

XShell/Xftp 无法连接 Ubuntu20

2022年4月2日 17:35

环境

Ubuntu: 20

VMware Workstation: 15.5

问题

在虚拟机VMware下，Xshell/Xftp 模式连不上虚拟机

解决方案

ubuntu初始安装，默认 ssh只安装客户端（即只能 ssh连接其他主机），没有安装服务端(即不能被其他主机ssh连接本机)，所以在此情况下，Xshell连接虚拟机失败。

需要操作：

```
# 安装 ssh 服务端
sudo apt-get install openssh-server
# 显示 sshd 即可以成功连接
ps -e |grep ssh
# 如果不显示 sshd
sudo /etc/init.d/ssh start
```

错误收集

2022年4月12日 14:07

E: Could not open lock file /var/lib/dpkg/lock - open (13:
Permission denied

https://blog.csdn.net/mao_hui_fei/article/details/107081870

哎呀好烦啊，这个错误真的找了好久

如果你的bochs并没有进入调试模式,或者出现'Segmentation fault'。请检查你是否是在bochs目录下运行的bochs。

Linux指令收集

2022年4月13日 10:08

Linux：

指定配置文件:

bochs -f bochsrc.disk

更改文件夹和子目录权限

chmod -R 777 文件名称	获取文件所有权限
ls -lb 文件名称	查看文件属性
chmod 777 -R *	修改当前目录下所有文件的属性

dd:

写入指令，但是我写入失败了，最后还是通过010edit复制的， dd if=mbr.bin of=/bin/hd60M.img bs=512 count=1 conv=notrunc

启动任务管理器:

gnome-system-monitor

卸载软件:

sudo apt-get remove --purge 要卸载的软件的名字 #卸载软件同时删除配置文件

sudo apt-get remove 要卸载的软件的名字 #卸载该软件

查看系统函数:

man 2 functionName

NASM汇编指令：

vstart:

指定当前段的起始地址

\$和\$\$:

相当于取当前地址，如果存在vstart,\$\$相当于section的起始地址，\$相当于基于段的偏移。如果使用了vstart关键字，想获取当前section的偏移,**section.段名称.start**

section:

定义段,格式:section 段名

编译:

nasm -o 目标名称.格式 源文件

bug集合

2022年7月19日 17:41

```
114 //写入lba高4位
115 uint8_t reg_device = BIT_DEV_MBS|BIT_DEV_MOD_LBA|(hd->dev_no?BIT_DEV_DEV:0)|(lba>>24); //三目运算符写反，排错4个小时
116 outb(reg_device(channel),reg_device);
117
```

忘记写++:

```
//检查是否存在文件系统
if(sb_buf->magic==0x10190498){
    dbg_printf("%s has filesystem\n",part->name);
} else{
    dbg_printf("formatting %s's partition %s .....\\n",hd->name,part->name);
    partition_format(part);
}
part++; //跳转到下一个分区 bug 没有写++
part_idx++;
```

内存池扇区数忘记*扇区字节数:

```
//初始化内核和用户物理内存池结构体
kernel_pool.phy_addr_start = kp_start;
kernel_pool.pool_size = kernel_free_pages*PG_SIZE; //忘记 "*PG_SIZE", 7/25回来修改bug
kernel_pool.pool_bitmap.bitmap_byte_len=kbm_length;
kernel_pool.pool_bitmap.Pbitmap = (uint8_t*)MEM_BITMAP_BASE;
```

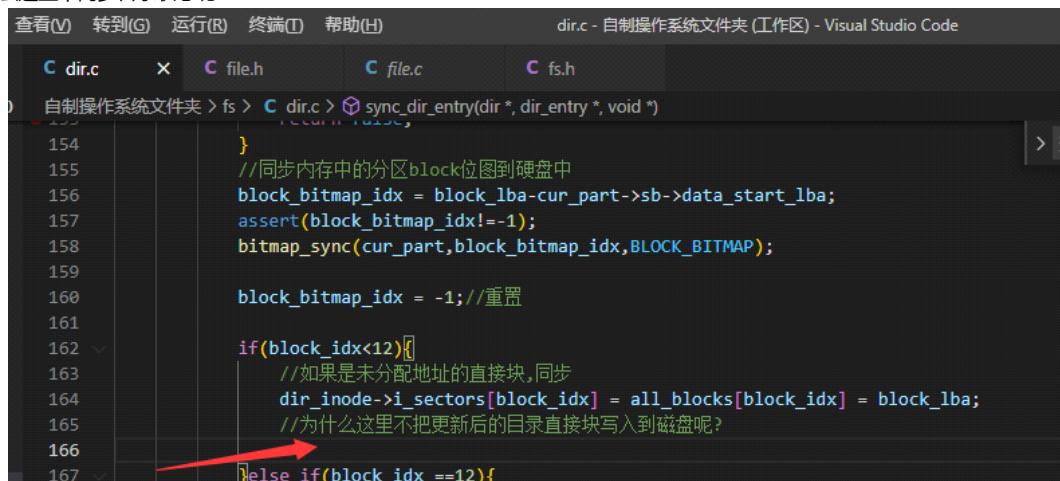
ide_write:

```
//将数据写入到硬盘
write2sector(hd,(void*)((uint32_t)buf+secs_done*512),secs_op); //bug, secs_done写成了secs_op
```

strcmp:

```
68 /*字符串比较,如果str1>str2则返回1,如果str1=str2,返回0,str1<str2,返回-1.不会进行长度检查可能会产生越界*/
69 int8_t strcmp(char* str1,char* str2){
70     assert(str1 != NULL && str2!=NULL);
71     while (*str1 != 0)
72     {
73         if(*str1>*str2) return 1;
74         else if (*str1<*str2) return -1;
75         str1++; //忘记++,导致死循环
76     }
77     return 0;
78 }
```

为什么这里不同步目录状态呢?



```
查看(V) 转到(G) 运行(R) 终端(T) 帮助(H) dir.c - 自制操作系统文件夹 (工作区) - Visual Studio Code
C dir.c X C file.h C file.c C fs.h
自制操作系统文件夹 > fs > C dir.c > sync_dir_entry(dir *, dir_entry *, void *)
154 }
155 //同步内存中的分区block位图到硬盘中
156 block_bitmap_idx = block_lba-cur_part->sb->data_start_lba;
157 assert(block_bitmap_idx!= -1);
158 bitmap_sync(cur_part,block_bitmap_idx,BLOCK_BITMAP);

159 block_bitmap_idx = -1;//重置

160 if(block_idx<12){
161     //如果是未分配地址的直接块,同步
162     dir_inode->i_sectors[block_idx] = all_blocks[block_idx] = block_lba;
163     //为什么这里不把更新后的目录直接块写入到磁盘呢?
164
165
166 }else if(block_idx ==12){

200 //改变当前目录的大小
201 dir_inode->i_bsize +=dir_entry_size;
202 //这里没有同步目录状态,在以后同步?
203 return true;
204 }
```

因为没有初始化磁盘结构体的成员变量的值, 导致后面分区名称和分区是否存在的判断错误:

```

425     disk* hd = &channel->devices[dev_no];
426     memset(hd, 0, sizeof(disk)); //保证所有成员初始化0，方便后面是否存在分区判断以及分区名称copy
427
428     //清空分区name缓冲区
429     memset(hd->prim_parts[p_no].name, 0, 8);
430     sprintf(hd->prim_parts[p_no].name, "%s%d", hd->name, p_no+1); //主分区编号从1到4
431     p_no++;
432     assert(p_no<4);
433
434     /*忘记++：导致内核进程无限循环
435      *初始化其他文件描述符为 -1 */
436     uint8_t id_x = 3;
437     while (id_x<MAX_FILES_OPEN_PER_PROC)
438     {
439         thread_pcb->fd_table[id_x] = -1;
440         id_x++;
441     }
442     thread_pcb->pid = allocate_pid();

```

全局变量初始化：

```

397     /**
398      * @brief
399      * 硬盘数据结构初始化
400      */
401     void ide_init(void){
402         ext_lba_base =0;
403         p_no = 0, l_no = 0;
404
405         uint8_t hd_cnt = *(uint8_t*)(0x475); //硬盘数量
406         if(hd_cnt > 0)

```

```

329     static void partition_scan(disk* hd,uint32_t ext_lba){
330
331         boot_sector* bs = sys_malloc(sizeof(boot_sector));
332         ide_read(hd,ext_lba,bs,1);
333         uint8_t part_idx = 0;
334         partition_table_entry* p = bs->partition_table;
335
336         while (part_idx++<4)//一个硬盘，最多4个分区表
337         {
338
339             if(part_idx==1){
340                 p+=4; //指向mbr末尾
341                 if(*(uint16_t*)(p)!=0xaa55){
342                     dbg_printf("magic error\n");
343                 }else{
344                     dbg_printf("magic success\n");
345                 }
346                 p-=4;
347             }
348             if(p->fs_type ==0x5)
349             {////扩展分区，在给硬盘分区时，自定义的类型
350                 if(ext_lba_base!=0){
351                     partition_scan(hd,p->start_lba+ext_lba_base);
352                 }else{//表示是第一次读取引导块
353                     ext_lba_base = p->start_lba;
354                     partition_scan(hd,p->start_lba);
355                 }
356             }else if(p->fs_type!=0)

```

虽然我在定义全局变量的时候，已经赋值为0，但是在实际运行中被设置为了奇怪的值。导致我在partition_scan中使用 ext_lba_base = ext_lba,这就导致我后面的逻辑分区基于第一个扩展分区的起始lba地址错误。

strrchr:

strrchr函数本身没有从后往前而是从前往后，虽然这次没有导致错误 因为只有一个根路径，但还是排查到了没有对返回的字符串地址+1，导致存入的文件名多了一个 '/'

```

    dbg_printf("start to create file %s\n", strrchr(search_record.searched_path, '/')+1);
    fd = file_create(search_record.parent_dir,strrchr(search_record.searched_path, '/')+1,flags);
    dir_close(search_record.parent_dir);
    break;

```

同步块位图：

应该用空闲块起始地址+idx

原先的错误用法是使用了 block_bitmap_start_lba

```

/**
 * @brief
 * 分配一个扇区
 * 成功返回扇区lba地址
 * 失败返回-1
 */
int32_t block_bitmap_alloc(partition* part){

    int32_t bit_idx = bitMap_scan(&part->inode_bitmap,1);
    if(bit_idx== -1){
        return -1;
    }
    bitmap_set(&part->inode_bitmap,bit_idx,1);

    return (part->sb->data_start_lba+bit_idx); |<-- red arrow
}
*/
/* Objekt-f

```

翻看代码发现的错误： block alloc 居然使用了inode 的位图。

```

/**
 * @brief
 * 分配一个扇区
 * 成功返回扇区lba地址
 * 失败返回-1
 */
int32_t block_bitmap_alloc(partition* part){

    int32_t bit_idx = bitMap_scan(&part->block_bitmap,1);
    if(bit_idx== -1){
        return -1;
    }
    bitmap_set(&part->block_bitmap,bit_idx,1);

    return (part->sb->data_start_lba+bit_idx);
}

```

dir.c delete_dir_entry 书中的循环遍历间接表并没有使用++：

```

312
313         while(indirect_block_idx < 140)
314     {
315         if(all_blocks[indirect_block_idx] != 0)
316         {
317             indirect_blocks++;
318         }
319     }

```

现象：

块申请的时候总是能申请到 第一块和第二块。

实际上，第一块是根目录的直接块

第二块是每个分区超级快

错误点：

1.在格式化分区时，没有同步内存中的块位图到磁盘中

```

//同步block位图
memset(buf,0,buf_readl_size);
*buf =3;
ide_write(hd, sb.block_bitmap_lba,buf,1);
dbg_printf("block bitmap write value:%d\n",buf);

```

2.在挂载分区时，读取磁盘中inode位图使用了错误的属性导致block位图被覆盖

```

23  /**
24  * @brief
25  * 在分区链表中找到名为part_name的分区，并将其指针赋值给cur_part
26  */
27  static bool mount_partition(list_elem* pelem,void* arg){
28      dbg_printf("mount_partition run\n");
29      char* part_name = (char*)arg;
30      partition* part = (partition*)struct_entry(pelem,partition,part_tag);
31      dbg_printf("part_name:%s,part->name:%s\n",part_name,part->name);
32      if(strcmp(part->name,part_name)==0){
33          cur_part = part;
34          disk* hd = cur_part->my_disk;
35
36          /*读取分区超级块*/
37          super_block* sb_buf = sys_malloc(sizeof(super_block));
38          if(cur_part->sb ==NULL){
39              cur_part->sb = (super_block* )sys_malloc(sizeof(super_block));
40          }
41
42          if(sb_buf==NULL||cur_part->sb ==NULL){
43              dbg_printf("sys_malloc error\n");
44              return;
45          }
46          //将硬盘中的超级块赋值给内存中的超级块
47          memset(sb_buf,0,SECTOR_SIZE); //super_block size = super_block
48          ide_read(hd,cur_part->start_lba+1,sb_buf,1);
49          memcpy(cur_part->sb,sb_buf,SECTOR_SIZE);
50
51          //复制硬盘中的block位图到buf
52          cur_part->block_bitmap.Pbitmap = (uint8_t*)sys_malloc(sb_buf->block_bitmap_sects*SECTOR_SIZE);
53          cur_part->block_bitmap.bitmap_byte_len = sb_buf->block_bitmap_sects*SECTOR_SIZE;
54          ide_read(hd,sb_buf->block_bitmap_lba,cur_part->block_bitmap.Pbitmap,sb_buf->block_bitmap_sects);
55
56          dbg_printf("blkok bitmap value:%x\n",*cur_part->block_bitmap.Pbitmap);
57
58
59          /*读取硬盘中的inode位图到buf*/
60          cur_part->inode_bitmap.Pbitmap = (uint8_t*)sys_malloc(sb_buf->inode_bitmap_sects*SECTOR_SIZE);
61          cur_part->inode_bitmap.bitmap_byte_len = sb_buf->inode_bitmap_sects *SECTOR_SIZE; ←
62          ide_read(hd,sb_buf->inode_bitmap_lba,cur_part->inode_bitmap.Pbitmap,sb_buf->inode_bitmap_sects);
63          list_init(&cur_part->open_inodes);
64          sys_free(sb_buf);
65
66          dbg_printf("mount %s done!\n",part->name);
67          return true;
68      }
69      return false;
70  }

```

位图同步错误：

现象：block的位图信息，在磁盘建立文件系统后正常，在创建文件或目录后被置零

问题点：此处同步的时候使用了错误的位图指针

```

/* btmp:位图类型
*/
void bitmap_sync(partition* part,uint32_t bit_idx,uint8_t btmp){

    uint32_t off_sec = bit_idx/4096; //保存位图，相对于位图起始扇区的扇区偏移

    uint32_t off_size = bit_idx/8; //被复制的位图字节偏移

    uint32_t sec_lba;
    uint8_t* bitmap_off;

    switch(btmp){
        case INODE_BITMAP:
        {
            sec_lba = part->sb->inode_bitmap_lba+off_sec;
            bitmap_off = part->inode_bitmap.Pbitmap+off_size;
            break;
        }
        case BLOCK_BITMAP:
        {
            sec_lba = part->sb->block_bitmap_lba+off_sec;
            bitmap_off = part->block_bitmap.Pbitmap+off_size;
            break;
        }
    }
    ide_write(part->my_disk,sec_lba,bitmap_off,1);
}

```

哎呀为什么块同步的bug这么多：

问题：同步之前内存中为3，磁盘中为3，同步之后磁盘中被归零了。检查block bitmap 同步函数

```
kernel main thread pid:0x1
*****sys_mkdir*****
NO sync memory block bitmap value:3
block_idx:0x2
No sync disk block bitmap value:3
YES sync disk block bitmap value:0
sync_dir_entry lba:0x2AA
*****sys_end*****
*****sys_mkdir*****
NO sync memory block bitmap value:?
block_idx:0x3
No sync disk block bitmap value:0
YES sync disk block bitmap value:0
sync_dir_entry lba:0x2AC
*****sys_end*****
```

其实不是同步函数问题

在

sys_mkdir 中使用了错误的方式计算了 block_idx

```
}
new_inode.i_sectors[0] = block_lba;
block_bitmap_idx = block_lba - cur_part->sb->block_bitmap_lba;
assert(block_bitmap_idx!=0);
```

```
| [ ] [ ] [ ] [ ] |
sdb7 has filesystem
sdb8 has filesystem
sdb9 has filesystem
mount_partition run
part_name:sdb1,part->name:sdb1
block bitmap value:F
mount sdb1 done!
block bitmap value:F
block bitmap value:F
```

此时位图同步正常

遍历sys_mkdir创建的目录异常：

现象：只有一个目录项，或没有目录项。因为遍历函数是根据 目录项的inode的i_bsize进行遍历

问题：sys_mkdir中创建目录时，没有修改目录项大小

```
//这里忘记修改inode的size，导致遍历函数无法完全遍历目录
new_inode.i_bsize = 2*cur_part->sb->dir_entry_size;
```

现象：

新创建的空白进程，虚拟内存池出现部分被占用情况。

通过排查创建新进程的步骤，发现申请占多个页虚拟地址的虚拟地址时，位图没有被正确设置。

问题：

怀疑是栈溢出导致，结构体成员互相覆盖。

怀疑是初始化未归零

实际上是在vaddr_get函数中使用bitmap_setEx设置虚拟地址位图，而bitmap_setEx中没有++

```
void bitmap_setEx(PBitMap pBitMap, uint32_t bit_offset, uint8_t value, uint32_t bit_cnt){
    while (bit_cnt--)
    {
        bitmap_set(pBitMap, bit_offset, value);
        bit_offset++;
    }
}
```

现象：

在r3中调用路径解析函数，报空地址异常

检查路径解析函数，发现其在通过中断进入r0后，sys_malloc无法通过用户内存块描述数组 (u_block_desc) 分配内存，内存块描述符内容填充错误

检查process_execute函数发现，其存在用户内存块描述数组函数 (block_desk_init)，检查函数后未发现问题

检查在创建进程的初期时的 u_block_desc 是否与进入r0后地址相同，发现存在问题，地址不同怀疑中断流程中esp切换错误

检查任务调度函数，检查tss更新函数，检查中断流程，发现其代码不在其进程中执行。

最后检查main函数，发现之前tm写的fork ()，在其子进程中调用了路径函数，而在fork中又漏写了 block_desk_init函数，导致数组未初始化，引

起的一系列异常

```
static int32_t copy_pcb_vaddrbitmap_stack0(task_struct* child_thread,\n    task_struct* parent_thread){\n    //复制pcb整个页，里面包含进程pcb信息和e级的栈\n    memcpy(child_thread, parent_thread, PG_SIZE);\n\n    child_thread->pid = fork_pid();\n    child_thread->elapsed_ticks = 0;\n    child_thread->status = TASK_READY;\n    child_thread->ticks = child_thread->priority;\n    child_thread->parent_pid = parent_thread->pid;\n    child_thread->general_tag.next = child_thread->general_tag.prev = NULL;\n    child_thread->all_list_tag.prev = child_thread->all_list_tag.next = NULL;\n\n    //bug 忘了初始化子进程的内存块描述符\n    block_desc_init(child_thread->u_block_desc);\n\n    //复制父进程的虚拟地址池位图
```

9.27:

在file_read函数中，由sys_malloc申请的地址，不能通过sys_free释放。

怀疑是arena的头部数据损坏。查看头部数据如下：

```
(0) [0x000000008a3c] 0008:c0008a3c (unk. ctxt): call .-22986 (0xc0003077) ; e836a6ffff\n<bochs:8> n\nNext at t=1316261241\n(0) [0x000000008a41] 0008:c0008a41 (unk. ctxt): mov dword ptr ss:[ebp-40], eax ; 8945d8\n<bochs:9> r\neax: 0x0804940c 134517772\nebx: 0x08049000 134516736\necx: 0x0000000a 10\nedx: 0x00000001 1\nesp: 0xc0137df8 -1072464392\nebp: 0xc0137e60 -1072464288\nesi: 0x00000000 0\nedi: 0x00000000 0\neip: 0xc0008a41\neflags 0x00000086: id vip vif ac vm rf nt IOPL=0 of df if tf SF zf af PF cf\n<bochs:10> x /12bx 0x8049000\n[bochs]:\n0x08049000 <bogus+> 0:> 0xdc 0x70 0x13 0xc0 0x05 0x00 0x00 0x00\n0x08049008 <bogus+> 8:> 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00\n<bochs:11>
```

头部数据正常

在file_read中继续执行后，内存被释放前。查看头部数据已被破坏

```
<bochs:14> x /12bx 0x8049000\n[bochs]:\n0x08049000 <bogus+> 0:> 0x55 0x89 0xe5 0x83 0xec 0x18 0xc7 0x04\n0x08049008 <bogus+> 8:> 0x24 0x00 0xa0 0x04\n<bochs:15>
```

segment_load

```
/**\n * bug 点\n *\n * 问题:\n *\n * 1.\n * 因为vaddr是有磁盘中的程序指定的要加载的目标地址, 当此目标地址与内存中的arena的地址重合时,\n * 将会导致在执行sys_read时, 使用磁盘中的程序数据覆盖当前arena的内存信息, 导致sys_read\n * 中的sys_free执行失败.\n *\n * 2.\n * 也可以这样子想, 因为sys_read中存在sys_malloc函数, 此函数根据当前进程所在的地址空间, 来划分arena,\n * 当进程处于用户模式时, arena也处于用户地址空间. 如果直接使用书中的memcpy函数, 则可能导致arena被破坏\n * 导致后面的sys_free失败.\n *\n *\n * 解决办法:\n *\n * 使用额外的内核缓冲区接收程序中的数据, memcpy覆盖内存中的数据的操作应该留到最后, 也就是在所有涉及到用户地址空间操作之后.\n */
```

bug点2：

意识到缓冲区也应该使用内核空间，否则在释放的时候会导致用户进程内存被释放，影响程序正常运行。如果不释放，则会导致用户进程地址被浪费。
使用get_kernel_pages后，出现pg错误，排查后发现有个长度为0的段要加载，很多内存申请和读取写入文件函数都没有判断为0的情况
给所有函数打补丁，但还是会出缺页异常。

```

/*
 * @brief
 * 将文件描述符fd指向的文件中,偏移为offset大小为filesize的段加载到指定虚拟地址(vaddr)
 * 成功返回true,失败返回false
 */
static bool segment_load(
    uint32_t fd, uint32_t offset, uint32_t filesize, uint32_t vaddr){
    int32_t pg_cnt = DIV_ROUND_UP(filesize, PG_SIZE);
    void* temp_buf = get_kernel_pages(pg_cnt); ←
    assert(temp_buf!=NULL);
    dbg_printf("load addr 0x%x,temp_buf:0x%x,filesize:%d,pg_cnt:%d\n", vaddr, temp_buf, filesize, pg_cnt);
    //虚拟地址所在页的起始地址(页框)
    uint32_t vaddr_fist_page = vaddr&0xfffff000;
    //段加载到内存中后,第一个页框可提供的内存大小
    uint32_t size_in_fist_page = PG_SIZE - (vaddr & 0x00000fff);

    // 1.判断(以vaddr开始)的一个页框是否能容纳整个段
    uint32_t occupy_pages = 0; //被加载段需要的页框数量
    if(filesize > size_in_fist_page){
        //容纳不下,计算需要的页框数量
        uint32_t left_size = filesize - size_in_fist_page;
        occupy_pages = DIV_ROUND_UP(left_size,PG_SIZE);
    }
    occupy_pages++;

    // 2.为写入段准备好可用的vaddr。
}

```

文件相互覆盖:

原因: 位图同步错误

解决: 一个扇区能申请的最大bit_idx是4096(512*8)的倍数, 每次同步512字节

```

/*
 * @brief
 * 同步内存中bitmap的第bit_idx位开始的512字节到硬盘
 * bmp:位图类型
 */
void bitmap_sync(partition* part, uint32_t bit_idx, uint8_t bmp){

    uint32_t off_sec = bit_idx/4096; //保存位图, 相对于位图起始扇区的扇区偏移

    uint32_t off_size = off_sec*512; //被复制的位图字节偏移

    uint32_t sec_lba;
    uint8_t* bitmap_off;

    switch(bmp){
        case INODE_BITMAP:
        {
            sec_lba = part->sb->inode_bitmap_lba+off_sec;
            bitmap_off = part->inode_bitmap.bits+off_size;
            break;
        }
        case BLOCK_BITMAP:
        {
            sec_lba = part->sb->block_bitmap_lba+off_sec;
            bitmap_off = part->block_bitmap.bits+off_size;
            break;
        }
    }
    ide_write(part->my_disk, sec_lba, bitmap_off, 1);
}

```

键盘输入:

上下键会导致其他按键一直触发无效异常

已解决,未对多扫描码按键的 ext_scancode 变量进行还原。

file_write:

