OceanOS项目文档

项目说明

Git仓库：[git@github.com:wldxy/OceanOS.git](mailto:git@github.com:wldxy/OceanOS.git)

运行使用方法：

git clone [git@github.com:wldxy/OceanOS.git](mailto:git@github.com:wldxy/OceanOS.git)

cd OceanOS

make

make bochs使用bochs虚拟机

或make qemu使用qemu虚拟机

BootLoader

OceanOs中bootloader选择使用Grub作为操作系统的启动的引导程序，

由于作者对汇编不是很熟练，开发初期使用Grub，可以避免在加载内核中诸如初步进入保护模式等相对难以理解的过程，而把重心放在对操作系统其他板块的实现。

内核选择安放在1.44MB的软盘上，并且将已经安装好GRUB，安装好的Grub的软盘镜像使用常见的FAT12文件系统，方便Grub读取内核。

在OceanOS中，Grub从boot.s文件中获取Grub启动系统需要的Multiboot标准的具体细节，同时Grub给我提供了Multiboot标准提供的系统加载启动过程中的一系列参数。

内核雏形

端口读写

端口读写是操作系统内核最基本的操作（io.h），OceanOS定义了一系列io操作。

|  |  |
| --- | --- |
| 函数名 | 作用 |
| inb(uint16\_t port) | 在端口port处读入一个字节长度的数据 |
| inb\_p(uint16\_t port) | 有io短暂延时的inb函数 |
| outb(uint16\_t port, uint8\_t data) | 在端口port处读出一个字节长度的数据 |
| outb\_p(uint16\_t port, char data) | 有io短暂延时的outb函数 |
| inw(uint16\_t port) | 在端口port处读入一个字长度的数据 |
| outw(uint16\_t port, uint16\_t data) | 在端口port处读出一个字长度的数据 |
| insb(uint32\_t port, void \*addr, int32\_t cnt) | 在端口port处读取长度为cnt的物理地址为addr长度的数据 |
| outsb(uint32\_t port, void \*addr, int32\_t cnt) | 在端口port处输出长度为cnt字节的物理地址为addr长度的数据 |
| insl(uint32\_t port, void \*addr, int32\_t cnt) | 在端口port处读取长度为cnt双字的物理地址为addr长度的数据 |
| outsl(uint32\_t port, void \*addr, int32\_t cnt) | 在端口port处读出长度为cnt双字的物理地址为addr长度的数据 |

以上io函数借鉴子linux的io基本操作函数。

文本模式的建立

文本模式是通过简单的字符的方式来进行打印和输出（monitor.c），具体说明为这是一种一种定义为 x 列字符乘以 y 行字符的视频模式，构建相对方便。基本文件模式的构建已经有显卡提供，剩下的只需要读写现存区域就可以实现基本的文本模式的输入输出。根据定义0xB8000～0xBFFFF地址段便是映射到文本模式的显存的。通过修改上述地址即可实现文本模式的输出，输入板块需要后面结合键盘驱动中实现。

为了方便输出文本模式下的输出，借鉴c中最常用的stdio.h中的printf函数的设计实现自己的在内核中输入输出的printf函数。

字符串操作

由于在操作系统设定中，有大量修改内存的操作与字符串的操作相当一致，所有OceanOS仿照c中的string.h的函数设计，设计了OceanOS中通用的一系列字符串处理操作（string.h）

引入GDT

GDT为全局段描述符表，GDT存放在内存中，但是CPU需要知道GDT所放置的

位置，寄存器GTDR就是设计来存放GDT的入口地址，程序中通过构建GDT后，使用gdt\_to\_gdtr()函数将gdt的入口地址放入gdtr中。

struct gdt\_struct {

uint16\_t limit\_low;

uint16\_t base\_low;

uint8\_t base\_middle;

uint8\_t access;

uint8\_t granularity;

uint8\_t base\_high;

} \_\_attribute\_\_((packed));

typedef struct gdt\_struct gdt\_t;

struct gdtr\_struct {

uint16\_t limit;

uint32\_t base;

} \_\_attribute\_\_((packed));

typedef struct gdtr\_struct gdtr\_t;

