1.

通过注释发现这段代码的功能是输出无符号八进制整数,其实上面就有无符号十进制数的代码,二者格式基本一致...

首先调用getuint函数,它通过一个可变参数列表的current指针获取列表中的下一个uint型变量, 此前判断出的long flag决定这个变量是不是long型的.

然后设置base变量为8 , 调用number程序段.number是专门用来解析输出一个数字的, 它提供了 位,宽度,占位符等等要求.在此程序中我们只要把base设8就可以,其余的都会在格式符’o’之前被解析.

2.

这个函数的功能是回溯输出eip,ebp,函数参数的信息.  
 ebp是当前函数所对应的栈基址,esp是栈顶(底,向下生长).于是任何函数所拥有的栈空间为[ebp, esp].调用函数所需要维护的就是ebp(为了划分区域)和eip(为了正确返回).

具体过程是,当在遇到一个函数调用的时候, 会先将调用的参数按从右向左顺序压入堆栈 , 然后将当前的eip, ebp进栈,再将当前的esp赋给ebp,这样就以ebp为界划分出了新函数的栈空间.

代码的过程就是与上述过程相反,当刚进入mon\_backtrace函数时,通过read\_ebp与read\_eip函数得到当前的ebp eip.这时因为刚刚进入函数, esp所指的内容就是上一个函数的ebp,esp上面一块指的就是上一个函数调用此函数时的eip,紧接着上面是调用的参数,在此程序中要求固定输出5个,虽然他们不一定都是参数.最后将ebp赋给esp,这样就又到了刚刚进入这个函数时的情况了,因此整个过程重复,用while循环. 在entry.S中看到ebp初始为0,就是为了让这个backtrace能有适当的终止条件.

呃,以上的ebp eip等都是用变量从栈中访问的,不是对应的寄存器.

//这个grade脚本好坑人,一开始地址没用0占位判错,debug好半天

2.5:

完善debuginfo\_eip函数.并调用它完善backtrace

这个函数的功能是通过传入一个地址,来获取这个地址指向的指令在哪个文件,哪个函数,哪行,函数内偏移量等等调试信息,并把他们填充到指定的Eipdebuginfo结构体中.

Jos提供了一个stab的调试信息文件,以及一个方法stab\_binsearch用于在stab文件中二分查找指定的地址对应的信息.已完成的过程是确定目标地址是在哪个文件,哪个函数,并且确定了最后的查找范围[lline,rline] , 同时把获得的函数名字符串,函数名长度等等信息保存在Eipdebuginfo中了,接下来只需要调用stab\_binsearch函数确定具体的行数就可以了.找到的stab的n\_desc域就是我们要获取的行号.

最后要在backtrace函数中调用它,按照所给的格式,输出

文件名:行号: 函数名+函数内偏移量

要注意的是函数名是我们得到的Eiodebuginfo中的eip\_fn\_name域不仅仅是函数名,后面还多了貌似参数的什么神秘信息,不过我们记录了函数名的长度eip\_fn\_namelen,取前这些位就行了.函数内偏移量我们没有直接记录,我的想法是通过eip变量减去Eipdebuginfo中获取的函数起始地址.个人感觉是可行的,但是make grade没过.

最后运行,发现前7次都是在backtrace函数中,最后一次在i386\_init中,这与源码调用层次一致,应该没什么问题吧???

3:

从此作业开始,我在源码内写了较多中文的注释,虽说之前注释也不是很少.

1) boot\_alloc函数是在boot时临时使用的,它分配了大小为”n对PGSIZE向上对齐”的一块区域.思路就是维护一个nextfree指针,起始时用end[]关键字赋值,可以认为这个end[]就是链接器标示出的内核程序结尾的kva.从end[]向上到nextfree,都是我们认为以分配出去的kva空间.它通过在entry.S中加载的entry\_pgdir被映射到了以0起始的4mb物理内存空间,所以应当控制它分配出去的内存大小.因此如果分配后的指针地址减去KERNBASE,即我们已实际分配的物理内存长度,大于npages \* PGSIZE,即我们实际拥有的物理内存大小, 就说明内存不足.

2) page\_init函数初始化维护物理内存页框的pages数组,并初始化维护空闲物理页框的链表page\_free\_list.注释中指出有三部分的物理内存是初始就被占用的,

1是base memory的第一页,给IDT和BIOS;

2是[IOPHYSMEM,EXTPHYMEM)这里,是IO hole

3是内核和我们之前分配出去的页目录pgdir.