重复写入文件,文件之间相互覆盖, 怀疑是位图问题

已解决,位图同步有两处错误。

删除键:

使用删除键后, 光标不能正常显示, 输入字符后, 光标恢复正常。

未解决, 光标位置通过端口写入正常, 但是不闪烁, 不清楚是否是显卡设置的原因。

用户程序:

没能正确拼接参数路径

未解决, 因为是临时代码所以没有去解决

shell:

连续输入两次不存在的文件名, 会导致fork启动并且产生异常

已解决,不应该按照书上的方式去写, 这样会导致不存在的文件也会启动一个进程,修改一下shell中的逻辑顺序

至于fork进程名异常是因为fork会在当前进程名后面拼接"_fork",所以fork次数太多后, 就会导致进程名太长触发异常。

完善MBR

2022年4月13日 14:58

MBR基础(代码部分)

2022年4月13日 15:06

```
;主引导程序
;因为blo通过jmp 0:0x7c00跳转到此处,此时cx=0,利用cx初始化其他寄存器。
;初始化栈,0x7c00以下的地址随便使用。
SECTION MBR vstart=0x7c00
    mov ax,cx
    mov ds,ax
    mov es,ax
    mov ss,ax
    mov fs,ax
    mov ax,0xb800
    mov sp,0x7c00
;中断例程, 不做详细解释, 参数书籍p72页。
;清屏
    mov ax,0x600
    mov bx,0x700
    mov cx,0
    mov dx,0x184f
    int 0x10
;获取光标位置
    mov ah,3
    mov bh,0
    int 0x10
;打印字符串
    mov ax,message
    mov bp,ax
    mov cx,5
    mov ax,0x1301
    mov bx,0x2
    int 0x10
;等待
;填充0, 以补足510个字节, 让0x55和0xaa在0盘0道1扇区的最后2个字节
    jmp $
message db "1 MBR"
times 510-($-$) db 0
db 0x55,0xaa
```

直接操作显存

2022年4月13日 14:52

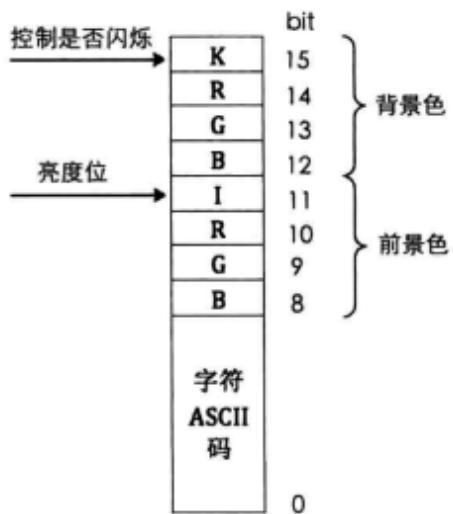
表 3-15

显存地址分布

起 始	结 束	大 小	用 途
C0000	C7FFF	32KB	显示适配器 BIOS
B8000	BFFFF	32KB	用于文本模式显示适配器
B0000	B7FFF	32KB	用于黑白显示适配器
A0000	AFFFF	64KB	用于彩色显示适配器

显示器默认是 80x25大小

一个字符，占用2个字节。如下图所示：



▲图 3-13 字符及其属性

操作显存代码部分

2022年9月13日 15:17

显存地址分布

起 始	结 束	大 小	用 途
C0000	C7FFF	32KB	显示适配器 BIOS
B8000	BFFFF	32KB	用于文本模式显示适配器
B0000	B7FFF	32KB	用于黑白显示适配器
A0000	AFFFF	64KB	用于彩色显示适配器

; 王引导程序

; 因为 bios 通过 jmp 0:0x7c00 跳转到此处，此时 cx=0，利用 cx 初始化其他寄存器。

; 初始化栈，0x7c00 以下的地址随便使用。

SECTION MBR vstart=0x7c00

```
    mov ax,cx
    mov ds,ax
    mov es,ax
    mov ss,ax
    mov fs,ax
    mov ax,0xb800
    mov gs,ax
    mov ax,cx
    mov sp,0x7c00
```

; 中断例程，不做详细解释，参数书籍 p72 页。

; 清屏

```
    mov ax,0x600
    mov bx,0x700
    mov cx,0
    mov dx,0x184f
    int 0x10
```

; 获取光标位置

```
    mov ah,3
    mov bh,0
    int 0x10
```

; 打印字符串

```
    mov byte [gs:0x0], 'H'
    mov byte [gs:0x1], 0x8C
    mov byte [gs:0x2], 'E'
    mov byte [gs:0x3], 0x8C
    mov byte [gs:0x4], 'L'
    mov byte [gs:0x5], 0x8C
    mov byte [gs:0x6], 'L'
    mov byte [gs:0x7], 0x8C
    mov byte [gs:0x8], 'O'
    mov byte [gs:0x9], 0x8C
```

; 等待

; 填充 0，以补足 510 个字节，让 0x55 和 0xaa 在 0 盘 0 道 1 扇区的最后 2 个字节

```
    jmp $
    message db "1 MBR"
    times 510-($-$) db 0
    db 0x55,0xaa
```

; 10001100

硬盘

2022年4月13日 15:56

表 3-17

硬盘控制器主要端口寄存器

IO 端口		端口用途	
Primary 通道	Secondary 通道	读操作时	写操作时
Command Block registers			
0x1F0	0x170	Data	Data
0x1F1	0x171	Error	Features
0x1F2	0x172	Sector count	Sector count
0x1F3	0x173	LBA low	LBA low
0x1F4	0x174	LBA mid	LBA mid
0x1F5	0x175	LBA high	LBA high
0x1F6	0x176	Device	device
0x1F7	0x177	Status	Command
Control Block registers			
0x3F6	0x376	Alternate status	Device Control

data寄存器:

相当于数据的门，数据通过此门进出，是一个**16位**寄存器。

Error和Features寄存器:

读硬盘时,如果发生错误，此寄存器保存错误信息，Sector count保存尚未读取的扇区数量。

写硬盘时，Features保存额外的参数。

8位寄存器

Sector count:

指定等待读取或写入的扇区数,硬盘每完成一个扇区后，就会将此寄存器的值-1。

8位寄存器

硬盘的寻址方式:

CHS:

使用柱面-磁头-扇区来定位。

LBA28:

使用28位bit来描述一个扇区地址,最大寻址范围2的28次方=128GB

LBA48:

48位bit描述一个扇区地址,128PB

寻址相关的寄存器(都是8位寄存器):

LBAlow,LBAmid,LBAhigh = 24位

device寄存器的低四位是LBA地址的高4位 =4位

Device Flags:

4:

指定通道上的主盘或从盘(0为主盘)

6:

是否启用LBA寻址,1表示启用

5,7:

MBS位，固定为1

Status和Command寄存器：

读取(Status):

0位是ERROR位，如果此位为1，表示命令出错。

3位是data request位，如果此位为1，表示硬盘已经把数据准备好了。

6位DRDY位，表示硬盘已经准备就绪，诊断使用

7位BSY位，1表示硬盘繁忙

写入(Command):

存放让硬盘执行的命令。

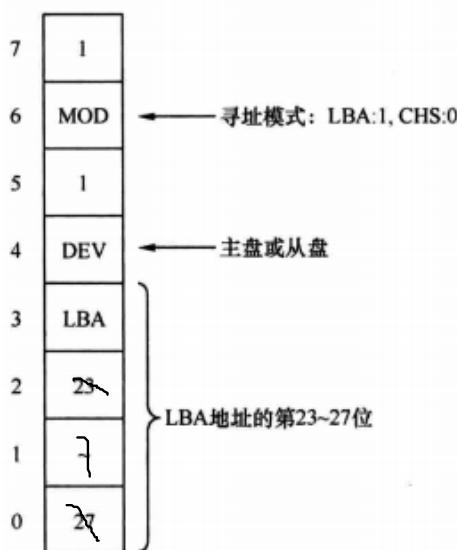
我们用到的命令：

identify:0xec 硬盘识别

read sector: 0x20 读扇区

write sector: 0x30 写扇区

端口示意图：



▲图 3-31 device 寄存器



▲图 3-32 status 寄存器

OUT 和 IN 在汇编中是端口读写操作指令

硬盘读取操作步骤：

1. 选择通道，往该通道的 Sector count 写入待操作的扇区数。
2. 往该通道上的三个LBA寄存器写入扇区的起始地址的低24位。
3. 往device寄存器写入LBA地址的高四位，设置LBA寻址，选择主盘或从盘。
4. 往Command寄存器写入操作命令。
5. 读取status寄存器，判断硬盘工作是否完成。

硬盘写入操作(接着上面的继续)：

将硬盘数据读出。

如何读出硬盘数据呢？

1.无条件传送方式

直接传送

2.查询传送方式

你准备好了吗？没有。你准备好了吗？好了。开始传输

3. 中断传送方式

我准备好了！我准备好了！

4. DMA

使用(硬件)

5. I/O处理机

使用全自动玩具

实模式下的内存布局：

实模式下的内存布局			
起始	结束	大小	用途
FFFF0	FFFFF	16B	BIOS 入口地址，此地址也属于 BIOS 代码，同样属于顶部的 640KB 字节。只是为强调其入口地址才单独贴出来。此处 16 字节的内容是跳转指令 jmp f000: e05b
F0000	FFFEF	64KB-16B	系统 BIOS 范围是 F0000~FFFFF 共 640KB，为说明入口地址，将最上面的字节从此处去掉了，所以此处终止地址是 0XFFEF
C8000	EFFFF	160KB	映射硬件适配器的 ROM 或内存映射式 I/O
C0000	C7FFF	32KB	显示适配器 BIOS
B8000	BFFFF	32KB	用于文本模式显示适配器
B0000	B7FFF	32KB	用于黑白显示适配器
A0000	AFFFF	64KB	用于彩色显示适配器
9FC00	9FFFF	1KB	EBDA (Extended BIOS Data Area) 扩展 BIOS 数据区
7E00	9FBFF	622080B 约 608KB	可用区域
7C00	7DFF	512B	MBR 被 BIOS 加载到此处，共 512 字节
500	7BFF	30464B 约 30KB	可用区域
400	4FF	256B	BIOS Data Area (BIOS 数据区)
000	3FF	1KB	Interrupt Vector Table (中断向量表)

<https://blog.csdn.net/jed>

接下来就是改造我们的MBR, 规定如下：

磁盘(LBR)的2扇区，放置Kernel Loader程序。

内存，0x500的位置放置Kernel Loader程序。

MBR(代码部分)

2022年4月14日 15:37

```
;主引导程序  
;因为bios通过jmp 0:0x7c00跳转到此处,此时cx=0,利用cx初始化其他寄存器。  
;初始化栈,0x7c00以下的地址随便使用。  
%include "bot.inc"  
SECTION MBR vstart=0x7c00  
    mov ax,cs  
    mov ds,ax  
    mov es,ax  
    mov ss,ax  
    mov fs,ax  
    mov sp,0x7c00  
    mov ax,0xb800  
    mov gs,ax  
  
    mov ax,0600h  
    mov bx,0700h  
    mov cx,0  
    mov dx,184fh  
    int 0x10  
  
    mov eax,LOADER_START_SECTOR  
    mov bx,LOADER_BASE_ADDR  
    mov cx,4  
    call rd_disk_m_16  
    int 3  
    jmp LOADER_BASE_ADDR  
;  
;读取硬盘n个扇区  
;eax = LBA扇区地址  
;bx=数据写入的内存地址  
;cx=读入的扇区数  
rd_disk_m_16:  
;  
;选择通道,往该通道的 Sector count 写入待操作的扇区数。  
    mov dx,0x1f2  
    push eax  
    mov ax,cx  
    out dx,al  
    pop eax  
;  
;往该通道上的三个LBA寄存器写入扇区的起始地址的低24位。  
    mov dx,0x1f3  
    out dx,al  
  
    shr eax,8  
    mov dx,0x1f4  
    out dx,al  
  
    shr eax,8  
    mov dx,0x1f5  
    out dx,al  
  
;  
;往device寄存器写入LBA地址的高四位, 设置LBA寻址, 选择主盘或从盘。
```

```

shr eax,0x8
and al,0x0f
or al,0xe0
mov dx,0x1f6
out dx,al
;往Command寄存器写入操作命令。
mov dx,0x1f7
mov al,0x20
out dx,al
;读取status寄存器，判断硬盘工作是否完成。
.not_ready:
in al,dx
and al,0x8
cmp al,0x8
jnz .not_ready

;读取cx个扇区,一个扇区512字节，一次读出2个字节,读取次数 = cx*512/2 =cx*0x100
mov ax,0x100
mul cx
mov cx,ax
mov dx,0x1f0 ;这里有一个坑啊,因为乘积的高位会保存到dx,所以dx的赋值要留到mul指令
之后
.go_on_read:
in ax,dx
mov [bx],ax
add bx,2
loop .go_on_read
ret

;填充0, 以补足510个字节, 让0x55和0xaa在0盘0道1扇区的最后2个字节
message db "1 MBR"
times 510-($-$) db 0
db 0x55,0xaa
;10001100

```

保护模式入门

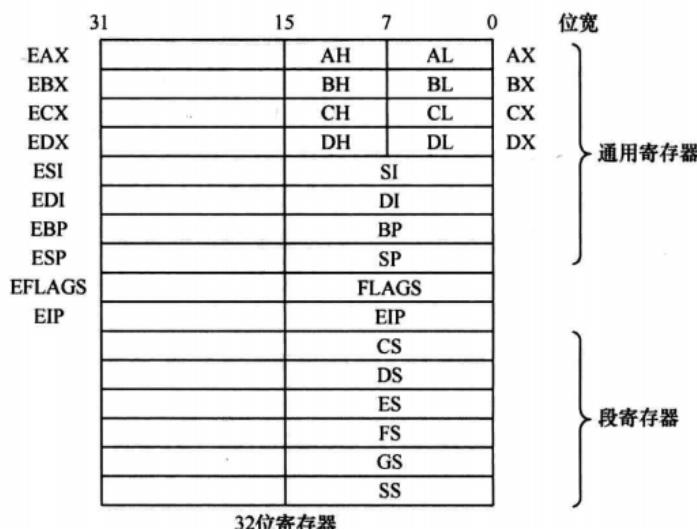
2022年4月14日 15:48

保护模式概述

2022年4月14日 15:49

实模式是指运行在32位cpu下的16位兼容模式，当然也可以执行32位的指令

保护模式寄存器扩展：



系统中只存在一个全局描述符表,系统在处理器切换到保护模式时创建全局描述符表,表的基址存放在**GDTR**(全局描述符表寄存器)里面.表中的项目(**段描述符**)指向各个段.

段描述符:

大小:64字节

内容:访问该段的权限, 该段的权限, 起始地址等等

此时段寄存器也不再保存段基址了, 而是保存**段选择子**

段选择子:

还用我说嘛, 自己去翻看x32内核

剩下的就是分析段选择子中的数据, 找到对应的表, 然后像查数组一样去查询到段描述符就OK了。

DCR(段描述符缓冲寄存器):

cpu获取到内存段信息后, 存入段描述符缓冲器, 以后每次访问相同的段时, 就直接读取该段寄存器对应的段描述符缓冲寄存器。

因为GDT是在内存中的, 操作内存很慢。

实模式下的寻址：

基址寄存器只能是bx、 bp

变址寄存器只能是si、 di

bx的默认段寄存器是ds， 用于访问数据段， bp默认的段寄存器是ss，
它用于访问栈。

初见保护模式

2022年4月19日 9:26

段描述符缓冲寄存器的失效时间:

保护模式下，只要往段寄存器中赋值，cpu就会更新段描述符缓冲器，例如，在保护模式下加载选择子，CPU就会重新访问全局描述符表。

三种描述符缓冲寄存器结构：

80286 处理器														
47~32	31	30~29	28	27~24	23~0	偏移量								
Limit	P	DPL	S	Type	base	字段								
80386/80486 处理器														
95~64	63~32	31~24	23	22~21	20	19~16	15	14	13~0	偏移量				
limit	base	0	P	DPL	S	Type	0	D/B	0	字段				
奔腾处理器														
95~79	78	77~72	71	70~69	68	67~64	63~32	31~0	偏移量					
0	D/B	0	P	DPL	S	type	base	Limit	字段					

▲图 4-2 段描述符缓冲寄存器结构

保护模式下的寻址扩展：

实模式下：基质寄存器只能是bx、bp,变址寄存器只能是si、di。

保护模式下：基质寄存器可以是任何寄存器，变址寄存器除了esp之外可以是任何寄存器

保护模式之运行模式反转：

CPU在实模式下，并不是变成了纯粹的16位CPU，还是能使用32位特性的。

指定编译位数：

bits 16	下面的代码编译成16位机器码
bits 32	下面的代码编译成32位机器码

反转前缀：

0x66 :

16位实模式，操作数大小将变为32位

32位保护模式，操作数大小将变为16位

使用反转指令在保护模式下push word:

77DBE9ED | 66:68 2301 | push 123

段寄存器和通用寄存器，无论是在保护模式下还是实模式下，都是push相应位数。

总结：

在保护模式使用16位寄存器或者push 两个字节，则使用反转前缀指令完成。
实模式下亦然。

全局描述符表

2022年4月19日 10:09

全局描述符表(GDT)是保护模式下内存段的登记表，这是不同于实模式的显著特征之一。

段描述符(8字节大小):



▲图 4-5 段描述符格式

段基质:

就是段基质

段界限 (Limit) :

段边界的扩展最值，最大扩展到多少和最小扩展到多少，一共有向上和向下两种方向，分别对应数据段、代码段和栈段。

段界限只是个单位量，单位要么是字节要么是4KB，由G位指定。G=0，粒度=1byte,G=1，粒度=4KB。

ps:因为段界面的下标是从0开始，也就是说0也代表一个范围(在4KB粒度下)所以段界限要+1。

计算公式：

$$\text{段界限边界值} = (\text{段界限} + 1) * \text{粒度} + 1$$

段描述符各字段解析:

S:

S=0,系统段:

各种“门”结构便是系统段，也就是硬件系统需要的结构，非软件使用的。

S=1,数据段:

凡是软件(包括操作系统)需要的东西都称为数据。

s字段必须和TYPE字段组合在一起才有具体的意义。

TYPE:

表 4-10 段描述符的 type 类型

系统段类型	第 3~0 位				说明	
	3	2	1	0		
未定义	0	0	0	0	保留	
可用的 80286 TSS	0	0	0	1	仅限 286 的任务状态段	
LDT	0	0	1	0	局部描述符表，只有第 1 位为 1	
忙碌的 80286 TSS	0	0	1	1	仅限 286。type 中的第 1 位称为 B 位，若为 1，则表示当前任务忙碌。由 CPU 将此位置 1	
80286 调用门	0	1	0	0	仅限 286	
系统段	任务门	0	1	0	1	任务门在现代操作系统中很少用到
	80286 中断门	0	1	1	0	仅限 286
	80286 陷阱门	0	1	1	1	仅限 286
	未定义	1	0	0	0	保留
	可用的 80386TSS	1	0	0	1	386 以上 CPU 的 TSS，type 第 3 位为 1
	未定义	1	0	1	0	保留
	忙碌的 80386 TSS	1	0	1	1	386 以上 CPU 的 TSS，type 第 3 位为 1
	80386 调用门	1	1	0	0	386 以上 CPU 的调用门，type 第 3 位为 1
	未定义	1	1	0	1	保留
	中断门	1	1	1	0	386 以上 CPU 的中断门
	陷阱门	1	1	1	1	386 以上 CPU 的陷阱门

对于非系统段，按代码段和数据段划分，这 4 位分别有不同的意义

非系统段	内存段类型	X	R	C	A	说明
	代码段	1	0	0	*	只执行代码段
		1	1	0	*	可执行、可读代码段
		1	0	1	*	可执行、一致性代码段
		1	1	1	*	可执行、可读、一致性代码段
	内存段类型	X	W	E	A	说明
数据段	数据段	0	0	0	*	只读数据段
		0	1	0	*	可读写数据段
		0	0	1	*	只读，向下扩展的数据段
		0	1	1	*	可读写，向下扩展的数据段

哎呀，这张图的w和E，R和C写反了啊！！！

系统段：

暂时不说

非系统段：

代码段：

A:

Accessed位,由CPU设置，每当该段被CPU访问过后，
CPU 将此位置1。

C:

Conforming。一致性代码段是指如果自己是转移的目标
代码段，并且自己是一致性代码段，自己的特权级一定要
高于当前特权级，转移后的特权级不能与自己的DPL为
主，而是与转移前的低特权级一致，也就是听从、依从转
移前的特权等级，C=1,表示该段是一致性代码段，C=0
时，表示该段非一致性代码段。

R:

R=1,表示可读,R=0,表示不可读

X:

x=1，代码可执行,x=0代码不可执行

E:

标识段的扩展方向, E=0, 标识向上扩展, 即地址越来越高, 通常用于代码段和数据段。E=1, 表示向下扩展, 地址越来越低, 通常用于栈段。

W:

指此段是否可写, W=1, 表示可写, W=0, 表示不可写。

DPL:

描述符特权级别, 用于描述一个内存段或一段代码的情况。

P:

Present, 段是否存在, 如果存在于内存中, P=1, 如果不存在则为0, 如果为0, CPU将抛出异常, 转到相应的异常处理程序, 此异常处理程序是我们自己写的。

AVL:

AVailAble, 可用的

L:

设置是否是64位代码段, L=1表示64位代码段, 否则表示32位代码段。

D/B:

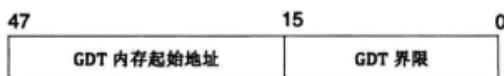
用来指示(段内偏移地址)及操作数的大小, 为了兼容286的16位保护模式, 操作数的大小相关的是指令, 与指令相关的内存段是代码段和栈段, 所以此字段是D(16位模式中的dx)或B(16位模式中的bx)。
对于代码段, 此位是D位, D=0, 表示指令中的有效地址及操作数都是16位
对于数据段, 此位是B位, B=0, 表示栈相关的操作都是16位

全局描述符表GDT:

全局描述符表GDT相当于是描述符的数组, 数组中的每个元素都是8字节的描述符。使用选择子中提供的小标在GDT中索引描述符。

为什么叫“全局”:

因为多个程序都可以在里面定义自己的段描述符, 是公用的, CPU通过GDTR寄存器存储GDT的内存地址和大小。GDTR是一个48位的寄存器



▲图 4-6 GDTR 寄存器

GDTR的初始化:

使用"lgdt"指令对GDTR进行初始化, 在实模式和保护模式下都可以使用这条指令, 可以在保护模式下重新指定GDT的位置。

lgdt指令格式:

lgdt 48位内存数据

段选择子:

段寄存器在实模式下存储的是段的基质, 即内存段的起始地址, 但是在保护模式下, 段基质存储在段描述符中, 段描述符又存在GDT中, 所以段寄存器不在存储段基质, 而是变成了段选择子。

因为段寄存器是16位, 所以选择子也是16位,

0-1位, RPL:

请求特权级, 请求者的当前特权级别。

2位, TI:

0表示在GDT中的索引, 1表示在LDT中的缩影。

3-15位:

段选择子的索引值。

ps:GDT中第0个段描述符是不可用的，如果段选择位初始化，选择值的值便会是0，为了避免错误的访问，CPU将触发异常。

LDT:

local Descriptor Table,它是CPU厂商在硬件一级原生支持多任务而创造的表，

按照CPU的设想，一个任务对应一个LDT。很少使用

LDTR寄存器指向LDT。

lldt指令初始化LDTR。

lldt指令格式：

lldt 16位寄存器/16位内存

因为ldt也存放在一块内存中，需要使用一个段描述符指向它，而段选择子用于描述段描述符。

TSS任务段中将会使用到LDT的内容。

系统段:

每个系统段都有各自的描述符。任何描述符的大小都是8字节。但无论描述符的种类有多少，它们的高32位中的8-12字节都是不变的。12位为S,8-11位为TYPE。中断门、陷阱门、任务门等都是在中断描述符表中，不在全局描述符表中。

打开A20地址线:

什么是A20地址线：

8086只有20根地址总线但逻辑上能访问的内存超过了寻址的范围(A0-A19),为了兼容8086在访问超过1MB时出现的地址回绕现象，IBM所设计的一个功能,即操作第21根地址线(A20)的有效性。

当A20Gate被打开时,cpu将正常访问到0x100000~0x10FFFF之间的内存。

当A20Gate被关闭时,CPU将采用8086的地址回绕。

打开方法:

将端口0x92的第一位置1

```
in al,0x92  
or al,0000_0010B  
out 0x92,al
```

保护模式的开关，CR0的PE位：

控制寄存器CRx。控制寄存器是CPU的窗口，既可以展示CPU的内部状态，也可以用于控制CPU的运行机制。

CR0:

控制寄存器CR0

■ 保留位

表 4-11

控制寄存器 CR0 字段

标 志 位	描 述
PE	Protection Enable
MP	Monitor coProcessor/Math Present
EM	Emulation
TS	Task Switched
ET	Extention Type
NE	Numeric Error
WP	Write Protect
AM	Alignment Mask
NW	Not Writethrough

WP	Write Protect
AM	Alignment Mask
NW	Not Writethrough
CD	Cache Disable
PG	Paging

PE:

启用保护

EM:

模拟

TS:

任务切换

ET:

扩展类型

WP:

写保护

AM:

对齐

CD:

禁用缓存

PG:

分页

当PE位为1时，系统将在保护模式下运行。

启用方式:

```
mov eax,cr0
or eax,0x00000001
mov cr0,eax
```

段寄存器	代码段 (X=1)		数据段 (X=0)	
	只执行 (R=0)	执行+可读 (R=1)	只读 (R=1, W=0)	读写 (W=1)
CS	通过	通过	不通过	不通过
DS	不通过	通过	通过	通过
ES	不通过	通过	通过	通过
FS	不通过	通过	通过	通过
GS	不通过	通过	通过	通过
SS	不通过	不通过	不通过	通过

KernelLoad(代码部分)

2022年4月22日 10:18

```
;内核加载器的内存起始地址
LOADER_BASE_ADDR equ 0x900
;内核加载器所在的扇区(LBR)
LOADER_START_SECTOR equ 0x2
;定义段描述符高32位,常用各字段值
DESC_BASE_HIGH1_ZERO equ 0x0
DESC_G_4K equ 1_0000000000000000000000000000000b
DESC_DB_32 equ 1_0000000000000000000000000000000b
DESC_L_32 equ 0_0000000000000000000000000000000b
DESC_AVL_ENABLE equ 1_0000000000000000000000000000000b
DESC_LIMIT_HIGH1_ZERO equ 0
DESC_LIMIT_HIGH1_ALL equ 1111_0000000000000000000000000000000b
DESC_P_ENABLE equ 1_0000000000000000000000000000000b
DESC_DPL0 equ 0
DESC_DPL1 equ 01_000000000000000b
DESC_DPL2 equ 10_000000000000000b
DESC_DPL3 equ 11_000000000000000b
DESC_S_DATA equ 1_000000000000000b
DESC_S_SYETM equ 0_000000000000000b
DESC_TYPE_CODE_EXECUTION equ 1000_000000000b
DESC_TYPE_DATA_RW equ 0010_000000000b
DESC_BASE_LOW2_ZERO equ 0
;定义显存段描述, 字段值
DESC_VIDEO_BASE_HIGH1 equ DESC_BASE_HIGH1_ZERO
DESC_VIDEO_LIMIT_HIGH1 equ DESC_LIMIT_HIGH1_ZERO
DESC_VIDEO_BASE_LOW2 equ 0000_1011b
;定义默认段描述符, 高32位的部分
;代码段 可执行 4GB寻址
DESC_CODE_HIGH4 equ DESC_BASE_HIGH1_ZERO+DESC_G_
4K+DESC_DB_32+DESC_L_32
+DESC_AVL_ENABLE+DESC_LIMIT_HIGH1
_ALL+DESC_P_ENABLE+DESC_DPL0
+DESC_S_DATA+DESC_TYPE_CODE_EXECUTION+DESC_BASE_L_
OW2_ZERO
;数据段 可读可写 4GB寻址
DESC_DATA_HIGH4 equ DESC_BASE_HIGH1_ZERO+DESC_G_
4K+DESC_DB_32+DESC_L_32
+DESC_AVL_ENABLE+DESC_LIMIT_HIGH1
_ALL+DESC_P_ENABLE+DESC_DPL0
+DESC_S_DATA+DESC_TYPE_DATA_RW+DESC_BASE_L_
OW2_ZERO
;显存段 0xB8000-0xBFFF 可读可写
DESC_VIDEO_HIGH4 equ DESC_VIDEO_BASE_HIGH1
+DESC_G_4K+DESC_DB_32+DESC_L_32
+DESC_AVL_ENABLE+DESC_VIDEO_LIMIT_HIGH1
+DESC_P_ENABLE+DESC_DPL0
+DESC_S_DATA+DESC_TYPE_DATA_RW+DESC_VIDEO_BASE_L_
OW2_ZERO
;定义选择子字段值
RPL0 equ 00b
RPL1 equ 01b
RPL2 equ 10b
RPL3 equ 11b
TI_GDT equ 000b
TI_IDT equ 100b
;打开A20
in al,0x92
or al,0000_0010b
out 0x92,al
;加载GDT
lgdt [gdtr_ptr]
;CR0的PE置1
mov eax,cr0
or eax,0x00000001
mov cr0,eax
;刷新流水线
jmp dword SELECTOR_CODE:p_mode_start
[bits 32]
p_mode_start:
;初始化各选择子
mov ax,SELECTOR_DATA_STACK
mov ds,ax
mov es,ax
mov ss,ax
mov esp,LOADER_STACK_TOP
mov ax,SELECTOR_VIDEO
mov gs,ax
;输出字符 protected mode
mov byte [gs:160],"p"
mov byte [gs:162],"r"
mov byte [gs:164],"o"
mov byte [gs:166],"t"
```

```
mov byte [gs:168],"e"
mov byte [gs:170],"c"
mov byte [gs:172],"t"
mov byte [gs:174],"e"
mov byte [gs:176],"d"
mov byte [gs:178]," "
mov byte [gs:180],"m"
mov byte [gs:182],"o"
mov byte [gs:184],"d"
mov byte [gs:186],"e"
jmp $
```

保护模式进阶

2022年4月22日 10:19

调用步骤:

```
ax寄存器写入0x88  
int 0x15  
if cf=0;统计结果
```

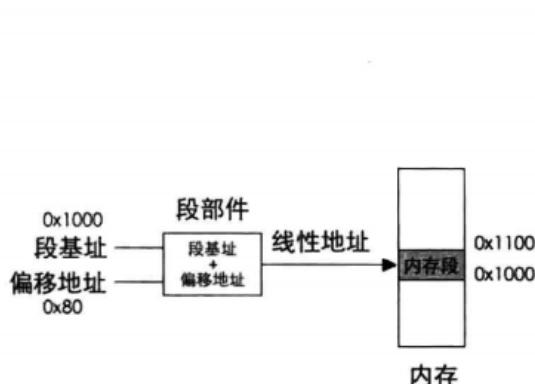
启用分页机制

2022年4月22日 15:50

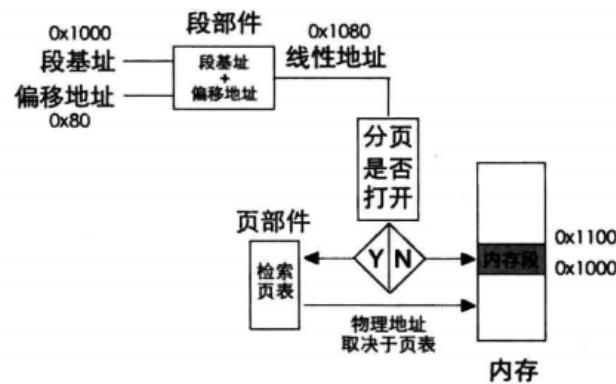
分页机制总结：

分页机制是由CPU提供的硬件支持，支持连续的虚拟地址映射到非连续的物理地址。分页机制建立在分段机制之上。

分段机制和分页机制：



▲图 5-6 内存分段机制下地址访问

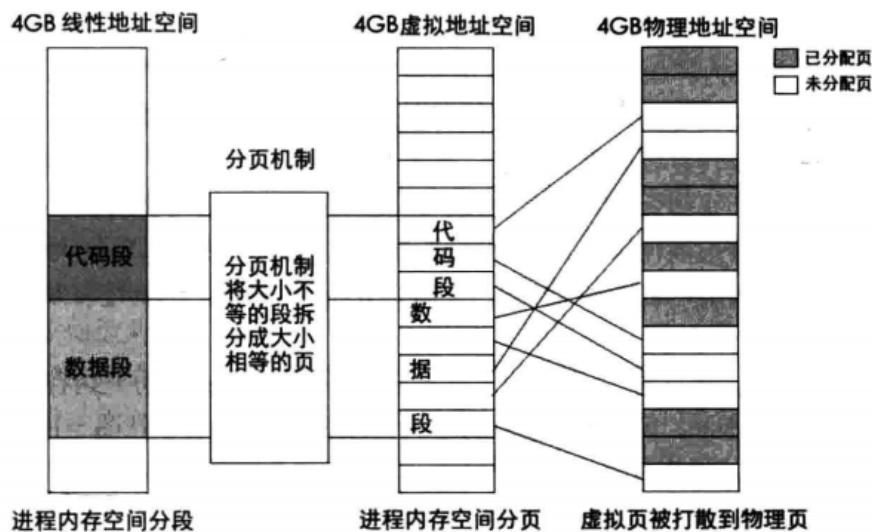


▲图 5-7 分页机制

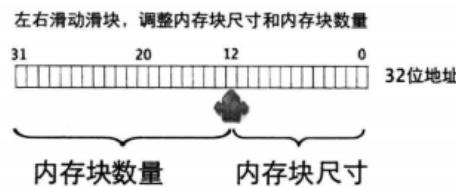
分页机制的作用：

将线性地址转换为物理地址

用大小相等的页代替大小不等的段



▲图 5-8 分页机制的作用



▲图 5-10 寻找合适的页尺寸

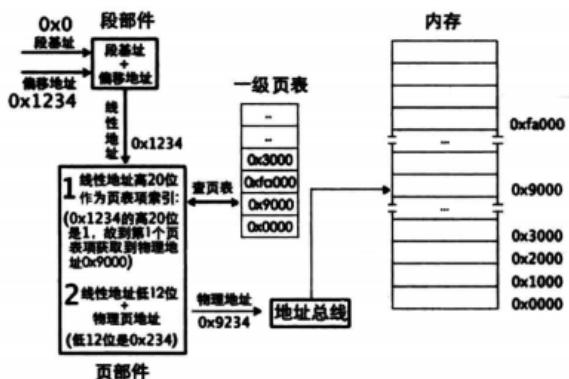
CPU中采用的,内存块大小(一页的大小)是2的12次方 4KB,内存块(页表项)的数量2的20次方1M个。

一级页表：

因为总的页表项是2的20次方个。所以可以使用20位二进制来表示一个页表项在页表(页表定义为连续的)中的下标。而一个页的大小为4k，所以可

以用2的12次方表示这个页指向的物理地址内的任何位置的偏移。所以指向一级页表的值，刚好可以用32位表示，前20位为页表项在页表中的下标，后12位为此页指向的物理起始地址的偏移。

一级页表转换图：



▲图 5-12 通过一级页表将线性地址转换成物理地址过程

二级页表：

在理解二级页表前，先梳理一下，每个标准页的大小是4KB不变。二级页表的操作就是总共1024个的标准页，平均放置在1024个页表中，每个页表分到1024个标准页的地址，一个地址的大小为4字节所以。 $4 \times 1024 = 4096$ ，一个页表刚好就是4KB的大小。可以使用一个东西来表示这一整个页，对吧？（这个东西呢就是接下来的页目录表项）

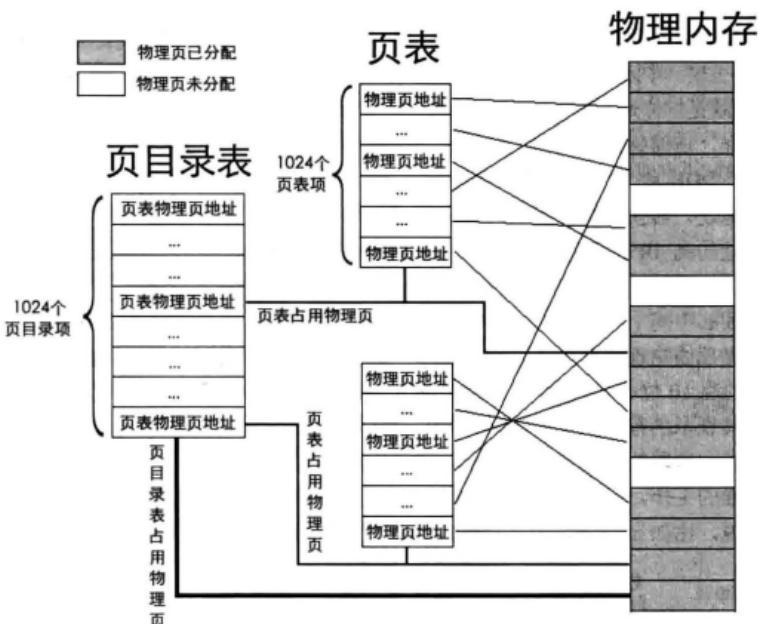
此时使用页目录表保存这1K个页表，每一个页目录表项(Page Directory Entry,PDE)都保存一个页表物理页地址。

同时页目录表和页表，本身也自己占用物理内存。

2022.5.25 ps:

每个进程都有自己的页目录表，一个页目录表最多可以指向4GB的物理内存地址。

二级页表图解：



▲图 5-13 二级页表占用内存示意

ps:因为每个物理内存页，都是4k大小即0x1000,所以在16进制表示下，物理内存页的地址后三位总是0。

页目录项:PDE

页表项:PTE

101012分页机制：

使用页目录表的物理地址+虚拟地址的高10位*4=页表的起始物理地址

页表的起始物理地址+虚拟地址的中间10位*4=物理内存块的起始地址。

物理内存块+后12位=虚拟地址对应的物理地址。

PDE和PTE:

31	12	11	9	8	7	6	5	4	3	2	1	0
页表物理地址31~12位	AVL	G	0	D	A	PCD	PWT	US	RW	P		

页目录项

31	12	11	9	8	7	6	5	4	3	2	1	0
物理页地址31~12位	AVL	G	PAT	D	A	PCD	PWT	US	RW	P		

页表项

▲图 5-16 页目录项及页表项

问题：

不是应该用32位表示物理地址吗？，为什么低12位可以被定义为属性位。

答：

因为每个物理页都是按照4K的倍数划分的，即0x1000的倍数，所以低12位肯定是0，这时候就可以利用这低12位添加其他属性。

各字段解析：

P:

Present, 存在位。若为1表示该页存在于物理内存中，若为0表示该表不在物理内存中。操作系统的页式虚拟内存管理便是通过P位和响应的pageFault异常来实现的。

RW:

ReadWrite,意为读写位。若为0表示可读不可写，若为1表示可读可写。

US:

User/Supervisor,意为 普通用户/超级用户位。若为1，表示处于User级，任意级别程序都可以访问，若为0，特权级别为3的程序不允许访问该页，其他特权级的可以访问。

PWT:

页级通写位，若为1，表示该页不仅是内存还是高速缓存。

PCD:

页级高速缓存静止位，若为1表示启用高速缓存，为0表示禁止将该页缓存

A:

Accessed,意为访问位，用于CPU统计该页内存的访问率。

D:

Dirty,意为脏页位，CPU对一个页面进行写操作时，就会将设置对应页表项的D位为1，此项仅针对页表项有效。

PAT:

页属性表位，能够在页面一级的粒度上设置内存属性。比较复杂，置0

G:

Global,全局位，将虚拟地址与物理地址的转换结果存放在TLB中，下次使用时不需要再转换，直接查表的得到物理地址。1表示是全局页，0表示不是全局页。

清空TLB的方法：

invlpg:清空一个虚拟地址

重载cr3: 清空TLB

AVL:

available, 可用位，针对于软件和操作系统

启用分页内存步骤：

1.规划页目录表和页表

2.将页目录表(最顶级的页表)写入到Cr3

3.寄存器Cr0的Pg位置1(Cr0的第31位)

CR3寄存器：

用于存储页表的物理地址，页目录基址存储器

31	12	11	5	4	3	2	1	0
页目录表物理地址 31~12 位			PCD	PWT				

除了PCD和PWT位，其余位都没有用，所以只需要将低12置空，高20位填充页目录表的高20位(因为低12位起始也是0)。

规划：

用户使用低3GB空间，系统使用高1GB空间。

物理地址(0x100000)开始存放页目录表和页表，可以是自定义的任何地址。

```

<bochs:219> info tab
cr3: 0x0000000100000
0x00000000-0x000fffff -> 0x00000000000000-0x00000000ffff
0xc0000000-0xc00fffff -> 0x000000000000-0x00000000ffff
0xffffc00000-0xfc00ffff -> 0x000000101000-0x000000101fff
0xffff00000-0xffffefff -> 0x000000101000-0x0000001fffff
0xfffff0000-0xffffffff -> 0x000000100000-0x000000100fff
<bochs:220> ^[A]

```

这是最后出来的结果,说明一下:

因为页目录表的第一项和偏移0xc00处相同(我们设置的),所以前两项指向的地址相同。

第三个:

1111111111 0000000000 000000000000

即将定位到页目录的最后一项(我们设置的指向页目录的首物理地址),所以它指向的物理地址,就是第一个页表的物理地址。

第四个:

1111111111 1100000000 000000000000

即指向页目录的最后一项,将页目录表当做页表后,又指向0xc00偏移处(指向的第一个页表),当做物理地址

第五个:

不再说明

总结:

在我们设计的页目录表和页表中

将前10位置为1,即可将页目录表当做页表使用。

即前10位为1,中间为需要定位的页表下标(不需要*4)后面为页表中的页表项(需要手动*4)

TLB:

CPU使用虚拟地址的高20位作为索引来查找TLB中的相关条目

虚拟地址的高 20 位 (虚拟页框号)	属性	物理地址的高 20 位 (物理页框号)
...		
...		

▲图 5-25 TLB 结构简图

TLB的更新应由开发人员手动更新。

更新方式:

invlp m 更新指定虚拟地址的TLB,例子:

重新加载cr3

加载内核

2022年4月24日 9:14

gcc编译指令:

```
gcc -m32 -c -o 目标文件路径 源文件路径
-c:
直接生成目标文件，不进行链接。
-o:
输出的文件以指定文件名来存储，有同名时覆盖
-m32:
指定32位编译
ld -m elf_i386 ./Kernel/kernel.o -Ttext 0xc0001500 -e main -o ./Kernel/kernel.bin
-Ttext:
指定文件载入内存的起始地址。
-e:
指定入口点地址或符号名称(默认名称为 _start )
-m elf_i386:
以32位的方式进行编译
```

linux命令:

```
file 文件路径:
查看文件属性
nm:文件路径:
查看文件重定位信息
|wc -l:
统计输出的行数
```

elf格式的二进制文件:

表 5-7 elf 眼中的目标文件

ELF 目标文件类型	描述
待重定位文件 (relocatable file)	待重定位文件就是常说的目标文件，属于源文件编译后但未完成链接的半成品，它被用于与其他目标文件合并链接，以构建出二进制可执行文件或动态链接库。为什么称其为“待重定位”文件呢？原因是在该目标文件中，如果引用了其他外部文件（其他目标文件或库文件）中定义的符号（变量或者函数统称为符号），在编译阶段只能先标识出一个符号名，该符号具体的地址还不能确定，因为不知道该符号是在哪个外部文件中，而该外部文件需要被重定位后才能确定文件内的符号地址，这些重定位的工作是需要在连接的过程中完成的
共享目标文件 (shared object file)	这就是我们常说的动态链接库。在可执行文件被加载的过程中被动态链接，成为程序代码的一部分
可执行文件 (executable file)	经过编译链接后的、可以直接运行的程序文件

程序中有很多段，也有很多节，段是由节来组成的，多个节经过链接后合并成一个段。

程序(段)头表:

program header table ,存储多个程序头

节头表:

section header table,存储多个节头

这两个数组中的元素称为 条目,entry

用一个固定的数据结构来表述程序头和节表头的大小以及位置信息。这边是ELF header，它位于文件最开始的部分，并且具有固定大小。



▲图 5-34 elf 文件格式布局

elf header自定义的数据类型:

path: /usr/include/elf.h

表 5-8

elf header 中的数据类型

数据类型名称	字节大小	对齐	意义
Elf32_Half	2	2	无符号中等大小的整数
Elf32_Word	4	4	无符号大整数
Elf32_Addr	4	4	无符号程序运行地址
Elf32_Off	4	4	无符号的文件偏移量

ELF 32位文件头:

```
typedef struct
{
    unsigned char      e_ident[EI_NIDENT]; /* Magic number and other info */
    Elf32_Half        e_type;           /* Object file type */
    Elf32_Half        e_machine;       /* Architecture */
    Elf32_Word        e_version;       /* Object file version */
    Elf32_Addr        e_entry;          /* Entry point virtual address */
    Elf32_Off         e_phoff;          /* Program header table file offset */
    Elf32_Off         e_shoff;          /* Section header table file offset */
    Elf32_Word        e_flags;          /* Processor-specific flags */
    Elf32_Half        e_ehsize;         /* ELF header size in bytes */
    Elf32_Half        e_phentsize;      /* Program header table entry size */
    Elf32_Half        e_phnum;          /* Program header table entry count */
    Elf32_Half        e_shentsize;      /* Section header table entry size */
    Elf32_Half        e_shnum;          /* Section header table entry count */
    Elf32_Half        e_shstrndx;       /* Section header string table index */
} Elf32_Ehdr;
```

ELF 32位文件头,各字段解析:

e_ident:

表 5-9

e_ident 数组功能简介

e_ident 数组成员	意 义
e_ident[0] = 0x7f	这 4 位是固定的 ELF 文件的魔数，如果它们的值如左列 4 行所示，表明这就是一个 ELF 文件
e_ident[1] = 'E'	
e_ident[2] = 'L'	
e_ident[3] = 'F'	
e_ident[4]	用来标识 elf 文件的类型 值为 0 表示该文件是不可识别类型 值为 1 表示该文件是 32 位 elf 格式的文件 值为 2 表示该文件是 64 位 elf 格式的文件
e_ident[5]	用来指定编码格式，其实就是指定大端字节序还是小端字节序 值为 0 表示非法编码格式 值为 1 表示小端字节序，即 LSB（最低有效字节） 值为 2 表示大端字节序，即 MSB（最高有效字节）
e_ident[6]	ELF 头的版本信息，默认为 1 值为 0 表示非法版本 值为 1 表示当前版本
e_ident[7~15]	暂且不用，保留，均初始化为 0

e_type:

0	ET_NONE	未知目标文件格式
1	ET_REL	可重定位文件
2	ET_EXEC	可执行文件
3	ET_DYN	动态共享目标文件
4	ET_CORE	程序崩溃转储文件
0xff00	ET_LOPROC	特定处理器文件的扩展下边界
0xffff	ET_HIPROC	特定处理器文件的扩展上边界

e_machine:

elf 目标文件所属的体系结构类型

体系结构类型	取 值	意 义
EM_NONE	0	未指定
EM_M32	1	AT&T WE 32100
EM_SPARC	2	SPARC
EM_386	3	Intel 80386
EM_68K	4	Motorola 68000
EM_88K	5	Motorola 88000
EM_860	7	Intel 80860
EM_MIPS	8	MIPS RS3000

e_version:

版本信息

e_entry:

程序入口点地址

e_phoff:

程序头表在文件内的字节偏移量

e_shoff:

节头表在文件内的字节偏移量

e_flags:

cpu 相关标志

e_ehsize:

elf header 的大小
e_phentsize:
 指定程序头表中每个条目的字节大小
e_phnum:
 程序头表中条目的数量。段的个数
e_shnum:
 节头表中条目的数量，节的个数
e_shstrndx:
 指明 string name table 在节头表中的索引

程序头表:

```
typedef struct
{
    Elf32_Word p_type;           /* Segment type */
    Elf32_Off  p_offset;         /* Segment file offset */
    Elf32_Addr p_vaddr;          /* Segment virtual address */
    Elf32_Addr p_paddr;          /* Segment physical address */
    Elf32_Word p_filesz;         /* Segment size in file */
    Elf32_Word p_memsz;          /* Segment size in memory */
    Elf32_Word p_flags;          /* Segment flags */
    Elf32_Word p_align;          /* Segment alignment */
} Elf32_Phdr;
```

各字段解析:

p_type:

类 型	取 值	说 明
PT_NULL	0	忽略
PT_LOAD	1	可加载程序段
PT_DYNAMIC	2	动态链接信息
PT_INTERP	3	动态加载器名称
PT_NOTE	4	一些辅助的附加信息
PT_SHLIB	5	保留
PT_PHDR	6	程序头表
PT_LOPROC	0x70000000	此范围内的类型预留给处理器专用
PT_HIPROC	0x7fffffff	

p_offset:

本段在文件内的起始偏移字节

p_vaddr:

本段在内存中的起始虚拟地址

p_paddr:

保留

p_filesz:

本段在文件中的大小

p_memsz:

本段在内存中的大小

p_flags:

本段相关的标志

表 5-13

p_flags 取值范围

类 型	取 值	说 明
PF_X	1	本段具有可执行权限
PF_W	2	本段具有可写权限
PF_R	4	本段具有可读权限
PF_MASKOS	0x0ff00000	本段与操作系统相关
PF_MASKPROC	0xf0000000	本段与处理器相关

p_align:

本段在内存中的对齐方式, 如果为0或1则不对齐, 否则应该是2的幂次数

特权级

2022年4月27日 9:40

特权级概述:

将计算机世界分为两类,访问者和受访者

访问者:

访问者是动态的,具有能动性,可以主动的访问各种资源,特权级可变

受访者:

静态的,被访问的资源,特权级不可变

TSS: 任务状态段

31	15	0
110位图在TSS中的偏移地址	(保留)	100
(保留)	ldt选择子	96
(保留)	gs	92
(保留)	fs	88
(保留)	ds	84
(保留)	ss	80
(保留)	cs	76
(保留)	es	72
	edi	68
	esi	64
	ebp	60
	esp	56
	ebx	52
	edx	48
	ecx	44
	eax	40
	eflags	36
	eip	32
	cr3(pdbr)	28
(保留)	SS2	24
	esp 2	20
(保留)	SS1	16
	esp1	12
(保留)	SS0	8
	esp 0	4
(保留)	上一个任务的 TSS 指针	0

32位TSS结构

TSS是处理器厂商提供的支持多任务的一种实现方式,但是windows自己实现了线程管理并没有使用TSS,只在权限切换的时候使用了TSS。

ss0:esp0 - ss2:esp2:

分别代表了0-2环的段选择子和栈地址。

为什么有4个特权级但是只保存了3个段选择子和栈地址?