在程序中（gdt.h）中gdt\_struct用来记录gdt的数据，\_\_attribute\_\_((packed))是说明在编译过程中，对改结构不做内存对齐的处理，原因是gdt\_sturct在内核中应将所有的参数构建成一个48位的数，而内存对齐可能导致数之间有空隔开或者数据存放顺序与结构中顺序不符，\_\_attribute\_\_((packed))在后续开放中有着重要的作用。

中断实现

中断实现的第一部分是设置IDT表。IDT作为和GDT一同作为系统重要的系统表，名为中断描述符表。IDT的处理与GDT类似，IDT存放位置在内存，IDTR寄存器以存放IDT的入口地址。设置IDT后，如果要真正实现中断，还需要中断处理函数，中断处理函数主要包括三个部分：

1. 中断发生后现场保护；2、中断发生后的针对不同中断的中断处理函数；3、中断处理结束后的现场恢复。

其中第一三部分由汇编代码实现，而第二部分由一个256个的全局中断函数数组组成

interrupt\_handler\_t handlers[256];

在发现中断后，调用相对应的中断码的中断处理函数即可，而注册中断处理函数只需要在handlers中断对应的中断码处赋予中断处理函数的地址即可。

接着是中断请求的实现，中断请求为硬件发生中断后的请求动作，硬件由8259A芯片来控制，而8259A共有15组IRQ供给硬件使用。下表为IRQ的对应的硬件使用。

|  |  |  |
| --- | --- | --- |
| **IRQ编号** | **设备名称** | **用途** |
| IRQ0 | Time | 电脑系统计时器 |
| IRQ1 | KeyBoard | 键盘 |
| IRQ2 | Redirect IRQ9 | 与IRQ9相接，MPU-401 MDI使用该IRQ |
| IRQ3 | COM2 | 串口设备 |
| IRQ4 | COM1 | 串口设备 |
| IRQ5 | LPT2 | 建议声卡使用该IRQ |
| IRQ6 | FDD | 软驱传输控制用 |
| IRQ7 | LPT1 | 打印机传输控制用 |
| IRQ8 | CMOS Alert | 即时时钟 |
| IRQ9 | Redirect IRQ2 | 与IRQ2相接；可设定给其它硬件使用 |
| IRQ10 | Reversed | 建议保留给网卡使用该IRQ |
| IRQ11 | Reversed | 建议保留给AGP显卡使用 |
| IRQ12 | PS/2Mouse | 接PS/2鼠标，若无也可设定给其他硬件使用 |
| IRQ13 | FPU | 协处理器用，例如FPU（浮点运算器） |
| IRQ14 | Primary IDE | IDE0传输控制用 |
| IRQ15 | Secondary Ide | IDE1传输控制用 |

其中IRQ0-IRQ15分别对应了中断码中的32-47，所有中断请求只需要在上述过得全局的中断处理函数数组中指定的中断码中注册中断请求函数，即可实现中断请求的初始化。

OceanOS系统中对于中断的应用集中在两个一个是IRQ1键盘中断与IRQ0计时器中断，键盘中断主要是为了实现键盘输入，这是输入系统的基础。定时器中断主要是为了记录从开机到现在运行时间，同时为后来的多进程的实现打下基础。

输入输出系统

输入输出系统由输入系统和输出系统组成。

输出系统有monitor.c和kernelio.c中的printf函数组成，实现了屏幕25\*80的文本模式下的文本的输出。

输入系统基于上述的键盘中断实现，键盘中断发生后，通过0x60端口读取键盘按动的数据，而键盘输入的处理根据注明的linux0.01实现，具体不同按键按动后的处理函数，在keyboard.c中可以找到。同时对于输入的数据将会放入一个简单实现的输入缓存区中。