用for循环,略过这些,对齐分配就可以了.

3)page\_alloc函数在空闲页链表中找到一个物理页去分配,并维护这个链表.返回这个页的指针.注意当参数alloc\_flags为1时,要清空这个分配的物理页框.调用string.h里的memset函数,要用到page2kva宏来获取这个物理页框对应的kva.虽然我们现在用的还是4mb的线性映射,但是应该能保证在加载完备的页目录之前,这个函数分配的内存不要超过4mb

4)page\_free函数将一个物理页添加到空闲页链表中.也是简单的链表维护操作.

mem\_init函数放在之后介绍

4:

1)pgdir\_walk函数试图在给定的页目录pgdir中找到线性地址(此情况下等于kva)va对应的页表项,如果找到了,返回它的指针;如果没找到,判断如果参数create==1,就给他分配一个新的页表页,增加其引用计数,初始化清空,依然返回对应页表项的指针;如果create==0,就返回NULL.要考虑内存不足的情况.

首先取va的前10项获取在pgdir中的下标,通过标志位PTE\_P判断对应的页表页是不是存在于物理内存中,如果是,就开心地继续;如果不,再考虑分配事宜.总之如果参数判断后能获得对应的页表起始的kva,就再将其加上va的第二个高10位部分,即在页表中的下标,得到其对应页表项的kva,返回.

需要注意的是,页目录的每一项存的是对应页表的物理地址,这是因为mmu的翻译只

”虚拟了”页目录地址,拿到虚拟地址通过cr3访问页目录之后所有的地址访问都是物理地址.

所以如果申请了新的一物理页,我们需要在页目录中保存的是他的物理地址,而在此函数之外(mmu之外)我们需要的是他的虚拟地址;如果页目录中有相应的页表页,要通过他保存的物理地址,获得他的物理页结构体指针,从而获得他的kva.这些可以用给出的宏实现.

2)boot\_map\_region函数在页目录pgdir中,将从kva va开始,大小为size的空间以权限perm映射到物理地址pa上.所谓boot\_,其实也是初始时临时用的,所以不改变page的引用计数.

方法就是以页为单位,对这个空间的每一页建立映射.通过pgdir\_walk获得当前页对应的页表项的指针,然后将这个页表项的信息的地址位赋值为对应的物理地址,权限位按照参数perm对应调整.

注意这个函数是以页为单位的,也就是只能映射对PGSIZE向上对齐的空间大小.虽然逻辑上不完善,但是其实就是想这么用的.

3)page\_lookup函数返回映射到虚拟地址va的物理页结构体指针.如果参数pte\_store不为0,就将对应的pte地址保存在它里面;如果没有相关物理页映射,就返回NULL.

就调用一下pgdir\_walk查找对应的pte,判断存不存在映射,判断是否需要储存.最后用这个pte获取物理页结构体的物理地址然后转换为结构体指针即可.依然靠给出的宏.

4)page\_remove()函数解除虚拟地址va的映射,如果va本来就没有映射,那么什么也不做.

用page\_lookup函数查找va对应的物理页结构体指针,同时给他传入一个pte\_store参数以保存对应的页表项.之后先禁用tlb中这个页表项的缓存,然后调用给出的page\_decref函数减少这个物理页的引用计数(它还同时判断这个物理页会不会变为空闲).

最后将对应的pte清空为0.

5)page\_insert函数在页目录pgdir中建立虚拟地址va到物理页框pp的映射,权限按照参数perm设置.如果va已经有了一个映射,那么移除他,再建立新的映射.有必要的话,新申请一个物理页来存对应的页表.映射建立成功返回0;失败返回常量-E\_NO\_MEM.

首先通过pgdir\_walk函数得到va对应的pte, 参数create为1以告知它有必要申请,如果它仍然返回NULL,说明内存不足,当前函数返回-E\_NO\_MEM.然后检查得到的pte是否已经存在映射,如果存在就调用page\_remove解除它.最后更新pte.

一个特殊情况是这个函数重复建立va到pp的映射,而此时pp的引用计数为1.那么我们也许会错误地将pp加入到空闲链表中.可以在解除旧的映射之前先将pp的引用计数加1,类似预定,那么在此过程pp就无论如何不会变为空闲的了,保证逻辑正确.

5.