Because:

TSS是用来记录权限切换,分为以下两种情况:

从低到高时,最多只有三种情况(3-2,2-1,1-0)

从高到低时,有来才有回,所以之前肯定经历过从低到高,3环的段选择子和栈地址已经被保存到高特权级的栈中了,返回时只需要retf即可。

TSS中的栈指针都是固定的,不会保存上一次的栈地址。

CPU如何找到TSS?

通过TR寄存器,每次处理器执行不同任务时,将TR寄存器加载不同任务的TSS即可。

CPL和DPL和RPL:

RPL:

段选择子的0、1位,因为具有访问者的身份,所以叫请求特权级,。

CPL:

CS段选择子的RPL,因为cpu执行的就是cs:eip处的代码,cpu执行的代码就是访问者,也就是cpu当前的特权级,也就是说处理当前特权级是CS.RLP(CPL)。

DPL:

段描述符中的DPL,表示访问该段内存所需要的权限,访问请求通过后,使用DPL给CPL赋值。

不提升特权级执行特权代码的方法:

使用一致性代码段:

目标段的权限大于等于当前的CPL,也就是CPL>=目标段的DPL。但是不会使用目标段的DPL改变调用者的CPL
ps:数据段不存在一致性,即不允许比自己DPL低的代码段访问。

门、调用门与RPL序:

中断门:

31	16 15 14 13 12 11	8 7 6 5 4	0	高 32 位
中断处理程序在目标段内的偏移量的 31~16位	P DPL S TYPE 0 0 0	未使用		
31	16 15	0		低 32 位
中断处理程序目标代码段描述符选择子	中断处理程序在目标代码段内的偏移量的 第15~0位			

D位为0表示16位模式
D位为1表示32位模式

中断门描述符格式

陷阱门:

31	16 15 14 13 12 11	8 7 6 5 4	0	高 32 位
中断处理程序在目标段内的偏移量的 31~16位	P DPL S TYPE 0 0 0	未使用		
中断处理程序目标代码段描述符选择子	中断处理程序在目标代码段内的偏移量的 第15~0位			
D位为0表示16位模式 D位为1表示32位模式				

陷阱门描述符格式

调用门:

31	16 15 14 13 12 11	8 7 6 5 4	0	高 32 位
被调用例程在目标代码段内的偏移量的 第31~16位	P DPL S TYPE 0 0 0	参数个数		
31	16 15	0		低 32 位
被调用例程所在代码段的描述符选择子	被调用例程在目标代码段内的偏移量的 第15~0位			

D位为0表示16位模式
D位为1表示32位模式

调用门描述符格式

任务门:

31	16 15 14 13 12 11	8 7	0	高 32 位
未使用	P DPL S TYPE 0 0 1 0 1	未使用		
31	16 15	0		低 32 位
TSS选择子	未使用			

任务门描述符格式

门所在的位置:

调用门:

GDT、IDT

使用方式:

call far
jmp far

中断门和陷阱门:

IDT

使用方式:

int 和 int3

任务门:

GDT、LDT、IDT

使用方式:

```
call far  
jmp far
```

除了任务门之外，其他门都指定了目标段选择子和段偏移，所以调用这三个门时会忽略调用时的偏移
门的DPL \geq 调用者的CPL。

目标的DPL \leq 调用者的CPL，不然的就会产生降权，这是不被允许的。

调用门传参:

call 调用门之前将参数压入当前特权级别的栈中，调用门描述符中指定参数个数，硬件将自动复制参数到目标特权栈中。

32位下调用门的进入过程:

- 1.当前栈压入参数
- 2.检查当前CPL和调用门的DPL以及目标段的DPL
- 3.确定目标段的特权，检查目标栈段的type和DPL(防止降权)
- 4.如果目标DPL高于当前CPL，则进行提权(段间转移)，压入ss_old,esp_old,参数,cs_old,eip_old
- 5.加载目标段的段选择子和段偏移+段基址
- 6.执行调用例程
- 7.由开发者指定是否使用retf返回
- 8.如果是平级调用，则只压入cs和eip。

32位下调用门的返回过程:

- 1.执行到retf时，返回低特权级别
- 2.检查cs_old段选择子的RPL，和返回代码处的DPL.
- 3.如果调用调用门时使用了参数，则需要在retf参数个数*参数大小(4Byte)
- 4.恢复esp_old,ss_old
- 5.检查数据寄存器指向的段描述符的DPL是否低于返回地址处的CPL，如果是则清空相应的数据寄存器。

为什么会将数据段寄存器清空?

因为调用门没有保存数据段寄存器，而寄存器的更新只在于赋值的时候，调用门返回时不会更新数据寄存器，为了避免越权访问，清空数据寄存器后，如果不赋值，则会指向GDT的第一个段描述符，触发CPU异常。

什么是RPL?

如果只使用DPL和CPL，当3环程序通过门提升自己的CPL后，就可以无限制的访问内核任何地方，为了保证内核的安全性，所以需要另外定义一个权限标识符，表明调用者真实的权限，即RPL，访问者原始特权级别。

所以在检查特权级别的时候，需要检查RPL、CPL与目标的DPL

在访问资源时，在数值上 DPL \geq RPL、CPL，则可以通过特权安全检查

如果用户伪造RPL怎么办?

对于门来说，cpu分两步检查特权级别:

- 1.RPL \leq 门的DPL
- 2.CPL \leq 门的DPL

即使伪造了RPL，因为CPL无法改变，也无法通过权限检查。

如果门的DPL为3是否就可以通过检查呢?

必要时，即使用户伪造了RPL为0并且门的DPL为3，在通过权限检查进入内核后，系统会使用访问者的CPL替换目标段的RPL，这样也可以避免越权访问

I/O特权级:

在保护模式下，不仅代码和数据分特权，IO指令同样有特权之分。

特权指令:

只能在0特权级下使用，否则触发异常

第一次异常于 00401742 (C0000096, EXCEPTION_PRIV_INSTRUCTION)!

例如：

hlt (使计算机停机),lgdt,lidt,ltr,popf

IO读写控制:

IO读写特权有标志寄存器eflags中的IOPL位和TSS中的IO位图控制，它们用来指定指向IO操作的最小特权级。

in、out、cli、sti

31...	21	20	19	18	17	16	15	14	13-12	11	10	9	8	7	6	5	4	3	2	1	0
保留	ID	VIP	VIF	AC	VM	RF	NT	IOPL	OF	DF	IF	TF	SF	ZF	AF	PF	CF				

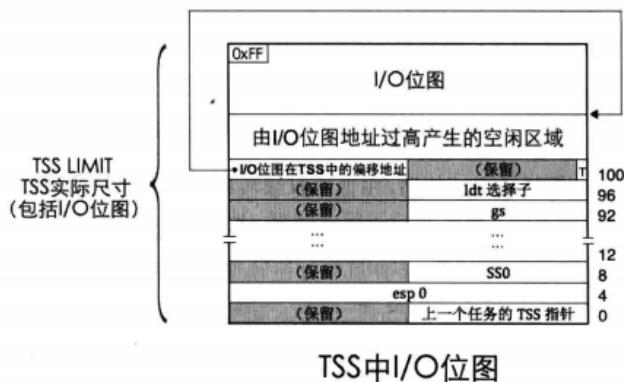
▲图 5-60 寄存器的 IOPL 位

只能通过POPF、lretd修改eflags前16位，如果不是在r0执行的该指令，cpu不会修改前16位。

eflags中的IOPL是所有端口的开关，每个任务都有自己的eflags。

io位图用于控制某个端口开关，即使在数值上cpl>lopl，只要io位图中的对应端口号打开，也可以访问。

CPU如何找到IO位图？



▲图 5-62 TSS 中的 I/O 位图

TSS的大小，如果存在IO位图，其大小为“IO位图偏移地址+8192+1”字节，最后这个1字节是IO位图的最后一字节0xff
为什么最后是0xff？

因为在计算机系统硬件中，IO端口是按字节来编址的，一个端口只能读写一个字节的数据，而 in 指令可以一次读取2字节，其实其本质操作如下：

原指令：

in ax, 0x1234

cpu拆分后：

in al,0x1234

in ah,0x1235

当端口号位于io位图的最后一一位(0x65534)时，cpu检查其IO位图时将会越界，又因为内存操作的最小单位为1字节，所以将其最后一个字节定位0xff，即使访问越界其端口bit也是为1(表示不可访问)。

如何证明IO位图不存在？

如果IO位图偏移地址不在TSS段描述符的limit范围之内，则表明没有IO位图。

PS:

在数值上如果CPL<IOPL，则无视io位图。

完善内核

2022年4月27日 10:18

汇编与c的混合调用

2022年4月28日 9:28

汇编语言与c的混合编程分为两大类:

- 1.单独的汇编代码文件与单独的c语言文件分别编译后，一起链接成可执行文件。
- 2.在c语言中嵌入汇编代码

```
nasm -f elf -o syscall_asm.o syscall_asm  
gcc -m32 -c -o syscall_c.o syscall_c.c  
ld -m elf_i386 -o syscall_c.bin syscall_c.o syscall_asm.o
```

linux的系统调用:

系统调用入口:

0x80中断，利用eax传递功能号。

定义文件:

/usr/include/x86_64-linux-gnu/asm/unistd_???.h

查看调用号:

man 2 系统调用名

直接使用功能号调用:

eax存放功能号

参数小于5个时，

ebx存放第1个参数。ecx存放第二个参数，edx存储第三个删除，esi，edi分别是第4和第五个参数。

导出和导入关键字:

导出:

global

导入:

extern

实现自己的打印函数

2022年5月6日 15:03

显卡的端口控制:

显卡中的端口：

表 6-2

VGA 寄存器

寄存器分组	寄存器子类	读端口	写端口	说 明
Graphics Registers	Address Register	3CEh		
	Data Register	3CFh		
Sequencer Registers	Address Register	3C4h		
	Data Register	3C5h		
Attribute Controller Registers	Address Register	3C0h		
	Data Register	3C1h	3C0h	
CRT Controller Registers	Address Register	3x4h		x 的值取决于 Input/Output Address Select 字段, 它决定映射的端口号为 3B4h-3B5h 或 3D4h-3D5h
	Data Register	3x5h		
Color Registers	DAC Address Write Mode Register	3C8h		
	DAC Address Read Mode Register	不可用	3C7h	
	DAC Data Register	3C9h		
	DAC State Register	3C7h	不可用	
External (General) Registers	Miscellaneous Output Register	3CCh	3C2h	
	Feature Control Register	3CAh	3xAh	写端口为 3BAh (mono 模式) 或 3DAh (color 模式)
	Input Status #0 Register	3C2h	不可用	Read-only at 3C2h
	Input Status #1 Register	3xAh	不可用	读端口为 3BAh (mono 模式) 或 3DAh (color 模式)

前四个是寄存器的分组,分为Address Register和Data Register.

为什么端口要分组?

因为读写端口时, 只能使用bx寄存器指定端口,ax和al接收数据,bx的最大范围是0xffff(65535+1),使用分组后将每一个寄存器分组为一个寄存器数组, 提供一个寄存器用于指定数组下标, 再提供一个寄存器用于对索引所指向的数组元素进行读写操作。通过两个寄存器定位寄存器数组的任何寄存器。

Address Register和Data Register:

Address Register作为数组的索引

Data Register 用来保存读或写的数据

CRT Controller 特例:

CRT Controller寄存器的端口并不是固定的, 具体值取决于Miscellaneous Output Register 寄存器的 Input/Output Address Select字段。

Miscellaneous Output Register 寄存器:

表 6-3 Miscellaneous Output Register (读端口 3CCh, 写端口 3C2h)

7	6	5	4	3	2	1	0
VSYNCP	HSYNCP	O/E Page		Clock Select	RAM En.	I/OAS	

各个字段详细解释:

表 6-4

Miscellaneous Output Register 各字段英文描述

字 段	描 述
VSYNCP (Vertical Sync Polarity)	"Determines the polarity of the vertical sync pulse and can be used (with HSP) to control the vertical size of the display by utilizing the autosynchronization feature of VGA displays. = 0 selects a positive vertical retrace sync pulse."
HSYNCP (Horizontal Sync Polarity)	"Determines the polarity of the horizontal sync pulse. = 0 selects a positive horizontal retrace sync pulse."
O/E Page (Odd/Even Page Select)	"Selects the upper/lower 64K page of memory when the system is in an eve/odd mode (modes 0,1,2,3,7). = 0 selects the low page = 1 selects the high page"
Clock Select	This field controls the selection of the dot clocks used in driving the display timing. The standard hardware has 2 clocks available to it, nominally 25 Mhz and 28 Mhz. It is possible that there may be other "external" clocks that can be selected by programming this register with the undefined values. The possible value of this register are: 00 -- select 25 Mhz clock (used for 320/640 pixel wide modes) 01 -- select 28 Mhz clock (used for 360/720 pixel wide modes) 10 -- undefined (possible external clock) 11 -- undefined (possible external clock)
RAM En. (RAM Enable)	"Controls system access to the display buffer. = 0 disables address decode for the display buffer from the system = 1 enables address decode for the display buffer from the system"
I/OAS (Input/Output Address Select)	"This bit selects the CRT controller addresses. When set to 0, this bit sets the CRT controller addresses to 0x03Bx and the address for the Input Status Register 1 to 0x03BA for compatibility with the monochrome adapter. When set to 1, this bit sets CRT controller addresses to 0x03Dx and the Input Status Register 1 address to 0x03DA for compatibility with the color/graphics adapter. The Write addresses to the Feature Control register are affected in the same manner."

最重要的字段 I/OAS:

当此位为0时候:

CRT controller寄存器的端口地址被设置为0x3Bx。input Status #1 Register 的端口地址被设置为 0x3BA。

当此位为1时候:

CRT controller寄存器的端口地址被设置为0x3Dx。input Status #1 Register 的端口地址被设置为 0x3Dx。

Feature Control register也一样被控制。

总的来说,为0时,所有带x的寄存器x都被替换为B, 为1时,被替换为D

表 6-5

CRT Controller Data Registers

索 引	寄 存 器	索 引	寄 存 器
00h	Horizontal Total Register	0Dh	Start Address Low Register
01h	End Horizontal Display Register	0Eh	Cursor Location High Register
02h	Start Horizontal Blanking Register	0Fh	Cursor Location Low Register
03h	End Horizontal Blanking Register	10h	Vertical Retrace Start Register
04h	Start Horizontal Retrace Register	11h	Vertical Retrace End Register
05h	End Horizontal Retrace Register	12h	Vertical Display End Register
06h	Vertical Total Register	13h	Offset Register
07h	Overflow Register	14h	Underline Location Register
08h	Preset Row Scan Register	15h	Start Vertical Blanking Register
09h	Maximum Scan Line Register	16h	End Vertical Blanking
0Ah	Cursor Start Register	17h	CRTC Mode Control Register
0Bh	Cursor End Register	18h	Line Compare Register
0Ch	Start Address High Register		

实现单个字符打印:



print

```
main.c
1 #include "../lib/kernel/print.h"
2 // debug virtual address 0008:0xc0001500
3 void main(void){
4     printf("\nprinntf sucess\n");
5     put_int32(12345);
6     printf("\n");
7     put_int32(0x78910);
8     printf("\n");
9     put_int32(0xABCD);
10    printf("\n");
11    printf(" \n");
12    printf("travel kernel world \n");
13    while (1)
14    {
15    };
16
17
18 }
```

The protected mode is started successfully
prinntf sucess
0x3039
0x78910
0xABCD
travel kernel world

内联汇编

2022年5月10日 15:50

什么是内联汇编:

GCC支持在c代码中直接嵌入汇编代码，在gcc内联汇编中，不支持inter汇编语法，只支持AT&T语法。

AT&T语法简介:

项目	AT&T	Intel	说明
寄存器命名	%eax	eax	Intel的不带%
操作数顺序	movl %eax, %edx	mov edx, eax	将eax的值赋值给edx
常数\立即数	movl \$3, %eax movl \$0x10, %eax	mov eax, 3 mov eax, 0x10	将3赋值给eax, 将0x10赋值给eax AT&T的常数加\$前缀
jmp指令	jmp *%edx jmp *0x4001002 jmp *(%eax)	jmp edx jmp 0x4001002 jmp [eax]	在AT&T的jmp地址前面要加星号*
操作数长度	movl %eax, %edx movb \$0x10, %al leaw 0x10(%dx), %ax	mov edx, eax mov al, 0x10 lea ax, [dx + 0x10]	b = byte (8-bit) s = short (16-bit integer or 32-bit floating point) w = word (16-bit) l = long (32-bit integer or 64-bit floating point) q = quad (64 bit) t = ten bytes (80-bit floating point)
访问内存高度	后缀b、w、l表示字节、字、长型	前缀byte ptr, word ptr, dword ptr	
引用全局或静态变量var的值	_var	[_var]	
引用全局或静态变量var的地址	\$_var	_var	
引用局部变量	基于栈指针 (ESP)	基于栈指针 (ESP)	
内存直接寻址	imm(base, index, indexscale)	[base + index * indexscale + imm]	两种结果的实际寻址都是 imm + base + index * indexscale
立即数变址寻址	-4(%ebp)	[ebp - 4]	
整数数组寻址	0x40014(%eax, 3)	[0x40014 + eax * 3]	
寄存器变址寻址	0x40014(%ebx, %eax, 2)	[ebx + eax * 2 + 0x40014]	
寄存器间接寻址	movw \$.%, %ds: (%eax)	mov word ptr ds:[eax], 6	

AT&T简单结构:

```
asm [volatile](  
    "assembly code"  
)
```

assembly code规则:

- 1.指令必须使用双引号引其阿里，无论双引号中是一条指令还是多条指令。
 - 2.一对双引号不能跨行，如果跨行需要在结尾用\'转义
 - 3.指令之间用";"分隔

荔枝：

```
char* str = "AT&T\n";
void main(void ){
    asm(
        "movl $4,%eax;\\" 
        "movl $1,%ebx;\\" 
        "movl str,%ecx;\\" //只能引用全局变量,如果是私有变量则会提示找不到符号。
        "movl $5,%edx;\\" 
        "int $0x80"
    );
}
```

扩展内联汇编:

格式·

```
asm [volatile] ("assembly code":output :input :clobber/modify )
```

output:

用来指定汇编代码的数据如何输出给c代码使用,使用此项指定输出的位置。

格式:

"操作数修饰符约束名"(c变量名)

output的操作数修饰符通常为"=".多个操作数之间用逗号 "," 分隔。

input:

指定c中数据如何输入给汇编使用

格式:

"[操作数修饰符] 约束名"(c变量名)

clobber/modify:

通知编译器, 可能造成寄存器或内存数据的破坏

寄存器约束名:

要求gcc使用指定寄存器, 将input或output中变量约束在某个寄存器中。

寄存器必须使用两个%%前缀。

a: 表示寄存器 eax/ax/al

b: 表示寄存器 ebx/bx/bl

c: 表示寄存器 ecx/cx/cl

d: 表示寄存器 edx/dx/dl

285

6章 完善内核

D: 表示寄存器 edi/di

S: 表示寄存器 esi/si

q: 表示任意这 4 个通用寄存器之一: eax/ebx/ecx/edx

r: 表示任意这 6 个通用寄存器之一: eax/ebx/ecx/edx/esi/edi

g: 表示可以存放到任意地点(寄存器和内存)。相当于除了同 q 一样外, 还可以让 gcc 安排在内存中

A: 把 eax 和 edx 组合成 64 位整数

f: 表示浮点寄存器

t: 表示第 1 个浮点寄存器

u: 表示第 2 个浮点寄存器

尝试使用内联扩展汇编写一个加法:

```
#include "stdio.h"
void main(void ){
char* str = "AT&T\n";
int out_int = 0;
int in_a = 3;
int in_b = 2;
asm("addl %%eax,%%ebx;":"=b"(out_int):"a"(in_a),"b"(in_b));
printf("sum:%d\n",out_int);
}
```

内存约束:

要求gc直接将input和output中的c变量的内存地址作为内联汇编代码的操作数, 汇编代码的操作数就变成了c变量的指针>m:

表示操作数可以使用任意一种内存形式

o:

操作数位内存变量, 单方问他是通过偏移量的形式访问。

n%:

占位符,从0-n, 表示input参数的序号

尝试写一个交换值:

```
#include "stdio.h"
```

```

void main(void ){
char* str = "AT&T\n";
int out_int = 0;
int in_a = 3;
int in_b = 5;
asm("movl %0,%1;:::c"(in_a),"m"(in_b));
printf("sum:%d\n",in_b);
}

```

ps:如果添加了output，则会出错，值不会进行更改

ps:错误原因，因为占位符是包括output的，所以存在output时，0就表示输出，当然就会出错啦。

立即数约束:

- i: 表示操作数为整数立即数
- F: 表示操作数为浮点数立即数
- I: 表示操作数为 0~31 之间的立即数
- J: 表示操作数为 0~63 之间的立即数
- N: 表示操作数为 0~255 之间的立即数
- O: 表示操作数为 0~32 之间的立即数
- X: 表示操作数为任何类型立即数

通用约束:

0~9:次约束只用作input部分，但表示可与output和input中第N个操作数用相同的寄存器或内存。

占位符:

当我们使用 r 约束符时，gcc编译器将替我们安排寄存器，此时我们失去了对寄存器的掌控，这时使用占位符来引用操作数。

占位符分为序号占位符和名称占位符两种

序号占位符:

序号占位符是对在output和input中的操作数,按照他们从左到右有出现的次序从0开始编号，一直到9.

引用符号为 %0~9

重点:

占位符指代约束所对应的操作数(寄存器或内存),不是圆括号中的c变量

在%和序号之间插入字符“h”表示操作数为ah，插入“b”表示al

名称占位符:

格式:

[自定义的名称] "约束名"(c变量名)

引用:

%[自定义的名称]

例子:

实现一个除法:

```

#include <stdio.h>
void main(void ){
char* str = "AT&T\n";
int out_int = 0;
int in_a = 3;
int in_b = 6;
asm("movl $0,%edx;div %[divisor];:[retvalue] =a"(out_int):[divisor]"c"(in_a),"a"(in_b));
printf("sum:%d\n",out_int);
}

```

强调:

无论是哪种占位符，它都是指代c变量经过约束后、由gcc分配的对应汇编代码中的操作数(寄存器、内存、立即数等)

操作数类型修改符:

用于修饰小牧所约束的操作数.

