  2. 返回值保存在eax中。  3. 注意栈中返回地址在哪个位置。  4. 要使用传递的参数，需要自行在栈中寻找，同时需要注意参数对应大小。  **如何在汇编代码中调用C语言的函数**  如上一部分所言，函数调用者要完成的操作是在栈中申请足够大的空间，然后将参数从右往左压入栈中，并且在函数返回后清空栈中的参数。所以要在汇编代码中调用C语言的函数需要做的是：   1. 申请栈空间 2. 参数从右往左压入栈中 3. 调用函数 4. 清空栈中的函数   **如何调用BIOS使得计算机关闭[2][7]**  要使得计算机通过中断关闭，可以按顺序调用int 15h来关闭本系统服务、驱动器、其余所有的服务，相应的中断如下：  mov ax, 5301h ;function 5301h  xor bx, bx ;device id: 0000h (=system bios)  int 15h ;call interrupt: 15h  mov ax, 530eh ;function 530eh  mov cx, 0102h ;driver version  int 15h ;call interrupt: 15h  mov ax, 5307h ;function 5307h  mov bl, 01h ;device id: 0001h (=all devices)  mov cx, 0003h ;power state: 0003h (=off)  int 15h ;call interrupt: 15h  **如何获取并输出系统时间**  获得系统时间是一件较容易的是，既可以从CMOS的RAM里面获得[1]，也可以调用其中断int 1ah/04h获得，但是在将获得的BCD码转变成字符串输出的时候就要注意在X86中采用的是小端方式存放数据，低位字节在高位地址。  **在磁盘中扇区的排列**  在本次实验中要使用较多的扇区，必须知道扇区的排列方式[8]。  首先了解一下软盘的物理以及逻辑结构  以一块软盘为例，一块软盘有两个面，正面和反面（在中断调用中是两个磁头），每个面被80个圆环分成80个磁道（也叫柱面）（1-80），每个磁道由18个扇区（0-17）组成（注意在中断调用中标号从1开始），每个扇区的大小为512字节。所以一块软盘的大小为 2\*80\*18\*512 = 1474560 字节 = 1440KB = 1.44M。  以下是举例：  物理地址 逻辑扇区地址  0面0磁道 1扇区 0  0面0磁道 2扇区 1  ： ：  0面0磁道 18扇区 17  1面0磁道 1扇区 18  1面0磁道 2扇区 19  ： ：  1面0磁道 18扇区 35  ： ：  我们把逻辑扇区地址可以看成大小为1.44M的数组的下标，每个数组元素的大小为512字节。  所以我们希望能根据 面 磁道 扇区 找到和 逻辑扇区的对应的关系，这样就大大方便了我们的调用中断。  经过观察与验证，逻辑扇区（相对扇区）号与磁面、磁道、扇区的关系是：    **操作系统运行时内存的分配**  在编写本次实验时遇到了内存的分配问题，所以应当知晓内存中各部分的用处，如下图：    于是我们可以使用就只有自由内存区，同时还要注意地址不能超出16位。在本次实验中使用第一个自由内存区，即从0x0500开始存放系统内核和系统例程，而将用户程序放在0xc200开始的地方。  内存和磁盘的分配方案为：    程序流程和算法思想  程序流程图如下：    在各个指令中，调用用户程序指令执行流程较为复杂，其流程图上方右侧图。  程序的流程设计来源于对于windows下的powershell和linux下的bash的使用经验，指令设计也是取材于最常用的指令。  数据结构与程序模块功能  Os.asm中的系统内核主要部分：    本次实验实现的内核功能较为简单，但是齐全，用简单的条件判断以及循环逻辑就可以实现。  汇编语言编写的函数库，汇编语言编写的函数目录如下，其中主要是printSentence、ClearScreen、getInput、dispatch、getDate、roll函数需要加以说明。    PrintSentence函数：    该函数的功能是在dhdl指定的行列位置打印出message参数指定的字符串，打印长度指定为len参数，而其实现则主要是利用int 10h/13h中断。  ClearScrenn函数：    这清屏函数的实现方式和我在之前的两个实验中采用的不一样，之前采用的是暴力清屏的办法，往显存区域写内容，这个方法的缺点是光标不会跟着移动，所以采用了in10h /06h或int10h / 07h中断来实现。  getInput函数，该函数参考自[1]，实现较长，分开说明：    这是getInput函数的主要逻辑，其主逻辑部分如下图左侧图，其中使用了101个字节长的缓冲区和存储了子程序地址的子程序表，以及栈顶指针。该功能实现主要依赖将缓冲区当作栈的方法，对于普通的字符输入，将其压入栈顶，并将其在当前光标位置打印出来，随后将光标后移一位；对于特殊字符的输入，如果是回车键，则在字符串的最后一位补上0作为字符结尾，并且将缓冲区的地址返回给调用者；而如果是退格键，除非栈空，否则将栈顶回退，并将光标回退一格（如果到了边界则换行），并采用直接写显存的方式将要删除的字符覆盖，下图右侧是详细的处理过程    其中显示的处理函数如下：    Dispatch函数：    这是一个进程分派器，接受程序在磁盘中的起始地址和程序大小，计算出所在的磁头号、磁道号以及扇区号，然后计算出扇区数目，将其加载至0xe000处，并且运行该函数，在函数退出后返回调用者。  计算磁头、磁道、扇区号的方法如原理部分所言。  Getdate函数的功能是将系统的年月日时间转变成字符串来返回，一下以“年”为例子说明：    首先使用中断int 1ah/04h来获取时间，其中年份信息以BCD码的形式存储在cl中，要做的工作主要是将小端方式存放的年份变成正常格式，在以ASCII码的格式存入字符串。  