最后mem\_init函数调用其他函数来初始化内存.这个函数说明了我们boot时要做的内存工作.

它先调用硬件检测实际物理内存.然后用boot\_alloc申请一个页目录kern\_pgdir并清空.然后建立了kva UVPT 到 kern\_pgdir物理地址的映射,这样之后就可以在kern\_pgdir通过UVPT这个规定的虚拟地址访问它自己了.这个映射的建立也是依靠与entry\_pgdir的4mb线性映射,才能用索引+下标的形式完成的.

调用boot\_alloc函数申请物理页结构体数组pages维护物理页,数组的大小由之前检测出的npages决定.从此我们就可以以页为方法维护物理内存了.接下来调用page\_init()初始化pages结构体.然后检查空闲链表,物理页的分配等等.

调用boot\_map\_region进行初始的内存区域映射.映射kva UPAGES到pages的4mb物理地址.这是为了用指定的UPAGES寻址pages结构体.从KSTACKTOP向下映射KSTSIZE大小到bootstack的物理地址上,这是设置内核的栈区.从KERNBASE向上映射整个32位虚拟地址空间到物理地址0上.这是建立内核虚拟地址空间.

调用check\_kern\_pgdir函数check一下页目录写的对不对啊. 然后把那个呕心沥血的kern\_pgdir加载到cr3里.再检查一下是不是能工作啊.gg wp.

问题:

1-1:

事实上从cr0的PE位被置1的时候,cpu就进入32位保护模式了.紧接着ljmp语句, 使 用了段选择子这种寻址方式, 可以认为这时就开始以保护模式工作了.

1-2:

Boot loader执行的最后一条指令是跳转到elf头的e\_entry标记的内核入口去;

内核的入口entry.S执行的第一条指令是

movw $0x1234,0x472

貌似是个热身运动......

1-3:

内核入口在ELFHDR的固定位置上面描述,就是那个32位e\_entry,因为这时还不涉及虚拟地址(segment是个花瓶),所以进入\_start也就是entry的物理 地址.

1-4:

还是在elf头里, 那个叫e\_phnum的32位int.

2-1:

console.c从底层实现了输出一个字符到显卡,串行口,并行口;

printfmt.c是解析参数列表和字符格式,从而知道输出什么字符

printf.c是调用这些的接口,cprintf函数接受参数后, 通过printfmt.c解析格式, 通过 console.c进行底层输出.

2-2:

这段语句是当crt的光标超过了crt的尺寸时, 将整个缓存上移一行,并且将最下面一行 赋值成空白

功能就是窗口内容满了的时候,向下刷一行

3-1:

X是unintptr\_t , 因为x最后被赋值成value , value经历过一次解引用, 在代码正确 的情况下是一个虚拟地址 , 故x也是虚拟地址.

4-1:

在mem\_init()里, 有四次映射:

在UVPT处映射了页目录自己 , 大小为一页表;

在UPAGES处映射了管理物理页的结构体pages数组,大小为PTSIZE;

在KSTACKSTOP - KSTKSIZE处映射了一段内存分配给栈, 大小为KSTSIZE;

在KERNBASE处映射了其上方所有虚拟内存, 大小为2^32 - KERNBASE;

因此,页目录中

第956项 , 指向pages ;

第957项 , 指向页目录自己;

第959项 , 指向kernel栈;

第960项之后, 指向所有物理内存.

4-2:

因为PTE和PDE上有相应的权限位 , 在试图访问的时候核实权限位就能阻止用户程 序的访问.

4-3:

虚拟地址空间里UPAGES的大小限为4mb = 2^22, 所以最多能有2^19个page 结构体维护物理内存页框 , 每个页框维护4kb = 2^12物理内存, 所以最多支持 2^31 = 2gb物理内存.实际上好像还没这么多,因为申请pages的时候用的 entry\_pgdir映射,只映射了4mb的物理内存,其中页目录什么的还占用了一部分.所以 实际给到pages结构体的内存小于4mb

4-4:

如果有2gb内存, 为了管理它们 , 需要一堆page结构体(4mb), 页目录(4kb),一堆 页表(4mb) (虚拟地址空间仍是32位).可以用多加几级页表来减小内存开销

/\*

这个jos最坑的地方就是KERNBASE居然不是内核加载的地方...KERNBASE+0x1000才是....我一直以为用了什么黑科技二次映射神秘硬件机制..

\*/