在output中:

= :

表示操作数只写,=a(out_var)等价于 movl %eax,out_var

+ :

表示操作数可读可写。+a(out_var)先将out_var的值读入到eax中，内嵌汇编结束时再将eax赋值给out_var
&:

表示output独占这个寄存器，input中不能使用被占用的寄存器

在input中：

%:

该操作数可以和下一个输入操作数互换

h - 输出寄存器高位部分中的那一字节对应的寄存器名称，如 ah、bh、ch、dh。

b - 输出寄存器中低部分 1 字节对应的名称，如 al、bl、cl、dl。

w - 输出寄存器中大小为 2 个字节对应的部分，如 ax、bx、cx、dx。

k - 输出寄存器的四字节部分，如 eax、ebx、ecx、edx。

中断

2022年5月11日 14:20

中断分类

2022年5月11日 14:23

中断是什么？：

打断cpu现在做的事情，让他先去执行别的任务，利用中断能过显著的提示并发，大幅提升效率。
外部中断：



▲图 7-1 外部中断类型

不可屏蔽的中断向量号为2。

内部中断：

内部中断可分为软中断和异常。

软中断：

int 空格 8位立即数,用来表示最多256种中断.

int3:

调试指令

into:

中断溢出指令 向量号4, eflags的of为1才会触发

异常：

是指令执行期间cpu内部产生的错误引起的，属于运行时错误。

Fault:

cpu将及其状态恢复到异常之前的状态，之后调用中断处理程序时，cpu将返回地址指向导致异常的那条指令，如缺页异常。

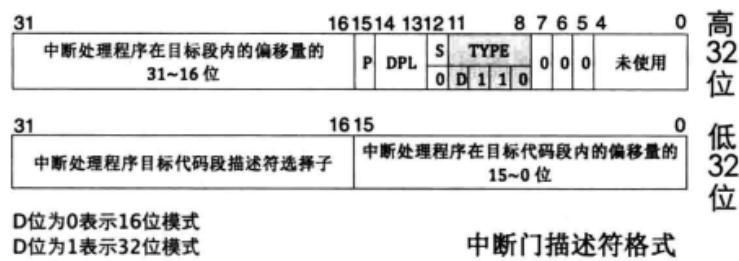
Trap:

软件调用了cpu设下的陷阱之中，好坏坏哦这个cpu，返回地址指向导致异常指令的下一条指令地址。

中断图：

表 7-1 异常与中断

Vector No.	Mnemonic	Description	Source	Type	Error code(Y N)
0	#DE	Divide Error	DIV and IDIV instructions.	Fault	N
1	#DB	Debug	Any code or data reference.	Fault/Trap	N
2	/	NMI Interrupt	Non-maskable external interrupt.	Interrupt	N
3	#BP	Breakpoint	INT3 instruction.	Trap	N
4	#OF	Overflow	INTO instruction.	Trap	N
5	#BR	BOUND Range Exceeded	BOUND instruction.	Fault	N
6	#UD	Invalid Opcode (UnDefined Opcode)	UD2 instruction or reserved opcode.1	Fault	N
7	#NM	Device Not Available (No Math Coprocessor)	Floating-point or WAIT/FWAIT instruction.	Fault	N
8	#DF	Double Fault	Any instruction that can generate an exception, an NMI, or an INTR.	Abort	Y(0)
9	#MF	CoProcessor Segment Overrun (reserved)	Floating-point instruction.2	Fault	N
10	#TS	Invalid TSS	Task switch or TSS access.	Fault	Y
11	#NP	Segment Not Present	Loading segment registers or accessing system segments.	Fault	Y
12	#SS	Stack Segment Fault	Stack operations and SS register loads.	Fault	Y
13	#GP	General Protection	Any memory reference and other protection checks.	Fault	Y
14	#PF	Page Fault	Any memory reference.	Fault	Y
15		Reserved			
16	#MF	Floating-Point Error (Math Fault)	Floating-point or WAIT/FWAIT instruction.	Fault	N
17	#AC	Alignment Check	Any data reference in memory.3	Fault	Y(0)
18	#MC	Machine Check	Error codes (if any) and source are model dependent.4	Abort	N
19	#XM	SIMD Floating-Point Exception	SIMD Floating-Point Instruction5	Fault	N
20~31		Reserved			
32~255		Maskable Interrupts	External interrupt from INTR pin or INT n instruction.	Interrupt	



中断门描述符格式

▲图 7-3 中断门描述符

中断向量表(IDT):

1.IDT的位置不固定

2.IDT的每个描述符使用8字节

中断向量表的位置由IDTR寄存器保存。



中断描述符表寄存器IDTR

IDT最大支持 $(0\text{fff}+1)/8 = 8192$ 个处理程序

但是cpu只支持 int 8位数据 = 0x256个处理程序

加载IDTR:

指令:	lidt 48位内存数据 0-15为IDT limit 后面32位为IDT的基地址
-----	---

中断处理过程以及保护:

中断处理过程,涉及到特权级检查,以及分为cpu外和cpu内两部分

cpu外:

外部设备的中断由中断代理芯片(intel 8259A)接收,处理后将该中断的中断向量号发送到cpu

cpu内:

CPU执行该中断向量号对应的中断处理程序。

我们先讨论cpu内的情况,即软件引起的中断。

1.如何定位中断门描述符

因为在x32中中断描述符是8字节,所以使用中断向量号*2+IDTR中的中断向量表的基址,所得地址便存放有中断描述符.

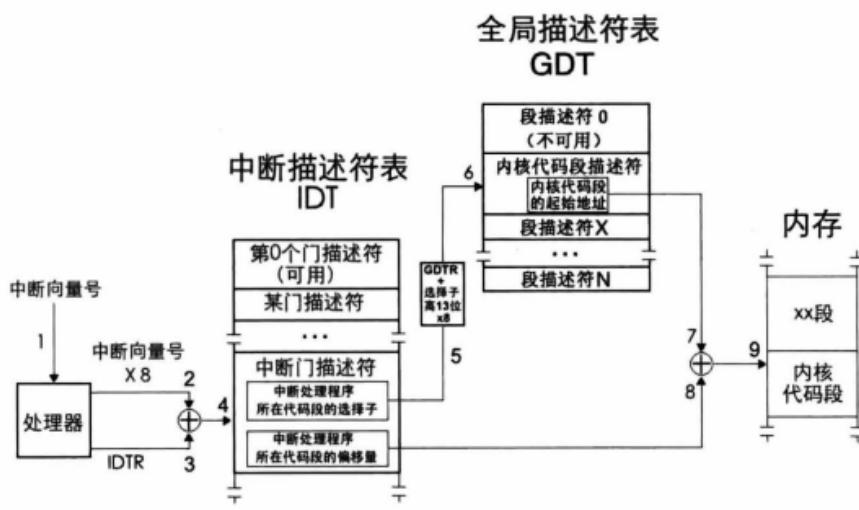
2.特权级检查

中断向量号中并没有RPL,所以只涉及到DPL和CPL的检查

1.如果是由软中断引发的中断,处理器将检查(在数值上)当前CPL<=DPL, CPL>=目标代码段的DPL。重申特权转移只能发生在由低向高,.除了iret等返回指令。如果是硬件引起的则只检查CPL和目标代码的DPL

3.执行中断程序

将目标代码段的段选择子加载到代码寄存器cs中,中断处理程序的偏移加载到eip中。



▲图 7-7 中断处理过程

中断发生后,eflags中的NT位和TF位被置零。

eflags中的if位:

限制外部设备的中断,cli 指令 关中断(if= 0) sti开中断

NT位 (nest task flags) :

任务嵌套位, =1时,表示当前任务是打断别了任务,运行了新的任务,执行完新任务后需要返回至旧任务继续执行。

tss任务段和任务门:

1.将旧任务的tss选择子写入到新任务tss中的“上一个任务tss的指针”

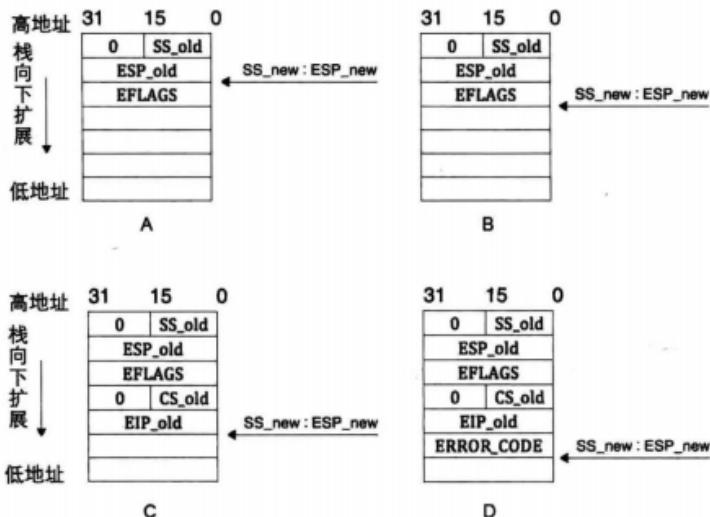
2.将新任务的nt位置1

3.iret返回时,判断是从中断(栈)返回还是从tss中获取返回地址。

x32中断发生时的压栈:

- 1.比较当前的cpl和中断描述符中目标代码段的dpl,如果需要发生提升权限,则通过tss获取到目标代码段的栈地址,则将当前的ss和eip。ss是16位,高位置零
- 2.在新(旧)栈中压入eflags
- 3.在新(旧)栈中压入cs和eip.
- 4.有些异常会有错误代码被压入栈中,用于报告异常是在哪个段中发生的,即Error_Code

ps.如果不会发生权限提升，则不会在tss中寻找新的栈地址，也不会在栈中保存旧的栈选择子和栈顶



中断发生，特权级变化时新栈中的内容

中断返回:

使用“iret”指令依次弹出eip,cs,eflags到寄存器，“iret”指令并不会检查栈中的数据是否正确，所以在返回前要保证数据的正确性，避免当前esp指向例如：Error_Code

iret:

为 iretw 和 iretd 的缩写，有 bits 指令 指定的字长决定。

中断返回时的权限检查:

- 当cpu执行到“iret”指令时候，从栈中检索cs选择子的rpl，判断是否会发生权限转移
- 检查old_cs段选择子指向的段描述符的目标代码段的dpl和old_cs的dpl，检查通过，则更新cs和eip
- 从栈中返回eflags
- 如果在第一步中判断返回时，要改变特权级则从栈中返回新的ss和esp。

ps:

如果中断返回时要改变特权级，检查数据寄存器ds es fs gs 如果（权限上）rpl高于cpl，则置零。

中断错误码：

31	15	3	2	1	0
保留 0	选择子高 13 位索引	TI	IDT	EXT	

错误码 (ERROR_CODE)

▲图 7-10 错误码

形如段选择。

低4位:

EXT:

是否是外部实践，来源于cpu外部=1

IDT:

选择子是否在IDT表中

TI:

是在GDT(= 0)还是LDT(=1) 表中

高位:

段选择子的下标(序号)

会压入错误码的中断向量号：

0-31

外部中断和int 软中断不会产生错误码。

可编程中断控制器8259A

2022年5月11日 14:23

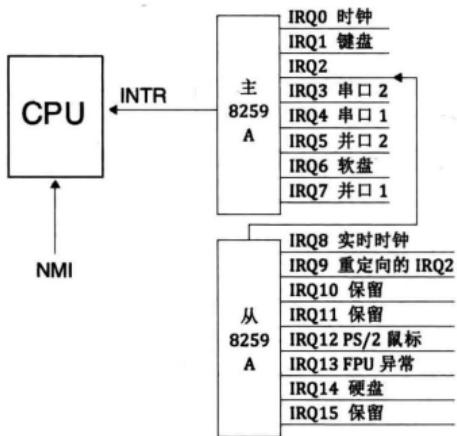
8259A介绍:

负责所有外设的中断，包括时钟中断，以后使用它完成进程的调度。

用于管理和控制可屏蔽中断，表现为屏蔽外设中断，对它们实行优先级判决。向CPU提供中断向量号。

Intel处理器共支持256个中断，但是8259A只支持管理8个中断。使用“级联”这种组合后，最多级联9个，即中断源，其中只有一片8259A为主片(主片)其余的为从片(从片)，来自从片的中断只能传递给主片，再由主片向上传递给CPU(即由主片发送INT中断信号)

中断请求(IRQ: interrupt ReQuest)



▲图 7-11 个人计算机中的级联 8259A

8259A芯片内部结构：

INT:

8259A选出优先级最高的中断请求后，发信号通知CPU

INTA(INT acknowledge):

用于接收CPU接收要中断请求后的响应

IMR(interrupt Mask Register):

中断屏蔽寄存器，宽度8位，用来屏蔽某个外设的中断。

IRR:(interrupt Request Register):

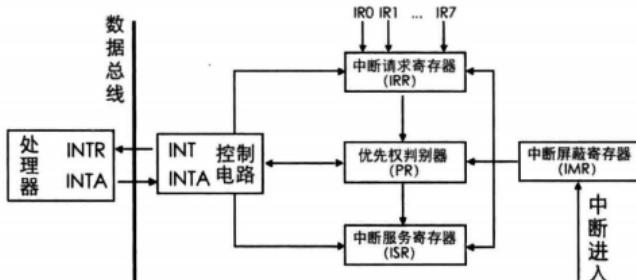
中断请求寄存器，宽度8位，接收IMR过滤后的中断信号并锁存，此寄存器中全是等待处理的中断。类似于消息队列。

PR(priority resolver):

优先裁决器。将正常处理的中断与新的中断进行比较，找出优先级更高的中断。

ISR: in_Service Register:

中断服务寄存器，宽度为8位，保存正则处理的中断。



中断通过8259A的流程:

1. 外设发出中断信号，CPU已将信号通路指向了8259A的某个IRQ接口
2. 检查IMR对应位是否为1，为1表示中断屏蔽，为0，表示中断放行，中断放行才可进入IRR寄存器。
3. 将IRR对应位置1，表示已进入等待处理队列
4. PR通过IRQ的接口号(数值越小，优先级越高)取出优先级最高的请求。
5. 此时8259A通过INT接口向CPU通知有新的中断。
6. CPU处理完手上的工作后，通过INTA回复8259A可以发送中断信息了。
7. 8259A将ISR寄存器中的对应位置1，PR寄存器中的对应位置0，这个过程是可以被高优先级的中断打断的，打断后恢复寄存器原来的状态
8. CPU再次发送信号要求发送中断向量号
9. 8259A通过起始中断向量号+IRQ接口号发送给CPU

8259A编程:

在8259A中有两组寄存器，一组是初始化命令寄存器(ICW),另一组是操作命令寄存器(OCW)，所以我们的操作分为，初始化和操作两个部分。

ICW:

ICW共有四个,ICW1~ICW4,ICW用于决定8259A的工作状态，这些设置是具有关联和依赖性的所以必须依次写入ICW1-ICW4。

ICW寄存器介绍:

ICW1(主片端口0x20,从片端口0xA0):

用于设置8259a的连接方式和中断信号的触发方式

ICW1

7	6	5	4	3	2	1	0
0	0	0	1	LTIM	ADI	SNGL	IC4

▲图 7-13 ICW1 格式

(以下1表示开启,0表示关闭)

ICW4:

是否写入ICW4, (x86必须设置写入)。

SNGL:

若SNGL为1,表示单片,若SNGL为0,表示级联。

ADI:

设置8085cpu的调用时间间隔, 我们是8086不需要设置

LTIM:

中断检测方式, =0表示边沿触发,=1表示电平触发。

其余位置为固定值。

ICW2(主片端口0x21,从片端口0xA1):

用于设置起始中断向量号(IRQ0的向量号), 其余IRQ接口顺延。我们只需要设置高5位, 剩下的3位由系统自动分配。

ICW2

7	6	5	4	3	2	1	0
T7	T6	T5	T4	T3	ID2	ID1	ID0

▲图 7-14 ICW2

ICW3(在级联模式下才会设置ICW3,主片0x21,从片0xA1):

对于主片,ICW3中置1的那一对应IRQ接口用于连接从片, 0表示连接外部设备。

对于从片, ICW3的低3位表示主片上用于连接当前从片的IRQ接口。

主片ICW3

7	6	5	4	3	2	1	0
S7	S6	S5	S4	S3	S2	S1	S0

▲图 7-15 主片 ICW3

从片ICW3

7	6	5	4	3	2	1	0
0	0	0	0	0	ID2	ID1	ID0

▲图 7-16 从片 ICW3

ICW4(主片0x21,从片0xA1);

ICW4应从高位写置低位, 低位依赖于高位的设置。

ICW4

7	6	5	4	3	2	1	0
0	0	0	SFNM	BUF	M/S	AEOI	uPM

▲图 7-17 ICW4

SFNM:

=0,表示全嵌套模式,=1,表示特殊全嵌套模式。

BUF:

=0, 表示非缓冲模式,=1,表示缓冲模式

M/S:

=1, 表示主片,=0表示从片,此设置必须工作在缓冲模式中才有效。

AEOI:

=0,表示手动结束中断, 即在中断处理函数中手动向825A的主、从片发送EOI信号。=1,表示自动结束中断。

uPM:

=0, 表示老处理器(8080,8085), =1, 表示x86处理器。

OCW寄存器介绍:

发送控制命令的寄存器

OCW1(主片0x21,从片0xA1):

用于屏蔽连接在8259A上的外设中断信号(本质是设置IMR寄存器)。

M0~M7对应8259A的IRQ0~IRQ7, =1时, 表示对应的IRQ接口中断信号被屏蔽了。

OCW1

7	6	5	4	3	2	1	0
M7	M6	M5	M4	M3	M2	M1	M0

▲图 7-18 OCW1

OCW2(主片0x20,从片0xA0):

用于设置中断结束方式和优先级模式

OCW2

7	6	5	4	3	2	1	0
R	SL	EOI	0	0	L2	L1	L0

▲图 7-19 OCW2

第一个作用发送EOI信号, 结束中断:

EOI=1且ICW4中的AE0I=0:

当SL=1,OCW的低3位指定ISR寄存器中哪一个中断被终止。

当SL=0,ISR寄存器中的优先级最高的中断被终止

第二个作用设置优先级控制方式:

R=0,固定优先级方式, 即IRQ借口好越低, 优先级越高。

R=1,循环优先级方式,

当SL=0时候,优先级由高到低是IRQ0->IRQ7,当IRQ0的中断被处理后, IRQ0的优先级变为最低, 而最高优先级赋予比之前第一级的IRQ接口(即当前荔枝中的IRQ1)。

当SL=1时候, 用低三位指定最低优先级。这里有一点需要注意,详情看下面的荔枝:

设置IRQ3为最低优先级

IR4>IR5>IR6>IR7>IR0>IR1>IR2>IR3

EOI:

发送结束中断请求,标识完成中断请求。向从片发送的时候, 需要向主片也发送

各字段图解:

表 7-2

OCW2 高位属性组合

R	SL	EOI	描述
0	0	1	普通 EOI 结束方式: 当中断处理完成后, 向 8259A 发送 EOI 命令, 8259A 会将 ISR 中当前级别最高的位置 0
0	1	1	特殊 EOI 结束方式: 当中断处理完成后, 向 8259A 发送 EOI 命令, 8259A 将 ISR 寄存器中由 L2~L0 指定的位清 0
1	0	1	普通 EOI 循环命令: 当中断处理完成后, 8259A 将 ISR 中当前优先级最高的位清 0, 并使此位的优先级变成最低, 使原来第二高的优先级成为最高优先级。其他优先级类推, 可以参照图 7-20
1	1	1	特殊 EOI 循环命令: 当中断处理完成后, 8259A 将 ISR 中由 L2~L0 指定的相应位清 0, 并使此位的优先级变成最低, 使原来第二高的优先级成为最高优先级。其他优先级类推, 可以参照图 7-20
0	0	0	清除自动 EOI 循环命令
1	0	0	设置自动 EOI 循环命令: 8259A 自动将 ISR 寄存器中当前处理的中断位清 0, 并使此位的优先级变成最低, 使原来第二高的优先级成为最高优先级。其他优先级类推, 可以参照图 7-20
1	1	0	设置优先级命令: 将 L2~L0 指定的 IR(i) 为最低优先级, IR(i+1) 为最高优先级。其他优先级类推, 可以参照图 7-20
0	1	0	无操作

OCW3(我们用不上, 不记录),page : 330

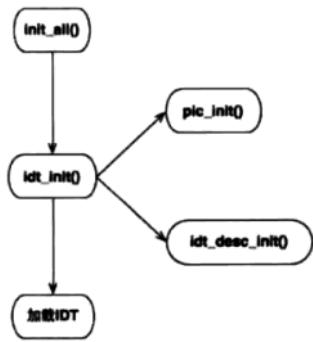
表 7-3

控制字中标识汇总

控制字	第 4 位	第 3 位
ICW1	1	/
OCW2	0	0
OCW3	0	1

编写中断处理程序

2022年5月11日 14:23



▲图 7-22 启用中断流程

inter汇编中的宏定义：

%define 定义一行宏指令

%macro 宏名称 参数个数

...

宏代码

...

%endmacro

使用%1-%n来在宏代码中使用参数

使用 宏名 参数1,参数2,参数n 来传递参数

部分中断程序代码

```
;初始化所有中断处理程序
;如果cpu自动压入错误代码，则什么都不做
#define PUSH_ZERO push 0
;如果cpu不压入错误代码，则手动压入0，保证栈结构相同
#define NO_PUSH nop
extern printf
extern put_int32
global intr_entry_table
section .data
    print_str db "interrupt occur",0xa,0
[bits 32]
intr_entry_table:
;因为编译器会组合相同属性的段，所以后面.data所有(中断方法的起始地址)会被保存在这里
;定义中断处理程式模板
//安装中断程序到中断描述符表
#include "../lib/kernel/interrupt.h"
#include "../lib/kernel/stdint.h"
#include "../lib/kernel/global.h"
#include "../lib/kernel/io.h"
#define IDT_DESC_CNT 0x21 //已定义的中断数量
extern intr_handler intr_entry_table[IDT_DESC_CNT];
/*中断门描述符结构体*/
typedef struct gate_desc
{
    uint16_t func_offset_low_word;
    uint16_t selector;
    uint8_t dcount;
    uint8_t attributes;
    uint16_t func_offset_high_word;
}Gate_desc, *PGate_desc;
```

```

[bits 32]
intr_entry_table:
;因为编译器会组合相同属性的段，所以后面.data所有(中断方法的起始地址)会被保存在这里
;定义中断处理程式模板
%macro INTERRUPT_VECTOR_FUNCTION 2
section .text
intr%entry:
    %2 ;传入的宏定义
    push %1
    call put_int32
    add esp, 4 ;平栈
    push print_str
    call printf
    add esp, 4 ;平栈
;发送EOI(结束中断请求)主片和从片都发送
    mov al, 0x20
    out 0xa0, al
    out 0x20, al
    add esp, 4 ;跳过ERROR_CODE
    iret ;从中断返回
section .data
    dd intr%entry ;与程序标号相同，此处被编译后存放对应中断向量的程式地址
%endmacro
INTERRUPT_VECTOR_FUNCTION 0x0 ,PUSH_ZERO
INTERRUPT_VECTOR_FUNCTION 0x1 ,PUSH_ZERO
INTERRUPT_VECTOR_FUNCTION 0x2 ,PUSH_ZERO
INTERRUPT_VECTOR_FUNCTION 0x3 ,PUSH_ZERO
INTERRUPT_VECTOR_FUNCTION 0x4 ,PUSH_ZERO
INTERRUPT_VECTOR_FUNCTION 0x5 ,PUSH_ZERO
INTERRUPT_VECTOR_FUNCTION 0x6 ,PUSH_ZERO
INTERRUPT_VECTOR_FUNCTION 0x7 ,PUSH_ZERO
INTERRUPT_VECTOR_FUNCTION 0x8 ,NO_PUSH
INTERRUPT_VECTOR_FUNCTION 0x9 ,PUSH_ZERO
INTERRUPT_VECTOR_FUNCTION 0xA ,NO_PUSH
INTERRUPT_VECTOR_FUNCTION 0xB ,NO_PUSH
INTERRUPT_VECTOR_FUNCTION 0xC ,NO_PUSH
INTERRUPT_VECTOR_FUNCTION 0xD ,NO_PUSH
INTERRUPT_VECTOR_FUNCTION 0xE ,NO_PUSH
INTERRUPT_VECTOR_FUNCTION 0xF ,PUSH_ZERO
INTERRUPT_VECTOR_FUNCTION 0x10 ,PUSH_ZERO
INTERRUPT_VECTOR_FUNCTION 0x11 ,NO_PUSH
INTERRUPT_VECTOR_FUNCTION 0x12 ,PUSH_ZERO
INTERRUPT_VECTOR_FUNCTION 0x13 ,PUSH_ZERO
INTERRUPT_VECTOR_FUNCTION 0x14 ,PUSH_ZERO
INTERRUPT_VECTOR_FUNCTION 0x15 ,PUSH_ZERO
INTERRUPT_VECTOR_FUNCTION 0x16 ,PUSH_ZERO
INTERRUPT_VECTOR_FUNCTION 0x17 ,PUSH_ZERO
INTERRUPT_VECTOR_FUNCTION 0x18 ,PUSH_ZERO
INTERRUPT_VECTOR_FUNCTION 0x19 ,PUSH_ZERO
INTERRUPT_VECTOR_FUNCTION 0x1A ,PUSH_ZERO
INTERRUPT_VECTOR_FUNCTION 0x1B ,PUSH_ZERO
INTERRUPT_VECTOR_FUNCTION 0x1C ,PUSH_ZERO
INTERRUPT_VECTOR_FUNCTION 0x1D ,PUSH_ZERO
INTERRUPT_VECTOR_FUNCTION 0x1E ,NO_PUSH
INTERRUPT_VECTOR_FUNCTION 0x1F ,PUSH_ZERO
INTERRUPT_VECTOR_FUNCTION 0x20 ,PUSH_ZERO

        uint16_t selector;
        uint8_t dcount;
        uint8_t attributes;
        uint16_t func_offset_high_word;
}Gate_desc,*PGate_desc;
//中断描述符表
static struct gate_desc idt[IDT_DESC_CNT];
/******8259A从片端口定义*****/
#define PIC_M_CTRL 0x20 //主片控制端口
#define PIC_M_DATA 0x21 //主片数据端口
#define PIC_S_CTRL 0xa0 //从片控制端口
#define PIC_S_DATA 0xa1 //从片数据端口
/******方法区*****/
/******8259A初始化*****/
static void pic_init() {
    /*
        先初始化从片ICM1-4
    */
    //初始化主片
    outb(PIC_M_CTRL, 0x11);
    outb(PIC_M_DATA, 0x20); //设置起始中断向量号0x20开始顺延至0x27
    outb(PIC_M_DATA, 0x4); //IRQ2接从片
    outb(PIC_M_DATA, 0x1); //????不是应该设置主片吗？
    //初始化从片
    outb(PIC_S_CTRL, 0x11);
    outb(PIC_S_DATA, 0x28); //设置起始中断向量号0x28开始顺延至0x2F
    outb(PIC_S_DATA, 0x2); //指示主片IRQ2连接此从片
    outb(PIC_S_DATA, 0x1);
    /*
        outb(PIC_M_DATA, 0xfe); //开启IRQ0中断,即时钟中断,为什么是时钟中断?,因为计数器通过out0数据线接到8259A的IRQ0
    */
    outb(PIC_S_DATA, 0xff);
    printf("pic_init Success\n");
}

/******IDT初始化*****/
//创建一个中断门描述符
static void make_idt_desc(PGate_desc p_gdesc,uint8_t attr,intr_handler function) {
    p_gdesc->func_offset_low_word = (uint32_t)function;
    p_gdesc->selector = SELECTOR_K_CODE;
    p_gdesc->dcount = 0;
    p_gdesc->attributes = attr;
    p_gdesc->func_offset_high_word = ((uint32_t)function&0xffff0000)>>16;
}
//初始化已定义的中断描述符表
static void idt_init() {
    int i;
    for (i = 0; i < IDT_DESC_CNT; i++) {
        make_idt_desc(&idt[i], IDT_ATTRIBUTE, intr_entry_table[i]);
        //待补充
    }
    printf("IDT Init Success\n");
}
void idt_load() {
    printf("idt_load Start\n");
    idt_init();
    //初始化8259A
    pic_init();
    //加载idt
    uint64_t idt_IDTR = (sizeof(idt)-1) | ((uint64_t)(uint32_t)idt<<16);
    asm volatile("lidt %0": :m"(idt_IDTR));
    printf("idt_load End");
}