Roll函数的功能是当光标超出底部边界的时候将屏幕向上滚动一行，有两种实现方式，第一种是调用bios自带的中断以实现这样的效果，第二种是从显存区域将当前界面的显示内容拷贝，再在上移一行之后的地方输出。此处展示较为简洁的第一种方式    这里的方式是将整个界面向上滚动一行。  C语言编写的函数库：    其中需要说明的是莫非C标准库实现的strlen、strcpy  Int2str和strcmp等字符串处理函数，以及采用散列技术的文件系统雏形。  实现的字符串处理函数：  这四个字符串处理函数逻辑是常见的逻辑，同时还是C语言是西安的，难度不高，只是有不少细节需要注意，比如字符串最后要加上0作为结尾。  采用散列技术的文件系统雏形：  在该文件系统中，文件的组织采用的是顺序文件组织，文件目录使用的是散列技术来存放和查找（名字，记录）对。  哈希函数如下图左侧所示，存放函数如下图右侧所示：    检索函数如下图所示：    至于序列化和反序列化，只需要在序列化的时候注意添加特殊字符和标记来表明数据类型和范围即可。  代码文档组成说明  提交的项目文件夹结构如下：    其中，在codes文件夹中，ball\_A.asm，ball\_B.asm，ball\_C.asm，ball\_D.asm，以及printBigName.asm是5个独立的用户程序，稍微修改自实验2，其余的checkInput.asm，cleanPrint.asm，printnames.asm，showBall.asm是稍微修改的实验2中系统例程，而剩下的loader.asm，os.c分别是引导程序和系统内核程序，utils.asm，utilsAsm.h是汇编语言实现的库函数，utilsC.c和utilsC.h是C语言实现的库函数。最后，manual.txt是用户帮助文件，run.bat是批处理文件，linkscript是链接器ld的脚本配置文件。  在imgs文件夹下，boot.img是最后的系统镜像。  **[实验过程]**  主要流程  创建.img虚拟软盘，编写汇编以及C语言的代码，其中loader和用户程序的代码使用nasm直接汇编出二进制文件，其余的和系统内核相关的代码使用[实验原理]部分所讲的混合编译技术得到二进制文件，再使用bochs调试，以及最后使用vmware展示结果。  输入输出说明  本次实验编写出来的操作系统是命令行形式的，其输入来自于用户的键盘输入，输出则为操作系统对于对应的用户输入的响应，包括了信息展示、过程状态的切换，以及程序调度。  结果展示  以下分别展示该操作系统的各个功能，首先是欢迎界面：  还有man指令显示的用户帮助文件：    然后是ls指令显示磁盘中的程序信息：    显示系统日期的date指令：    清空屏幕的clear指令：    滚屏功能展示：    还有的关机和重启功能就不予展示了。  程序调用，分别是程序ball\_A和printBigname：    遇到的问题以及解决情况   1. 在本次实验中遇到的最大的问题自然是怎么样用NASM和GCC混合编译C语言和汇编语言。这个问题最终从一个GitHub项目里面找到了解决方案。其实回想起来，主要的问题就是要解决GCC和NASM汇编出来的文件的变量位数和地址位数的不匹配问题，以及注意C语言程序往栈中压入的变量位数和汇编语言往栈中压入的变量位数不相同。而解决的方法最后都能从GCC和NASM以及最后的连接器的用户手册找到，但是问题在于用户手册涵盖的信息过于庞大，要自己一点一点地找到所需要的全部参数很困难。这一次的实验让我习惯于产看工具的用户手册，也知道了寻找解决问题的基本方案能够节省不少时间。 2. 在学习X86函数调用约定时，起初不能理解enter和leave的功能，在理解了EBP和ESP在调用函数时起到的作用之后，这些疑惑便引刃而解了。所以理解原理时解决问题的前提条件。 3. 在X86函数调用约定中，一般的返回值放在eax寄存器中，但是要注意的时如果返回的时汇编语言中的变量地址，要注意把高16位置0，否者会出现访问越界的情况。 4. 在加载磁盘中的扇区时，由于没有注意逻辑（相对）扇区和磁面、磁道、扇区的对应关系，花费了不少时间和精力来调试，最后发现根本没有数据被加载，从中断的返回标志中找到了问题所在。 5. 理解错了滚动屏幕的中断的含义，然后自己实现了一遍滚屏功能。 6. 在函数运行时，esp的值初始化为0xffff，而我的内核放在了附近的位置，导致存储位置冲突，程序崩溃。   **[实验总结]**  屏幕剪辑  **[参考文献]**  [1]汇编语言（第三版）, 王爽.  [2] <https://blog.csdn.net/weixin_37656939/article/details/79684611>, BIOS中断大全.  [3] <https://zhuanlan.zhihu.com/p/28659560>, 程序段的划分  [4] <https://github.com/Urinx/SomeCodes/tree/master/Asm/Boot>, 内存区域的划分  [5] <https://github.com/richardtsai/homework>, 参考的项目  [6] <https://zh.wikipedia.org/wiki/X86%E8%B0%83%E7%94%A8%E7%BA%A6%E5%AE%9A#cdecl>, X86调用约定  [7] <https://blog.csdn.net/wbcuc/article/details/7873314>, 调用中断实现关闭计算机  [8] <https://blog.csdn.net/beyondxj/article/details/77104988>，模拟软盘的数据结构，加载指定软盘扇区中的数据 |