在使用方面，为了方便运行自己实现的模拟函数，基于c中著名的stdio.h中的scanf的实现，结合上述实现的简单输入缓存和键盘输入实现scanf。

OceanOS的输入输出系统由上述的scanf和printf，以及monitor的一系列操作实现。

内存管理系统

物理空间分配

物理页面分配使用

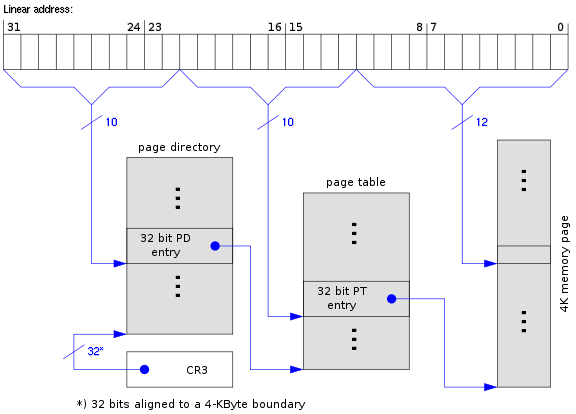
物理空间分配中最重要的就是物理内存页面的分配，首先说明的是在OceanOS页面空间大小为4KB，这是常见的分页管理中使用的页面大小，在OceanOS中使用的页面分配算法为buddy算法，buddy算法是现在常见的linux使用的页面分配算法。OceanOS实现较为简单地实现（memlib.c）。

Buddy算法的设计思想，分配的内存空间均为2的k次幂，若申请n个页的内存空间，实际分配的空间大小为2k个页面（2k-1<n<=2k）。具体实现为，内存一开始为一个连续的内存空间，初始化生成一个长度为看k+1的可用空间列表，将所有空间作为一个大小为2k个页面的内存块挂在空闲空间数组的链表的最后一个节点上。分配空间策略，在申请n个页面是，Buddy算法从最适合的第k项（2k-1<n<=2k）开始从列表中，开是搜索列表，如果第k项为空，就接着检索k+1项，直至检索到第一个符合的内存块2m，将其分配为2个2m-1的内存块，并更新页面位置同时接着分裂，直至分裂到2k的内存块分配给申请者。

当申请的内存块被释放的时候，buddy将内存块回收到列表中，通过内存的大小来决定内存块的位置，同时尝试把同样大小的内存块的进行合并，直至无法合并为止。

虚拟页面管理

虚拟内存管理主要仿照由Linux初期的二级页面管理实现，实现细节主要仿照《JamesM's kernel development tutorials》中的虚拟页面部分实现（memory.h）。



根据Linux的设计中可以得知，linux虚拟页面管理中将页目录放置在cr3中，当遇到一个线性地址需要转换为物理地址时，先是根据前10位获取到相对应的pgd中的pte（页表项）地址，再根据地址的中间10位找到pte表中对应的pte项，根据pte可以换取到该地址相对应的页的物理地址，同时根据最后12位的业内偏移，可以得到最后的物理地址。

任意大小内存分配

任意大小内存分配只要为了实现malloc和free这两个函数，这里使用的是最简单的最先适应算法，首先申请到的内存块，内存块之间使用链表进行连接管理，通过遍历空闲内存块来寻找可以分配的内存块，第一个找到的合适内存块就进行切割分配，如果不存在则向物理内存管理申请更多的内存页，在进行分配（memlib.c）。

多进程管理

这里多线程实现没有使用TSS来进行多进程管理，所以暂时没有实现ring0向ring3 的转换，这里只实现了内核级的多进程管理，所以实现相对简单，通过init\_kernel函数可以实现创建了一个process 的实例，同时创建一个STACK\_SIZE大小的堆，并通过一个简单的压栈机制，把函数，参数，退出函数压至栈中，同时创建一个进程PCB，并且加之运行进程链表之中，进行进程调度，该进程管理主要只有两个进程状态，一为运行，一为等待，等待可以通过设置进程的状态来修改。同时，进程调度中最重要的就是定时器的帮助，我们需要把我们的调度函数放到前面已经完成定时器中，在没经过一个时间短后进行进程调度。

文件系统

OceanOS中没有实现一个真正文件系统，而是仿照Linux的VFS文件系统来设计一个文件系统管理器（file.h），目前可以实现在内存上的模拟一个文件的管理，实现一个跑在内存上的文件系统，实际实现文件系统相对较为复杂，OceanOS暂时还没有实现。