```

观察中断程序流程

2022年5月18日 16:31

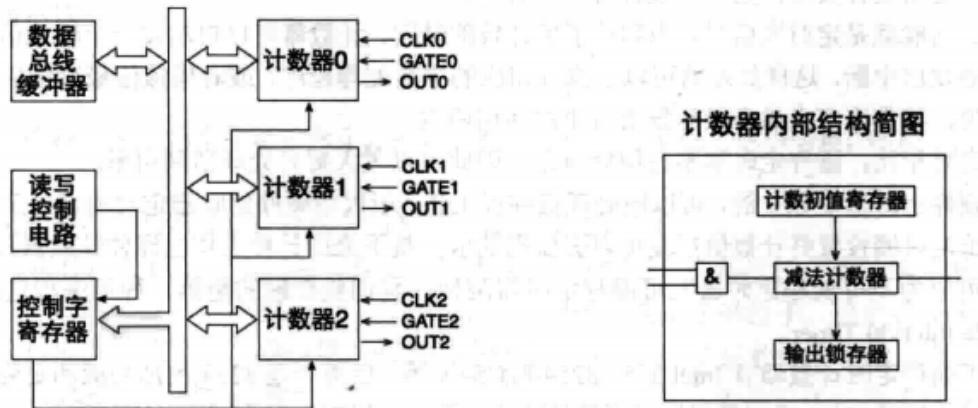
bochs调试指令:

sb x	以当前指令为准，执行x条指令后暂停
sba x	处理器加电以后执行到x条指令暂停
print-stack	显示堆栈

定时器(计时器)8253

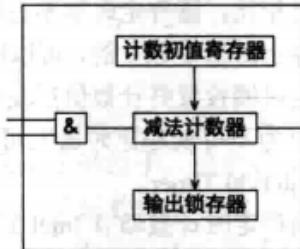
2022年5月19日 14:48

8253内部结构详情：



▲图 7-45 8253 内部结构简图

计数器内部结构简图



▲图 7-46 计数器内部结构

每个计数器都有三个引脚：

CLK:

每当此引脚收到一个时钟(脉冲)信号，减法计数器就将计数值减1.此引脚的最高频率为10MHz,8253默认设置为2MHz

GATE:

OUT:

计时器输出引脚，当定时工作结束，也就是计数值为0。

每个计数器内部都有三个16位寄存器（图7-46）：

计时器工作方式：

1. 将计数初值保存到计数初值寄存器，减法计数器载入初值
2. CLK引脚每收到一个信号，减法计数器-1，将结果保存到输出锁存器
3. 计数=0时，out引脚发送相应的信号。

三个计数器的功能：

表 7-4

8253 计数器

计数器名称	端口	作用
计数器 0	0x40	在个人计算机中，计数器 0 专用于产生实时时钟信号。它采用工作方式 3，往此计数器写入 0 时则为最大计数值 65536
计数器 1	0x41	在个人计算机中，计数器 1 专用于 DRAM 的定时刷新控制。PC/XT 规定在 2ms 内进行 128 次的刷新，PC/AT 规定在 4ms 内进行 256 次的刷新
计数器 2	0x42	在个人计算机中，计数器 2 专用于内部扬声器发出不同音调的声音，原理是给扬声器输送不同频率的方波

8253控制字：

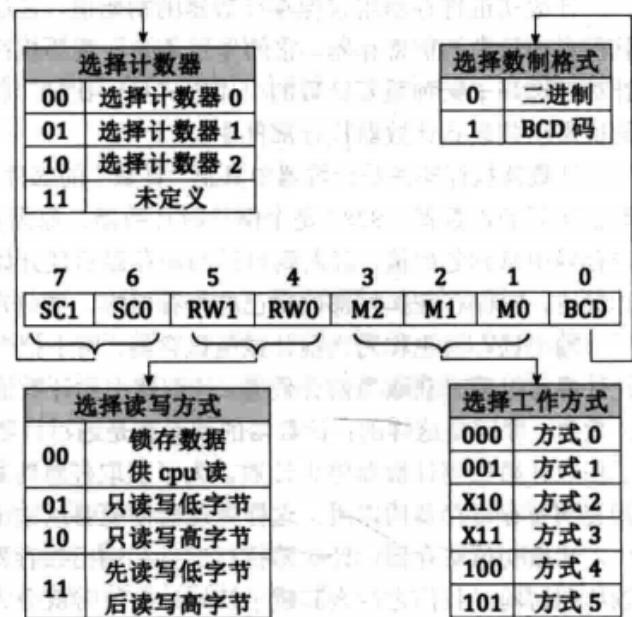
控制字寄存器：

端口:0x43

大小:8位

控制字寄存器中保存的内容称为**控制字**，通过控制字设置指定的计数器的工作方式。

控制字由8位2进制组成,图解控制字格式：



8253控制字格式

▲图 7-47 8253 控制字及部分说明

BCD码:以2进制的形式表示10进制, $0x1234=1234$

8253工作方式:

表 7-5 8253 工作方式	
工作方式	描述
方式 0	计数结束中断方式 (Interrupt on Terminal Count)
方式 1	硬件可重触发单稳方式 (Hardware Retriggerable One-Shot)
方式 2	比率发生器 (Rate Generator)
方式 3	方波发生器 (Square Wave Generator)
方式 4	软件触发选通 (Software Triggered Strobe)
方式 5	硬件触发选通 (Hardware Triggered Strobe)

表 7-6 8253 工作模式总结

工作方式	计数启动方式	终止计数方法	循环计数 (自动重装初值)	特 点
0	写入计数初值 (软件)	GATE=0	否	用来实现定时器或对外部事件计数
1	GATE 上升沿 (硬件)	—	否	用来产生单稳脉冲
2	写入计数初值 (软件)	GATE=0	是	用来实现对时钟脉冲 CLK 的 N 分频
3	写入计数初值 (软件)	GATE=0	是	用来产生连续的方波, 也用来实现对时钟脉冲 CLK 的 N 分频
4	写入计数初值 (软件)	GATE=0	否	—
5	GATE 上升沿 (硬件)	—	否	—

当控制字写入计数后, 在方式 0 中, 计数器的 OUT 端都会变成低电平, 而在另外 5 个工作方式中, 计数器的 OUT 端都会变成高电平

8253初始化步骤:

- 1.设置控制字
- 2.设置计数初值要使用计数器的端口, 计数器0-2的端口分别为0x40-0x42,8位初值直接写入,16位初值, 先写低8位, 再写高8位.

MakeFile

2022年5月20日 9:25

make简介

2022年5月20日 9:30

makefile文件中定义待编译文件中的依赖关系，供make文件使用。

make程序更具makefile中的规则，发现某个依赖文件更新后，根据makefile中的规则执相应的指令。

基本语法：

目标文件:依赖文件

[TAB命令]命令

例子:

1:2

@echo "test ok"

目标文件:

要比较的文件 (要生成的文件)

依赖文件:

目标文件的依赖文件

命令:

任何命令,@表示不打印输出这条命令

执行指令:

make -f 文件名

如果不指定文件名称，将会在当前目录中寻找下三个文件，GNUmakefile, makefile, Makefile

伪目标:

当规则中不存在目标文件和依赖文件。

.PHONY修饰伪目标名

例子:

.PHONY:clean

clean:

rm ./build/*.o

表 8-1

约定俗成的伪目标名称

伪目标名称	功 能 描 述
all	通常用来完成所有模块的编译工作，类似于 rebuild all
clean	通常用于清空编译完成的所有目标文件，一般用 rm 命令实现
dist	通常用于将打包文件后的 tar 文件再压缩成 gz 文件

第 8 章 内存管理系统

续表

伪目标名称	功 能 描 述
Install	通常将编译好的程序复制到安装目录下，此目录是在执行 configure 脚本通过--prefix 参数配置的
printf	通常用于打印已经发生改变的文件
tar	通常用于将文件打包成 tar 文件，也就是所谓的归档文件
test	通常用于测试 makefile 流程

tar	通常用于将文件打包成 tar 文件，也就是所谓的归档文件
test	通常用于测试 makefile 流程

make递归式推到目标

例子:

没有例子

自定义变量和系统变量:

变量定义格式:

变量名=字符串

注意，只支持字符串，不能使用双引号或单引号括起来

引用方式:

`$(变量名称)`

```
test2.o:test2.c
|     gcc -c -o test2.o test2.c
test1.o:test1.c
|     gcc -c -o test1.o test1.c
objfiles = test1.o test2.o
test.bin:$(objfiles)
|     gcc -o test.bin $(objfiles)
all:test.bin
|     @echo "compile done"
```

系统变量:

表 8-2 命令相关及参数相关的系统变量—命令相关的系统变量（部分）

变 量 名	描 述
AR	打包程序， 默认是 “ar”
AS	汇编语言编译器， 默认是 “as”
CC	C 语言编译器， 默认是 “cc”
CXX	C++语言编译器， 默认是 “g++”
CPP	C 预处理器， 默认是 “\$(CC) -E”， 如 gcc -E
FC	Fortran 的编译器和预处理器， Ratfor 的编译器， 默认是 “f77”
GET	从 SCCS 文件中提取文件程序， 默认是 “get”
PC	Pascal 语言编译器， 默认是 “pc”
MAKEINFO	将 texinfo 文件转换为 info 文件， 默认是 “makeinfo”
RM	删除命令， 默认是 “rm -f”
TEX	从 TeX 源文件中创建 TexDVI 文件的程序， 默认是 “tex”
WEAVE	将 Web 转换为 TeX 的程序， 默认是 “weave”
YACC	处理 C 程序的 Yacc 词法分析器， 默认是 “yacc”
YACCR	处理 Ratfor 程序的 Yacc 词法分析器， 默认是 “yacc -r”
参数相关的系统变量（部分），几乎都无默认值	
ARFLAGS	打包程序\$(AR) 的参数， 默认值为 rv
ASFLAGS	汇编语言编译器参数
CFLAGS	C 语言编译器参数
CXXFLAGS	C++编译器参数
CPPFLAGS	C 预处理器参数
FFLAGS	Fortran 语言编译器参数
LDFLAGS	链接器参数
PFLAGS	Pascal 语言编译器参数
YFLAGS	Yacc 词法分析器参数

隐含规则：

#：注释

/:换行

当.o文件不存在时， make将自动使用cc(gcc)寻找对应的.c文件，并编译

自动化变量：

\$@:表示所有目标文件

\$<:表示依赖文件的左边第一个

\$^:表示所以依赖文件

\$?:依赖文件比目标文件 更加新的文件集合

```
test2.o:test2.c
    gcc -c -o test2.o test2.c
test1.o:test1.c
    gcc -c -o test1.o test1.c
objfiles = test1.o test2.o
test.bin:$ (objfiles)
    gcc -o $@ $^
all:test.bin
    @echo "compile done"
```

模式规则(字符串匹配)：

%:匹配多个费控字符

5.24makefile:

```
BUILD_DIR = ./build
#kernel头文件夹
KERNEL_H_DIR = ./bin/lib/kernel
#kernel文件夹
KERNEL_DIR = ./bin/kernel
#device文件夹
DEVICE_DIR = ./bin/device
#kernel.bin的虚拟内存地址
ENTRY_POINT =0xc0001500
#32位文件
FILE_32BIT =-m32
#不使用c语言内建函数
FNO_BUILTIN =-fno-builtin
#nasm文件编译为elf文件
ASFLAGS =-f elf
#gcc编译
CC = gcc
#汇编编译
NASM = nasm $(ASFLAGS) -o
CFLAGS =$(FILE_32BIT) $(FNO_BUILTIN) -c -W -Wstrict-prototypes -Wmissing-prototypes
OBJS = -m elf_i386 -Ttext ENTRY_POINT -e main -o $(BUILD_DIR)/kernel.bin $(BUILD_DIR)/main.o \
$(BUILD_DIR)/print.o $(BUILD_DIR)/interrupt.o $(BUILD_DIR)/kernel.o $(BUILD_DIR)/init.o \
$(BUILD_DIR)/timer.o
# C编译代码 #
$(BUILD_DIR)/main.o:./bin/kernel/main.c
    $(CC) $(CFLAGS) -o $@ $<
$(BUILD_DIR)/init.o:./bin/kernel/init.c
    $(CC) $(CFLAGS) -o $@ $<
$(BUILD_DIR)/interrupt.o:./bin/kernel/interrupt.c
    $(CC) $(CFLAGS) -o $@ $<
$(BUILD_DIR)/timer.o:./bin/device/timer.c
    $(CC) $(CFLAGS) -o $@ $<
$(BUILD_DIR)/debug.o:./bin/kernel/debug.c
    $(CC) $(CFLAGS) -o $@ $<
# 汇编编译代码 #
$(BUILD_DIR)/print.o:./bin/lib/kernel/print.S
    $(NASM) $@ $<
$(BUILD_DIR)/kernel.o:./bin/kernel/kernel.s
    $(NASM) $@ $<
compile:$(BUILD_DIR)/main.o $(BUILD_DIR)/init.o $(BUILD_DIR)/interrupt.o $(BUILD_DIR)/timer.o \
$(BUILD_DIR)/print.o $(BUILD_DIR)/kernel.o \
$(BUILD_DIR)/debug.o
link:
    ld -m elf_i386 -Ttext $(ENTRY_POINT) -e main -o kernel.bin $(BUILD_DIR)/main.o $(BUILD_DIR)/print.o \
$(BUILD_DIR)/interrupt.o \
    $(BUILD_DIR)/kernel.o $(BUILD_DIR)/init.o $(BUILD_DIR)/timer.o $(BUILD_DIR)/debug.o
write_disk:
    dd if=./kernel.bin of=./bin/hd60M.img bs=512 count=200 seek=9 conv=notrunc
clear:
    @cd $(BUILD_DIR) && rm -f ./*
```


实现assert断言

2022年5月20日 11:39

断言：

程序运行到断言处时，某个值一定等于多少，否则出错，类似于if.else

```
#ifndef _LIB_KERNEL_DEBUG
#define _LIB_KERNEL_DEBUG
void panic_spin(char* fileName, int line, char* functionName, char*condition);
/*
... 表示接收可变参数
__VA_ARGS__ 表示所有的可变参数
*/
#define PANIC(...) panic_spin(__FILE__, __LINE__, __func__, __VA_ARGS__)
/*定义 ASSERT */
#ifndef NDEBUG //如果存在 NDEBUG标志 则使断言失效
#define ASSERT(COUNDITION) {(void) 0}
#else
#define ASSERT(COUNDITION) {\
    if(COUNDITION) {} \
    else {PANIC(#COUNDITION);} } //符号“#”让编译器将参数转换为字符串字面量.
#endif
#endif
```

```
#include "../lib/kernel/interrupt.h"
#include "../lib/kernel/stdint.h"
#include "../lib/kernel/print.h"
#include "../lib/kernel/debug.h"
/*
打印文件名称、行号、方法名称、断言条件
*/
void panic_spin(char* fileName, int line, char* functionName, char*condition){
    intr_disable();
    printf("Report a error, eror information:\n");
    printf("FileName:");
    printf(fileName);
    printf("\nLine:");
    put_int32(line);
    printf("\nFunctionName:");
    printf(functionName);
    printf("\nCondition:");
    printf(condition);
    while(1);
}
```

位图

2022年5月24日 17:43

位图(bitmap):

用于大容量的资源管理，使用1位来映射其他单位大小的资源。

位图的定义与实现:

```
#include "../lib/kernel/bitmap.h"
#include "../lib/kernel/string.h"
#include "../lib/kernel/debug.h"
typedef struct _BITMAP
{
    uint32_t bitmap_byte_len;//位图的字节长度
    uint8_t* Pbitmap;//位图的字节指针
}BitMap,*PBitMap;
void bitMapInit(PBitMap pBitMap)
{
    memset(pBitMap->Pbitmap, 0, pBitMap->bitmap_byte_len);

}

bool bitMap_scan_test(PBitMap pBitMap, uint32_t bit_offset) {
    uint32_t byte_offset = bit_offset/8;
    bit_offset = bit_offset%8;
    return (pBitMap->Pbitmap[byte_offset] & (BITMAP_MASK<<bit_offset))>>bit_offset;
}

int bitMap_scan(PBitMap pBitMap, uint32_t cnt) {
    uint32_t byte_idx = 0;
    while (pBitMap->Pbitmap[byte_idx] == 0xFF && byte_idx < pBitMap->bitmap_byte_len)//找到存在空位的字节偏移处
    {
        byte_idx++;
    }
    //找到连续未被占用的位偏移
    uint8_t bit_idx = -1;
    //记录连续空闲的位数
    uint8_t leisure_cnt = 0;
    //一个字节内的位偏移
    uint8_t bit_offset = 0;
    while (byte_idx < pBitMap->bitmap_byte_len)
    {
        bool flag = pBitMap->Pbitmap[byte_idx] & (BITMAP_MASK << bit_offset);
        //判断是否达到需要的连续空位长度
        if (leisure_cnt == cnt) {
            //找到后跳出
            bit_idx = byte_idx * 8 + bit_offset-cnt;
            break;
        }
        if (flag) leisure_cnt = 0; else leisure_cnt++;
        bit_offset++;
        if (bit_offset == 8) {
            bit_offset = 0;
            byte_idx++;
        }
    }
    return bit_idx;
}

void bitmap_set(PBitMap pBitMap, uint32_t bit_offset, uint8_t value) {
    assert(value>=0&&value<=1);
    uint32_t byte_offset = bit_offset / 8;
    bit_offset = bit_offset % 8;
    if (value == 0) {

}
```

```
    pBitMap->Pbitmap[byte_offset] &= ~(BITMAP_MASK << bit_offset);
}
else {
    pBitMap->Pbitmap[byte_offset] |= (value << bit_offset);
}
return;
}
```

内存管理系统

2022年5月25日 14:46

目标:

程序想要运行，必须被加载进内存当中，如何分配以及分配多少字节空间给用户？将是我笔记的重点，目标实现malloc和free函数。
内存池规划：

1. 使用物理内存池和虚拟内存池
2. 物理内存池又划分为内核物理内存池，和用户物理内存池。
3. 因为每个进程都有自己的4GB空间，所以每个进程（任务）都有自己的虚拟内存池。

为什么需要内存池？

因为在保护模式下存在虚拟地址和物理地址，哪些虚拟地址被占用了，哪些物理地址被占用了，需要有个管理办法。

物理内存池如何规划？

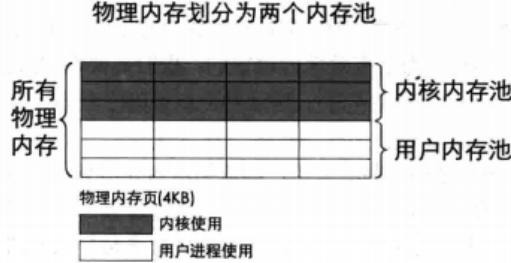
内核物理内存和用户物理内存各占一半，物理内存的申请应该有一个基本单位，我们定义为4KB(后面将实现精度更高的管理)。每次申请内存的大小必须为基本单位的倍数。当内存池空间不够时，将返回错误信息。

虚拟内存池如何规划？

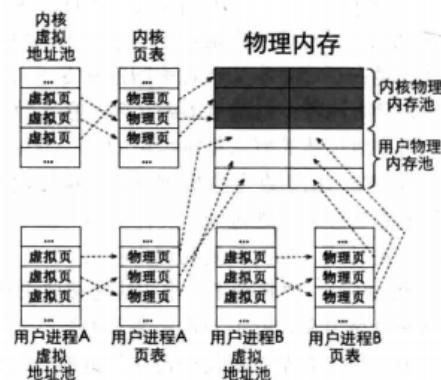
用户进程提出虚拟内存申请，由内核程序在当前进程的虚拟内存池中选定虚拟地址，在用户物理内存池中选定物理内存页，然后在该进程自己的页目录表中的页表中建立起映射关系。

内核提出虚拟内存申请，步骤相同，但是在内核物理内存池中选定物理内存地址，然后再建立映射。

虚拟地址池和物理地址池后，它们的关系如图 8-19 所示。



▲图 8-18 内存池示意



▲图 8-19 虚拟地址池与物理地址池

复习一下虚拟地址转换物理地址：

- (1) 高 10 位是页目录项 pde 的索引，用于在页目录表中定位 pde，细节是处理器获取高 10 位后自动将其乘以 4，再加上页目录表的物理地址，这样便得到了 pde 索引对应的 pde 所在的物理地址，然后自动在该物理地址中，即该 pde 中，获取保存的页表物理地址（为了严谨，说的都有点拗口了）。
- (2) 中间 10 位是页表项 pte 的索引，用于在页表中定位 pte。细节是处理器获取中间 10 位后自动将其乘以 4，再加上第一步中得到的页表的物理地址，这样便得到了 pte 索引对应的 pte 所在的物理地址，然后自动在该物理地址（该 pte）中获取保存的普通物理页的物理地址。
- (3) 低 12 位是物理页内的偏移量，页大小是 4KB，12 位可寻址的范围正好是 4KB，因此处理器便直接把低 12 位作为第二步中获取的物理页的偏移量，无需乘以 4。用物理页的物理地址加上这低 12 位的便是这 32 位虚拟地址最终落向的物理地址。

线程

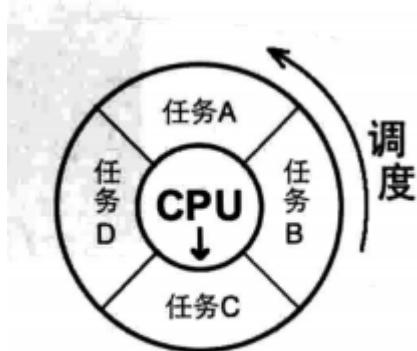
2022年6月1日 10:51

实现内核线程

2022年6月1日 10:51

执行流:

当计算机只有1个CPU时，计算机在同一时刻只能处理一个任务，使用伪并行调度可避免长时间处理单一任务。



▲图 9-1 伪并行调度

任务轮转的工作由任务调度器来完成。

任务调度器:

操作系统中用于把任务轮流调度上处理器运行的一个软件模块，它是操作系统的一部分。调度器在内核中维护一个任务表(也称进程表、线程表或调度表)，然后按照一定的算法，从任务表中选择一个任务，将任务放到处理器上运行，当任务运行的时间片到期后，再从任务表中找到另外一个任务放到处理器上运行。

任何代码块，无论大小都可以成为执行流，只要在它运行的时候，准备好所依赖的上下文环境(寄存器映像、栈、内存)

线程和普通函数的区别:

线程是一套机制，可以为一般的代码块所依赖的上下文环境，让代码块具有独立性，使之成为一个调度单元，能让CPU单独执行。处理器并不是在执行进程的时候捎带执行的线程，而是单独、专门执行了线程函数。

线程与进程的区别:

每个进程都有自己的地址空间，线程只具备独立的寄存器资源和独立的栈空间，线程必须依附于进程。

强调:

只有线程才具备能动性，它才是处理器的执行单元，因此它是调度器眼中的调度单位。进程只是个资源整合体。它将进程中所有线程运行时用到的资源收集在一起，供进程中的所有线程使用，真正上处理运行的**都是线程**。

线程状态:

需要等待外界条件达成的状态:阻塞态

正在处理器上运行的状态:运行态

外界条件成立，随时可以运行：就绪态

提问:

操作系统去哪里找到任务?

任务所需要的环境去哪里找到?

任务执行的状态?

任务什么时候被换下cpu?

答案:

使用一个结构体(也就是下面将要提到的pcb)保存进程的所有信息, 将结构体放置在一个表中, 将表的首地址赋值给固定的寄存器, 这样子操作系统就可以设置执行流的方向。

PCB:

Process control block ,进程的身份证, 用于记录进程相关信息

多线程调度

2022年6月1日 10:51

因为没有同步锁产生的错误:

```
int k = 100;
while(i<0x900){
    int c = 20000;
    while(c--);
    i++;
}
printf("main i:");
put_int32(i);
while(1);

void k_threadTest(void* number){
    int k = 100;
    while(i<0x900){
        int c = 20000;
        while(c--);
        i++;
    }
    printf("thread1 i:");
    put_int32(i);
    while(1);
}

void k_threadTest1(void* number){
    int k = 100;
    while(i<0x900){
        int c = 20000;
        while(c--);
```



```
mem_pool_init start
kernel_pool_start:0x200000
kernel_pool.pool_size:0x7F00
kernel_pool.pool_bitmap.Pbitmap:0xC009A000
kernel_pool.pool_bitmap.bitmap_byte_len:0xFE0
user_pool_start:0x8100000
user_pool.pool_size:0x7F00
user_pool.pool_bitmap.Pbitmap:0xC009AFE0
user_pool.pool_bitmap.bitmap_byte_len:0xFE0
mem_init done

*****page_table_add*****
vaddr:0xC0100000
page_phyaddr:0x200000
pdeptr:0xFFFFFC00
pteptr:0xFFFF00400
*****page_table_add*****
vaddr:0xC0101000
page_phyaddr:0x201000
pdeptr:0xFFFFFC00
pteptr:0xFFFF00404
*****page_table_add*****
thread2 i:0x900thread1 i:0x901main i:0x902
IPS: 61,788M ] A: NUM CAPS SCRL HD:0-M
```

输入输出系统

2022年5月20日 9:26

同步机制-锁

2022年6月13日 16:48

术语介绍:

公共资源:

公共内存、公共文件、公共硬件等，被所有任务共享的一套资源

临界区:

程序要想使用某个资源，必然通过一些指令去访问资源。这些指令就是临界区。

互斥:

某一时刻公共资源只能被一个任务独享，不允许多个任务同时出现在自己的临界区中。

竞争条件:

多个任务以非互斥的方式同时进入凌杰去，大家对公共资源的访问是以竞争的方式并行进行的。

总结:

为了避免产生竞争条件条件，必须保证任意时刻只能有一个任务处于临界区。因此需要保证各线程自己临界区中的所有代码都是原子操作，要么一条都不执行，要么一气呵成。

信号量:

我们的锁是用信号量来实现的，信号量就是个计数器，用来记录锁累计信号的数量，我们将实现一个二元信号量。

操作:

up(释放锁):

将信号量增加1.

唤醒此信号量上等待的线程。

down(获取锁):

判断信号量是否大于0,

大于，则将其减1.

小于，则阻塞当前线程，在此信号量上等待。

由于获取和释放信号量的操作也是全局性的，所以需要保证其原子性。

如何实现线程的唤醒和阻塞?

thread_block(阻塞):

线程的阻塞是由线程自己决定的，将线程的从就绪列表中除去，这样调度器将不会再执行此线程。

thread_unblock(唤醒):

被阻塞的线程是无法自己唤醒的，所以需要在up操作时，唤醒等待在此信号量上的线程，也就是更改其线程状态为就绪状态。

用锁实现终端输出

2022年6月13日 16:49



console



sync



console



sync

从键盘获取输入

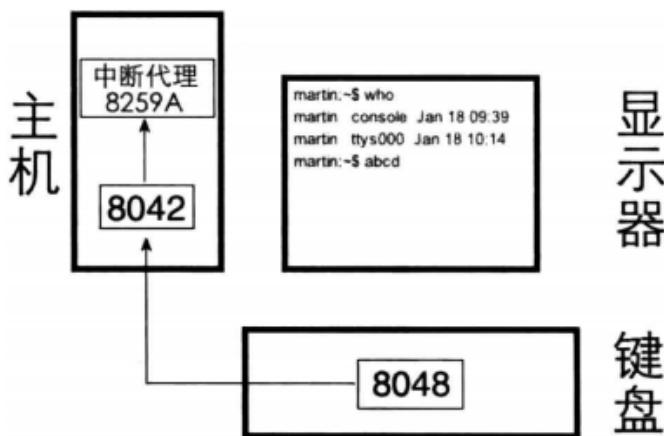
2022年6月13日 16:49

键盘:

键盘是个独立的设备，在它内部有个叫做键盘编码器的芯片，通常是Intel 8048，当键盘上发生按键操作，它就向**键盘控制器**报告哪个键被按下，按键是否弹起。

键盘控制器:

键盘控制器位于主板之上，通常为8042，作用为接收来自键盘编码器的按键信息，将其解码后保存，然后向中断代理发中断，之后处理器执行相应的中断处理程序读入8042处理保存过的按键信息。



键盘扫描码:

8048和8042约定的一张编码表，记录了所有按键按下和松开的两种状态所对应的数值。

通码(makecode):

按键被按下时的编码，按键上的触点接通了内部电路。

断码(breakcode):

按键松开弹起时产生的编码。

键盘扫描码:

扫描码共有三套XT键盘,AT键盘,IBM ps/2。如今常用的键盘扫描码都是第二套扫描码,但是8042新盘会将所有类型的扫描码转换为第一套扫描码。

第一套键盘扫描码:

表 10-1

第一套键盘扫描码

键	通码	断码	键	通码	断码
以下是主键盘区及功能区					
<esc>	01	81	<caps lock>	3a	ba
F1	3b	bb	a	1e	9e

F2	3c	bc	s	1f	9f
F3	3d	bd	d	20	a0
F4	3e	be	f	21	a1
F5	3f	bf	g	22	a2
F6	40	c0	h	23	a3
F7	41	c1	j	24	a4
F8	42	c2	k	25	a5
F9	43	c3	l	26	a6
F10	44	c4	;	27	a7
F11	57	d7	"	28	a8
F12	58	d8	<enter>	1c	9c
~	29	a9	<L-Shift>	2a	aa
!1	02	82	z	2c	ac
@2	03	83	x	2d	ad
#3	04	84	c	2e	ae
\$4	05	85	v	2f	af
%5	06	86	b	30	b0
^6	07	87	n	31	b1
&7	08	88	m	32	b2
*8	09	89	<,	33	b3
(9	0a	8a	>,	34	b4
)0	0b	8b	?/	35	b5
-	0c	8c	<R-shift>	36	b6
+=	0d	8d	<L-ctrl>	1d	9d
<backspace>	0e	8e	<L-alt>	38	b8
<tab>	0f	8f	<space>	39	b9
q	10	90	<R-alt>	e0,38	e0,b8
w	11	91	<R-ctrl>	e0,1d	e0,9d
e	12	12			
r	13	93			
t	14	94			
y	15	95			
u	16	96			
i	17	97			
o	18	98			
p	19	99			
{[1a	9a			
]}	1b	9b			
\	2b	ab			

以下是附加键及小键盘区

PrintScreen SysRq	e0,2a,e0,37	e0,b7,e0,aa	NumLock	45	c5
Scroll Lock	46	c6	/	e0,35	e0,b5
Pause Break	e1 1d 45 e1 9d c5	无	*	37	b7
Insert	e0,52	e0,d2	-	4a	ca

Home	e0,47	e0,c7	7Home	47	c7
Page Up	e0,49	e0,c9	8Up	48	c8
Delete	e0,53	e0,d3	9PgUp	49	c9
End	e0,4f	e0,cf	4Left	4b	cb
Page Down	e0,51	e0,d1	5	4c	cc
←	e0,46	e0,c6	6Right	4d	cd
→	e0,4d	e0,cd	1End	4f	cf
↑	e0,48	e0,c8	2Down	50	d0
↓	e0,50	e0,d0	3PgDn	51	d1
			0Ins	52	d2
			.Del	53	d3
			+	4e	ce
			Enter	e0,1c	e0,9c



▲图 10-15 键盘扫描码示意

在第一套键盘扫描码中,通码和断码相差0x80,即一个字节的最高位为1表示断码,为0表示通码。对于两个字节的扫描码0xe0,表示expand。

在第二套键盘扫密码中, 通码通常为一字节大小, 断码为两字节(第一个字节为0xf0, 第二个字节为通码的值)。

8042介绍:

通过8042对8048进行设置和读取信息。

寄存器	端口	读/写
Output Buffer (输出缓冲区)	0x60	读
Input Buffer (输入缓冲区)	0x60	写
Status Register (状态寄存器)	0x64	读
Control Register (控制寄存器)	0x64	写

输出缓冲区:

键盘中断处理程序必须用in指令读取"输出寄存器"，否则8042无法继续相应键盘指令。

- 状态寄存器

8位宽度的寄存器，只读，反映8048和8042的内部工作状态。各位意义如下。

- (1) 位0：置1时表示输出缓冲区寄存器已满，处理器通过in指令读取后该位自动置0。
- (2) 位1：置1时表示输入缓冲区寄存器已满，8042将值读取后该位自动置0。
- (3) 位2：系统标志位，最初加电时为0，自检通过后置为1。
- (4) 位3：置1时表示输入缓冲区中的内容是命令，置0时表示输入缓冲区中的内容是普通数据。
- (5) 位4：置1时表示键盘启用，置0时表示键盘禁用。
- (6) 位5：置1时表示发送超时。
- (7) 位6：置1时表示接收超时。
- (8) 位7：来自8048的数据在奇偶校验时出错。

- 控制寄存器

8位宽度的寄存器，只写，用于写入命令控制字。每个位都可以设置一种工作方式，意义如下。

- (1) 位0：置1时表示启用键盘中断。
- (2) 位1：置1时表示启用鼠标中断。
- (3) 位2：设置状态寄存器的位2。
- (4) 位3：置1时表示状态寄存器的位4无效。
- (5) 位4：置1时表示禁止键盘。
- (6) 位5：置1时表示禁止鼠标。
- (7) 位6：将第二套键盘扫描码转换为第一套键盘扫描码。
- (8) 位7：保留位，默认为0。

代码：

已完成,8042已将扫描码转为第一套扫描码，按住不松只会触发一次中断。

编写键盘驱动

2022年6月13日 16:49

转义字符:

- 一般转移字符, \+单个字母
 - 八进制转移字符\0+三位八进制表示的ASCLL码
 - 十六进制转义字符,\x+两位十六进制表示的ASCLL码

处理扫描码：

1. 第判断是否是控制键，如shif、ctrl、caps lock、backspace
 2. 将字符按键的扫描码转换为对应的ascll码用于展示，使用二维数组来记录映射关系，通码作为数组的索引。

```
#include "kerboard.h"
#include "../lib/kernel/stdint.h"
#include "../lib/kernel/print.h"
#include "../lib/kernel/io.h"
#include "../lib/kernel/interrupt.h"
#define KBD_BUF_PROT 0x60 //8042输出缓冲区

/*定义转义字符*/
#define esc '\033'
#define backspace '\b'
#define tab '\t'
#define enter '\r'
#define delete '\177'

/*定义不可见字符*/
#define char_invisible 0
#define ctrl_l_char char_invisible
#define ctrl_r_char char_invisible
#define shift_r_char char_invisible
#define shift_l_char char_invisible
#define alt_l_char char_invisible
#define alt_r_char char_invisible
#define caps_lock_char char_invisible

/*定义控制字符的通码和断码*/ //有些功能按键只需要知道是否处于按下状态, 所以没有断码
#define shift_l_make 0x2a
#define shift_r_make 0x36
#define shift_r_break 0xb6
#define shift_l_break 0xaa
#define alt_l_make 0x38
#define alt_r_make 0xe038
#define alt_r_break 0xe0b8
#define ctrl_l_make 0x1d
#define ctrl_r_make 0xe01d
#define ctrl_r_break 0xe09d
#define caps_lock_make 0x3a
/*
定义一个静态二维数组
索引:通码
数组元素:按键ascii码和按键与shift结合后的ascii码
*/
static char keymap[] [2]={
/* 0x00 */ {0, 0},      //占位
/* 0x01 */ {esc, esc},
/* 0x02 */ {'1', '!'},
/* 0x03 */ {'2', '@'},
/* 0x04 */ {'3', '#'}
}
```

```

/* 0x05 */ {'4', '$'},
/* 0x06 */ {'5', '%'},
/* 0x07 */ {'6', '^'},
/* 0x08 */ {'7', '&'},
/* 0x09 */ {'8', '*'},
/* 0x0A */ {'9', '('},
/* 0x0B */ {'0', ')'},
/* 0x0C */ {'_-, '_'},
/* 0x0D */ {'=, '+'},
/* 0x0E */ {backspace, backspace},
/* 0x0F */ {tab, tab},
/* 0x10 */ {'q', 'Q'},
/* 0x11 */ {'w', 'W'},
/* 0x12 */ {'e', 'E'},
/* 0x13 */ {'r', 'R'},
/* 0x14 */ {'t', 'T'},
/* 0x15 */ {'y', 'Y'},
/* 0x16 */ {'u', 'U'},
/* 0x17 */ {'i', 'I'},
/* 0x18 */ {'o', 'O'},
/* 0x19 */ {'p', 'P'},
/* 0x1A */ {'[, '{'},
/* 0x1B */ {']', '}'},
/* 0x1C */ {enter, enter},
/* 0x1D */ {ctrl_l_char, ctrl_l_char},
/* 0x1E */ {'a', 'A'},
/* 0x1F */ {'s', 'S'},
/* 0x20 */ {'d', 'D'},
/* 0x21 */ {'f', 'F'},
/* 0x22 */ {'g', 'G'},
/* 0x23 */ {'h', 'H'},
/* 0x24 */ {'j', 'J'},
/* 0x25 */ {'k', 'K'},
/* 0x26 */ {'l', 'L'},
/* 0x27 */ {';', ':'},
/* 0x28 */ {'\'', '\"'},
/* 0x29 */ {'`', '~'},
/* 0x2A */ {shift_l_char, shift_l_char},
/* 0x2B */ {'\\', '|'},
/* 0x2C */ {'z', 'Z'},
/* 0x2D */ {'x', 'X'},
/* 0x2E */ {'c', 'C'},
/* 0x2F */ {'v', 'V'},
/* 0x30 */ {'b', 'B'},
/* 0x31 */ {'n', 'N'},
/* 0x32 */ {'m', 'M'},
/* 0x33 */ {'<, '<'},
/* 0x34 */ {'., '>'},
/* 0x35 */ {'/, '?'},
/* 0x36 */ {shift_r_char, shift_r_char},
/* 0x37 */ {'*', '*'},
/* 0x38 */ {alt_l_char, alt_l_char},
/* 0x39 */ {' ', ' '},
/* 0x3A */ {caps_lock_char, caps_lock_char}
};

/*定义静态变量记录控制键的状态*/
static bool ctrl_status, shift_status, alt_status, ctrl_status, caps_lock_status, ext_scancode;
/*
键盘中断处理程序
*/
static void intr_keyboard_handler(void) {
    bool break_code = false;
    bool temp_shift = shift_status;

```

```

//读取输出端口的值，否则8042不再响应键盘中断
uint16_t scanned_code = (uint16_t)inb(KBD_BUF_PROT);
/*如果scanned_code是0xe0则表示会产生多个扫描码,结束当前中断接收下一个扫描码*/
if(scanned_code==0xe0) {
    ext_scancode = true;
    return;
}
if(ext_scancode) {
    scanned_code = 0xe000|scanned_code;//组合双字节扫描码
}
break_code = ((scanned_code&0x80)!=0);//判断是否是断码
if(break_code) {
    uint16_t make_code = scanned_code&0x7f;//获取断码对应的left通码
    if(make_code==shift_l_make) {
        shift_status = false;
    }else if(make_code==alt_l_make) {
        alt_status = false;
    }else if(make_code==ctrl_l_make) {
        ctrl_status = false;
    }
    return;
}
//判断是否是控制码
if(scanned_code==shift_l_make||scanned_code==shift_r_make) {
    shift_status = true;
    return;
}else if(scanned_code==alt_l_make||scanned_code==alt_r_make) {
    alt_status = true;
    return;
}else if(scanned_code==ctrl_l_make||scanned_code==ctrl_r_make) {
    ctrl_status = true;
    return;
}else if(scanned_code==caps_lock_make) {
    caps_lock_status = ~caps_lock_status;
    return;
}
//组合控制码
if(caps_lock_status==true) {
    temp_shift=true;
}
if(shift_status==true&&caps_lock_status==true) {
    temp_shift=false;
}
if(scanned_code>0&&scanned_code<0x3b) {//判断是否在我们已定义的扫描码数组之内
    char cur_char = 0;
    if ((scanned_code>=0x1&&scanned_code<=0xd) || scanned_code==0x1a || scanned_code==0x1b || //这些特殊扫描码将
忽略caps Lock
    scanned_code==0x27 || scanned_code==0x28 || scanned_code==0x29 || scanned_code==0x2b || scanned_code>=0x33)
    {
        cur_char = keymap[scanned_code][shift_status];
    }else{
        cur_char = keymap[scanned_code][temp_shift];//取对应字符
    }
    put_char(cur_char);
}else{
    printf("\nunknow make code\n");
}
return;
}
/*

```

```
键盘设备初始化
*/
void keyboard_init(void) {
    //注册中断例程
    register_handler(0x21, intr_keyboard_handler);
}
```

环形输入缓冲区

2022年6月22日 13:48

总结:

对于有限大小的公共缓冲区，如何同步生产者与消费者的运行，以达到对共享缓冲区的互斥访问，并且保证生产者不会过度生产，消费者不会过度消费。

bug修改记录:

```
bool ioq_empty(ioqueue* ioq) {
    return ioq->head==ioq->tail; // "=="写做了"="
}

static void ioq_wait(task_struct** wait_task) { //添加缓冲区等待线程时，这段代码要关中断，原因如注释中所说。
    enum intr_status old_state = intr_disable();
    *wait_task = get_running_thread_pcb(); //记录被休眠的线程(必须在前面，否则设置线程阻塞后没法记录，就没法解锁)，因为这个函数已经被我上锁了，所以不怕被打断了，被坑了半小时。
    ioqueue* tempIoq = (ioqueue*)struct_entry(wait_task, ioqueue, consumer);
    printf("now tempIoq->consumer:");
    put_int32(tempIoq->consumer);
    printf("\n");
    thread_block(TASK_BLOCKED);
    intr_set_status(old_state);
}
/*
唤醒等待在此缓冲区上的消费者或生产者线程, wait_task
*/
static void ioq_wakeup(task_struct** wakeup_task) { //thread_unlock中会判断待解锁线程的状态，如果不上锁执行，则会触发assert
    enum intr_status old_state = intr_disable();
    thread_unlock(*wakeup_task);
    *wakeup_task=NULL; //删除被休眠的线程记录
    intr_set_status(old_state);
}

/*
功能:解除线程pthread阻塞
*/
void thread_unlock(task_struct* pthread) {
    enum intr_status old_status = intr_disable(); //关中断必须位于方法的最前方以免被线程先换，但是这里有个疑问，就是执行push ebp mov ebp,esp的时候被交换走了怎么办？？？？
    assert((pthread->status==TASK_BLOCKED) || (pthread->status==TASK_HANGING) || (pthread->status==TASK_WAITING));
    assert(pthread->status!= TASK_READY);
    list_push(&thread_ready_list, &pthread->general_tag);
    pthread->status = TASK_RUNNING;
    intr_set_status(old_status);
}
```

又找了半天的bug:

其实不算bug，因为ioq_getchar是在关中断的情况下运行的，所以ioq_getchar外部的代码决定了整个线程的耗时，如果外部代

码耗时过短，则会导致第一个开始获取锁的线程一直能获取到锁，而其他线程即使在第一个线程释放锁后加入到就绪列表，也会因为第一个线程的继续运行(这时第一个线程又获取到了锁)，而无法获取到锁，陷入等待锁链表。如此多线程退化为单线程执行。

解决办法：

在外部判断 缓冲区是否为空。

在外部添加while循环增加耗时

两种方法起始都是一个原理，增加线程的耗时,避免其他线程无法得到交换。

用户进程

2022年6月24日 15:58

LDT

2022年6月27日 10:18

其实咱们在学习过程中会遇到很多类似的情况：如果一开始就不懂，之后无论怎么解释都不明白。造成这一现象可能的原因是。

- (1) 自己在这方面基础薄弱，该了解的基础知识不足。
 - (2) 或许自己只是某个知识点没搞清楚，从而导致整个知识链是混乱的，但自己并不知道自己理解错了。
 - (3) 对方站在他自己的知识层面上解释，因此对咱们来说显得太 high-level，不够通透。
 - (4) 对方自己就不是特别明白，只是对此知识熟练了，因此认同、接受了而已。

LDT(Local Descriptor Table):

LDT与GDT是相对应的,LDT是任务私有的结构.

如何找到LDT?

为LDT创建一个描述符，放置在GDT中。

LDT描述符:



LDT描述符格式

如何找到LDT的描述符?

cpu通过lldt将ldt加载到LDTR寄存器

lldt “16位通用寄存器”或“16位内存单元”

“ldt描述符在GDT中的索引”

windows和linux中都没有使用LDT

ps:后面再做详细了解

TSS

2022年6月27日 10:32

介绍:

CPU自动用此结构体变量保存任务的状态(上下文环境, 寄存器值)和自动从此结构体变量中载入任务的状态, 当加载新任务时, CPU自动将当前任务存入当前任务的TSS, 然后将新任务TSS中的数据载入到对应的寄存器。任务切换的本质就是TSS的切换。

TR寄存器:(TR中存储的是段选择子)

专门存储TSS信息的寄存器。



TSS描述符格式

B位:

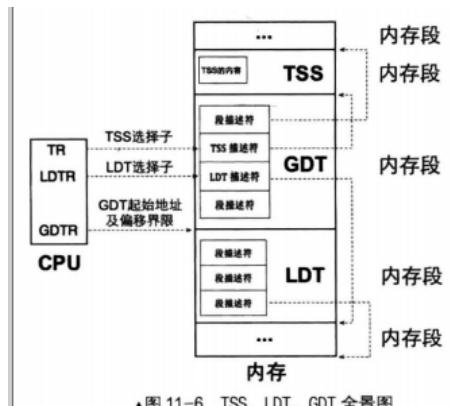
表示任务是否正在执行或存在嵌套调用已被挂起。

31	15	0
10位图在TSS中的偏移地址	(保留)	T
(保留)	ldt选择子	100
(保留)	gs	96
(保留)	fs	92
(保留)	ds	88
(保留)	ss	84
(保留)	cs	80
(保留)	es	76
	edi	72
	esi	68
	ebp	64
	esp	60
	ebx	56
	edx	52
	ecx	48
	eax	44
	eflags	40
	eip	36
	cr3(pdbr)	32
(保留)	SS2	28
	esp 2	24
(保留)	SS1	20
(保留)	esp1	16
(保留)	SS0	12
(保留)	esp 0	8
(保留)	上一个任务的 TSS 指针	4
		0

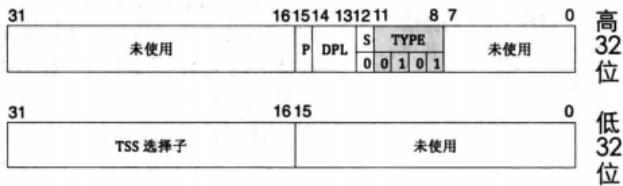
▲图 11-4 32 位 TSS 结构

| ltr "16 位通用寄存器" 或 "16 位内存单元"

TSS由用户提供, 由CPU自动维护



▲图 11-6 TSS、LDT、GDT 全景图



▲图 11-7 任务门描述符格式

iret:

iret共有两个功能

1. 从中断返回当前任务的中断前代码处
2. 从当前任务是被嵌套调用时，它会调用自己TSS中上一个任务的TSS指针的任务，也就是但回到上一个任务。

CPU如何区分是返回上一个任务继续执行还是返回中断前的状态呢？

NT位是eflags中的第14位,1bit宽度，表示任务嵌套，指当前任务是被前一个任务调用后才执行的。

TSS的字段“上一个任务的TSS指针”记录是哪个任务调用了当前任务，形成了任务嵌套关系的链表。

cpu通过这两个位置来判断返回位置。

新任务创建CPU执行流程：

1. 将新任务的NT位置1
2. 将旧任务的TSS选择子写入到新任务的“上一个任务的TSS”字段中

中断发生时通过任务门进行任务切换过程：

1. 从该任务门描述符中取出任务的TSS选择子
2. 用新任务的TSS选择子在GDT中索引TSS描述符
3. 检查TSS描述符的P位
4. 从TR寄存器中获取旧任务的TSS位置，保存旧任务的状态到旧TSS中
5. 将新任务的tss中的各种状态值加载到对应寄存器
6. 将TR寄存器指向新任务的TSS
7. 将新任务的eflags的NT位置1.
8. 将旧任务的TSS写入到新任务的TSS中的“上一个任务TSS”字段中
9. 执行新的任务

从新任务返回流程：

1. 执行到iret时,检查NT位是否为1
2. 将TSS描述符的B位置为0
3. 将当前任务的状态写入到TSS
4. 获取当前任务的上一个任务，将上一个任务TSS选择子加载到TR寄存器,恢复上一个任务的状态
5. 执行上一个任务

现代操作系统采用的任务切换方式：

必须使用TSS来完成CPU向更高特权级转移时所使用的栈地址，我们效仿linux。

Linux—TSS用法：

- a. 只是用其中的0特权和3特权。
- b. Linux为每个CPU创建一个TSS，在各个CPU上所有任务共享同一个TSS，各CPU的TR寄存器保存各CPU扇的TSS，在用ltr加载TSS后，该TR寄存器永远指向同一个TSS，在进程切换时，只需要把TSS中的SS0以及更新为新任务的内核栈的短地址以及栈指针。

定义并初始化TSS：

```
<bochs:6> info gdt
Global Descriptor Table (base=0xc0000919, limit=55):
GDT[0x0000]=??? descriptor hi=0x10000000, lo=0x10000000
GDT[0x0008]=Code segment, base=0x00000000, limit=0xffffffff, Execute-Only, Non-Conforming, Accessed, 32-bit
GDT[0x0010]=Data segment, base=0x00000000, limit=0xffffffff, Read/Write, Accessed
GDT[0x0018]=Data segment, base=0xc00b8000, limit=0x00007fff, Read/Write, Accessed
GDT[0x0020]=32-Bit TSS (Available) at 0xc0008f40, length 0x0006b
GDT[0x0028]=Code segment, base=0x00000000, limit=0xffffffff, Execute-Only, Non-Conforming, 32-bit
GDT[0x0030]=Data segment, base=0x00000000, limit=0xffffffff, Read/Write
You can list individual entries with 'info gdt [NUM]' or groups with 'info gdt [NUM] [NUM]'
<bochs:7> n
Next at t=157330968
(0) [0x0000000042dc] 0008:c00042dc (unk. ctxt): ltr ax ; 0f00d8
<bochs:8> n
Next at t=157330969
(0) [0x0000000042df] 0008:c00042df (unk. ctxt): mov dword ptr ss:[esp], 0xc00059dd ; c70424dd5900c0
<bochs:9> info gdt
Global Descriptor Table (base=0xc0000919, limit=55):
GDT[0x0000]=??? descriptor hi=0x10000000, lo=0x10000000
GDT[0x0008]=Code segment, base=0x00000000, limit=0xffffffff, Execute-Only, Non-Conforming, Accessed, 32-bit
GDT[0x0010]=Data segment, base=0x00000000, limit=0xffffffff, Read/Write, Accessed
GDT[0x0018]=Data segment, base=0xc00b8000, limit=0x00007fff, Read/Write, Accessed
GDT[0x0020]=32-Bit TSS (Busy) at 0xc0008f40, length 0x0006b
GDT[0x0028]=Code segment, base=0x00000000, limit=0xffffffff, Execute-Only, Non-Conforming, 32-bit
GDT[0x0030]=Data segment, base=0x00000000, limit=0xffffffff, Read/Write
You can list individual entries with 'info gdt [NUM]' or groups with 'info gdt [NUM] [NUM]'
<bochs:10> █
```

实现用户进程

2022年6月28日 10:51

进入特权级3:

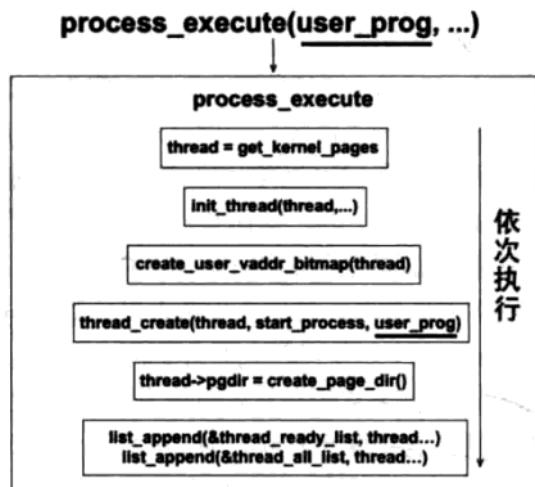
一般情况下CPU不允许从高特权级到低特权级，除了中断和调用门返回的情况。

使用中断进入特权级3:

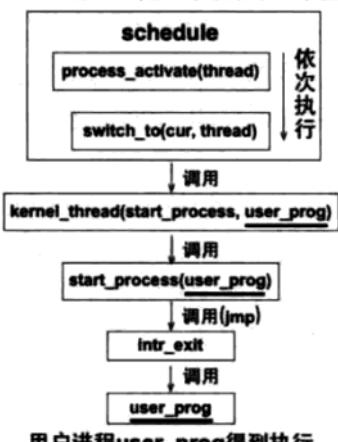
通过修改即将返回的中断栈空间中的值,使用iret进入特权级3

需要注意的点:

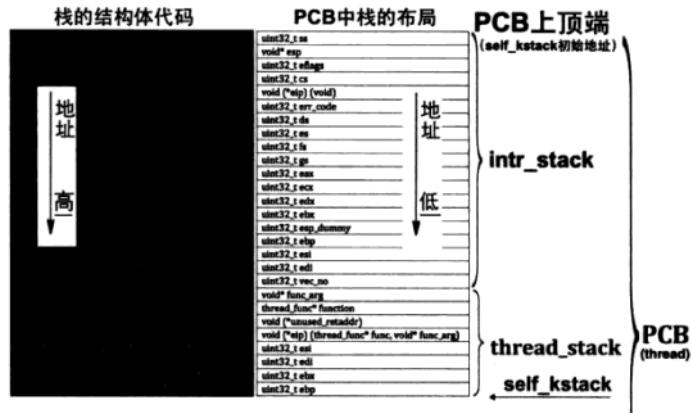
- a. 中断返回的函数位于kernel.s的intr_exit
- b. 通过栈布局用户进程需要的寄存器和运行环境
- C. cs.Cpl = 3
- d. 其他段选择子的rpl = 3
- e. eflags.if = 1,iopl = 0



时钟中断发生，调度器schedule
从就绪队列thread_ready_list中获取下一个任务thread



用户进程user_prog得到执行





低地址

为实现之后的进程堆管理，这里先预先学习一下.bss段的位置：

```
Section to Segment mapping:
Segment Sections...
00
01 .interp
02 .interp .note.gnu.property .note.gnu.build-id .note.ABI-tag .gnu.hash .dynsym .dynstr
a.plt
03 .init .plt .plt.got .plt.sec .text .fini
04 .rodata .eh_frame_hdr .eh_frame
05 .init_array .fini_array .data.rel.ro .dynamic .got .data .bss
06 .dynamic
07 .note.gnu.property
08 .note.gnu.build-id .note.ABI-tag
09 .note.gnu.property
10 .eh_frame_hdr
11
```

可以看到.bss和.data两个节被合并到序号05的段中，

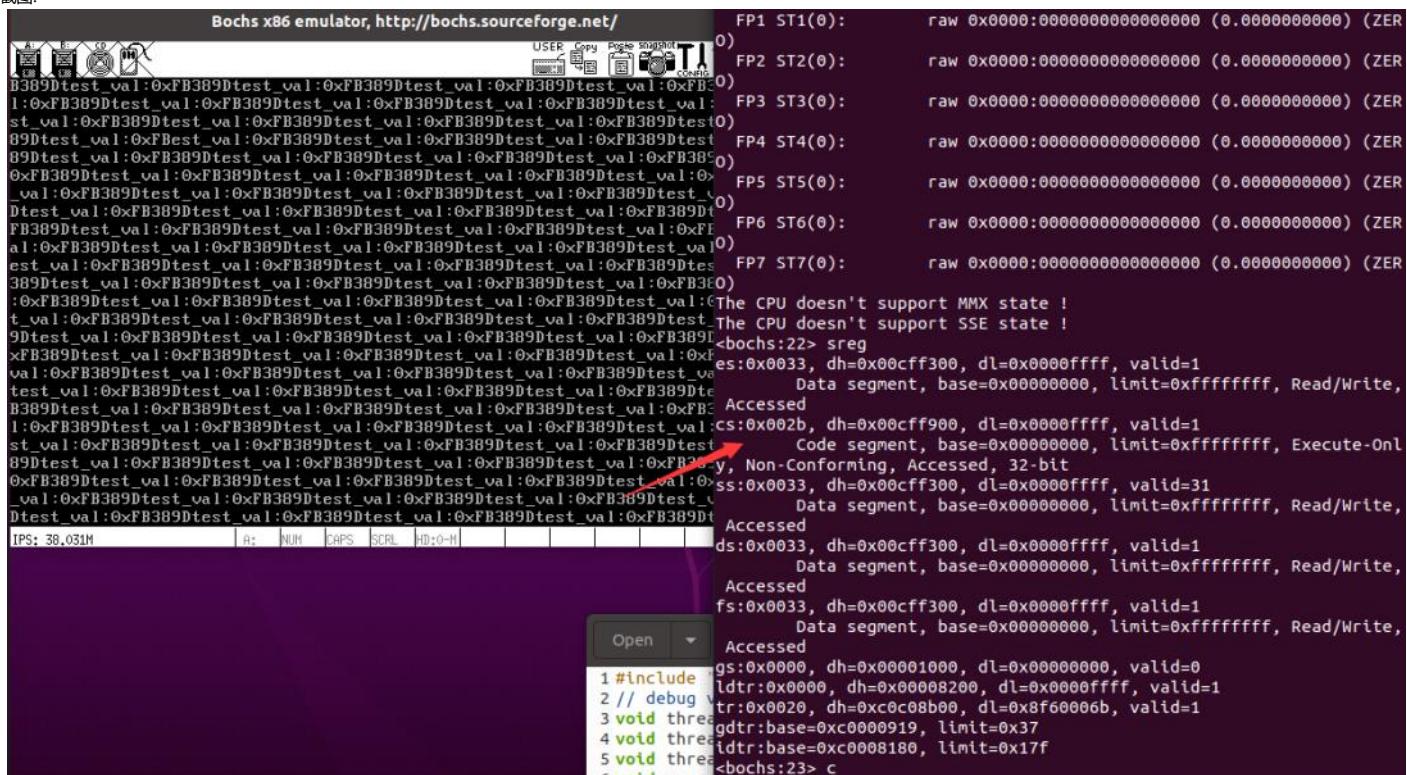
Program Headers:					
Type	Offset	VirtAddr	PhysAddr	Flags	Align
PHDR	0x0000000000000040	0x0000000000000040	0x0000000000000040		
	0x0000000000002d8	0x0000000000002d8	R 0x8		
INTERP	0x000000000000318	0x000000000000318	0x00000000000000318		
	0x000000000000001c	0x000000000000001c	R 0x1		
[Requesting program interpreter]	/lib64/ld-linux-x86-64.so.2				
LOAD	0x0000000000000000	0x0000000000000000	0x0000000000000000		
	0x0000000000006a90	0x0000000000006a90	R 0x1000		
LOAD	0x0000000000007000	0x0000000000007000	0x0000000000007000		
	0x000000000000331a1	0x000000000000331a1	R E 0x1000		
LOAD	0x0000000000003b000	0x0000000000003b000	0x0000000000003b000		
	0x000000000000fb18	0x000000000000fb18	R 0x1000		
LOAD	0x000000000004b110	0x000000000004c110	0x000000000004c110		
	0x0000000000028b8	0x000000000003368	RW 0x1000		

我们可以看到序号5的段为rw属性，即数据段中，所以我们只需要知道数据段的起始地址和结束地址，在结束地址之上就是我们堆的起始地址。

实现思路：

- 首先就是初始化进程在中断处的内核栈情况
- 为三环进程申请一个3环栈空间，初始化三环内存虚拟地址位图结构体
- 填充中断处的内核栈情况，设置cs和eip指向3环选择子和进程入口点
- 添加进程到就绪链表和全局链表
- 通过中断返回到3环进程空间

实现截图：



cs指向了下标为5的段选择子，是我们在tss.c设计的3环选择子无疑。

系统调用

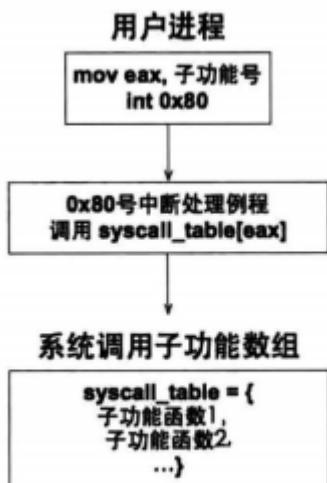
2022年6月30日 13:55

实现系统调用

2022年6月30日 13:55

系统调用实现思路:

1. 利用中断门实现系统调用,0x80中断号作为系统调用入口
2. 在IDT中安装0x80号中断处理程序
3. 创建系统调用子功能表，通过eax中存放的子功能号，调用不同的功能
4. 使用宏实现用户进程的 _syscall, eax存放子功能号, ebx, ecx, edx，分别保存第1-3个参数。



▲图 12-5 系统调用实现流程

宏定义大括号用法:

```
#include "syscall.h"  
  
#define _syscall0(NUMBER) ({  
    uint32_t retval;  
    asm volatile ("int $0x80" : "=a" (retval) : "a" (NUMBER) : "memory");  
    retval; //大括号中的最后一个语句的值会被当初大括号代码块的返回值,要添加";"号  
})
```

实现用户进程 Printf

2022年7月4日 13:47

可变参数的原理：

printf(format,arg1,arg2,...)

通过判断format字符串中的字符“%”的数量，在栈中寻找参数。

gcc中实现可变参数的内建函数：

下面是3个宏的说明。

(1) va_start(ap,v), 参数ap是用于指向可变参数的指针变量，参数v是支持可变参数的函数的第一个参数（如对于printf来说，参数v就是字符串format）。此宏的功能是使指针ap指向v的地址，它的调用必须先于其他两个宏，相当于初始化ap指针的作用。

(2) va_arg(ap,t), 参数ap是用于指向可变参数的指针变量，参数t是可变参数的类型，此宏的功能是使指针ap指向栈中下一个参数的地址并返回其值。

(3) va_end(ap), 将指向可变参数的变量ap置为null，也就是清空指针变量ap。

好啦，有关可变参数的原理就介绍到这，如果您此时尚未完全明白也没关系，下节咱们会实现以上这三个宏，在代码中实践会是理解它们的最佳方式。

实现系统调用write：

printf -> vsprintf -> write

3步走：

1. 3环定义
2. 0环实现
3. 添加到系统调用表

实现vsprint：

1. 查找格式化字符串中的“%”
2. 使用可变参数替换“%”
3. 将新的字符串交给write输出
4. 返回字符串长度

完成：

```
syscall_init success
kernel main thread pid:0x1
user process peb:0xC0100000
thread_pcb->pid:0x2
user process peb:0xC0103000
thread_pcb->pid:0x3
thread_pcb->pid:0x4
The execution process is about to start
*****
user_process_1 PID 2
numberd:999
,numberd:-6512
numberx:@
.string:hello
.char:#
success
*****
The execution process is about to start
user_process_1 PID 3
thread1 start
thread1 PCB:
0xC0106000
test_pid1:0x2
test_pid2:0x3
```


完善堆内存管理

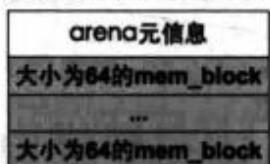
2022年7月6日 9:58

我们之前的内存分配都是以0x1000字节为单位，如何管理一页内小字节的内存单元？

arena(舞台)占一页大小：

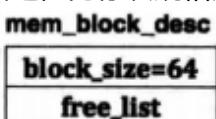
用于满足小内存的申请。

内存块规格为64字节的arena



▲图 12-13 arena 简图

用于描述，内存块规格大小相同的多个arena：

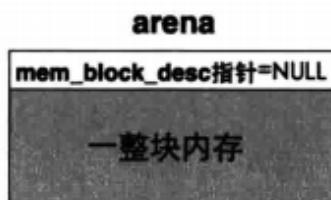


使用内存块元信息(arena元信息)指向内存块描述符(mem_block_desc),这样就可以获取自身内存块的规模

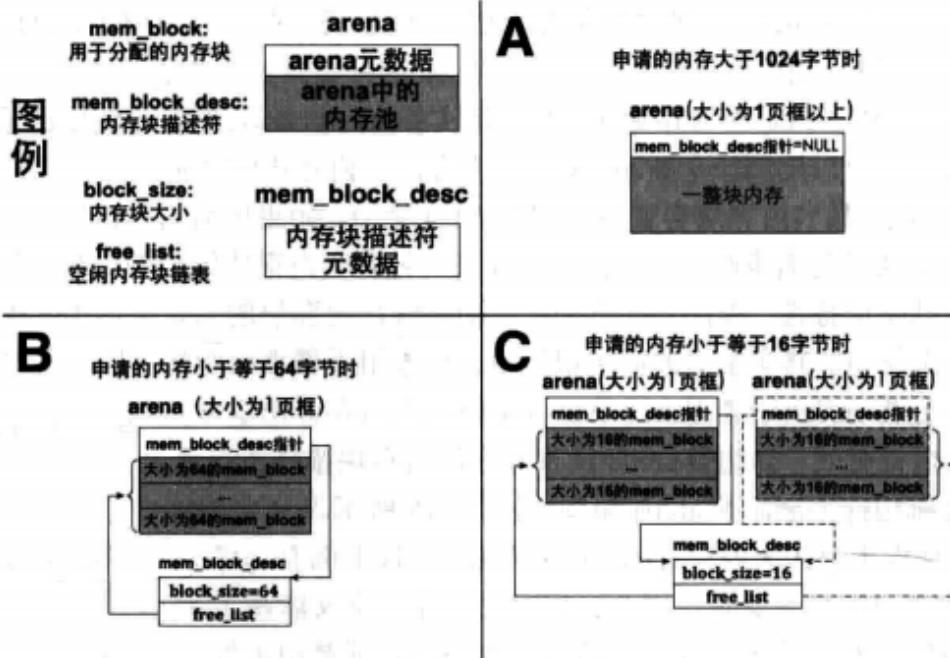
新的内存管理模式：

所有的内存申请都需要经过内存块描述符，根据需要划分的内存大小，选择不同的内存块描述符，再通过free_list链表找到空闲的内存。

大块内存(大于1024字节的内存申请)：



▲图 12-15 大内存 arena 简图



▲图 12-16 arena 与 mem_block_desc 的逻辑关系

释放内存：

申请内存的步骤：

```
/*
功能：分配pg_cnt个页空间
返回值：成功返回虚拟地址起始地址，失败返回NULL
*/
static void* malloc_page(enum pool_flags pf, uint32_t pg_cnt) {
    void* vaddr_start = vaddr_get(pf, pg_cnt);
    if(vaddr_start==NULL) { //申请虚拟内存失败
        return NULL;
    }
    pool* pMem_pool = pf&PF_KERNEL?&kernel_pool:&user_pool;
    uint32_t _vaddr = (uint32_t)vaddr_start;
    //因为物理地址可以不连续所以需要以页位单位进行申请和绑定
    while (pg_cnt--)
    {
        void* phyaddr_start = palloc(pMem_pool);
        if(phyaddr_start==NULL) { //申请物理内存失败
            //回滚虚拟内存池位图
            //待补充
            return NULL;
        }
        page_table_add((void*)_vaddr, phyaddr_start); //建立映射
        _vaddr+=PG_SIZE;
    }
    return vaddr_start;
}
```

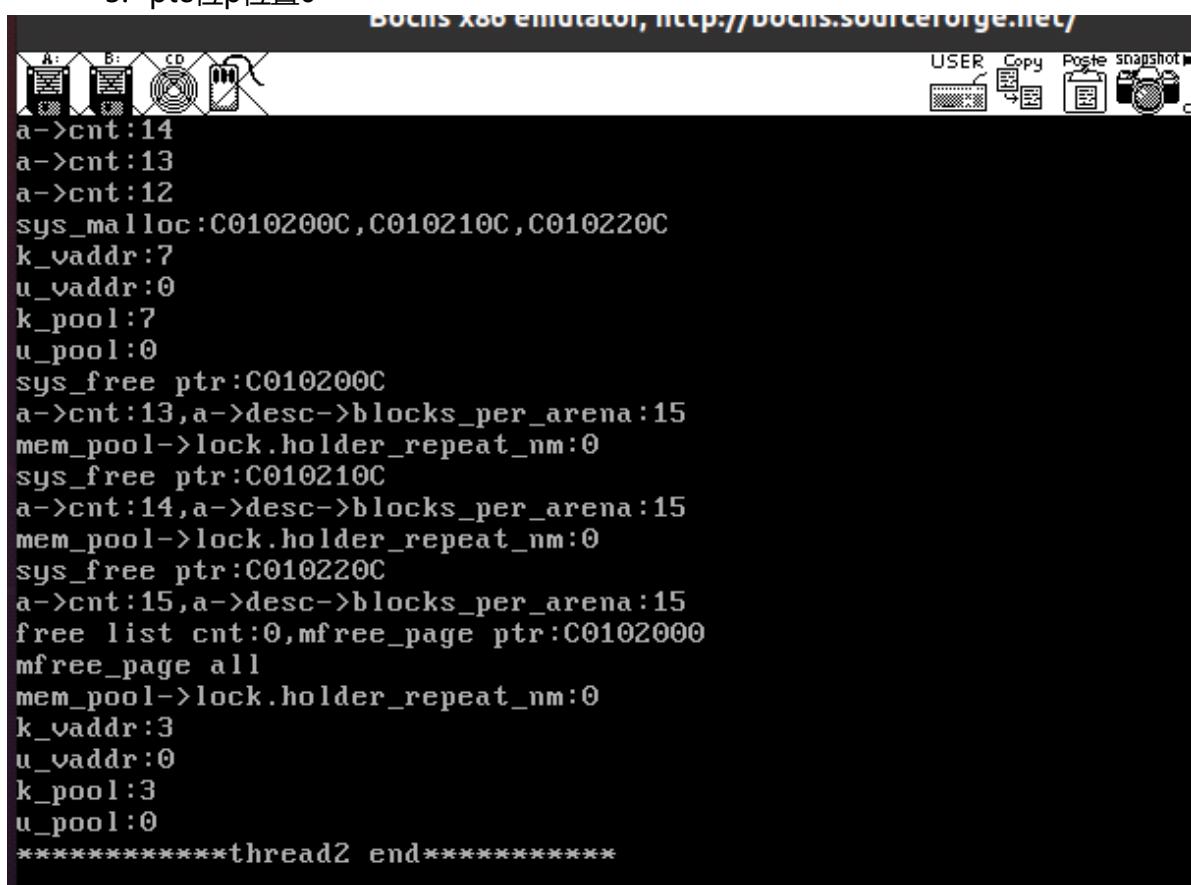
1. vaddr_get 设置对应虚拟地址内存池的虚拟地址位图，获取虚拟地址
2. palloc 设置对应物理地址内存池的物理地址位图，获得物理地址

3. page_table_add 建立映射,使pte指向实际的物理地址

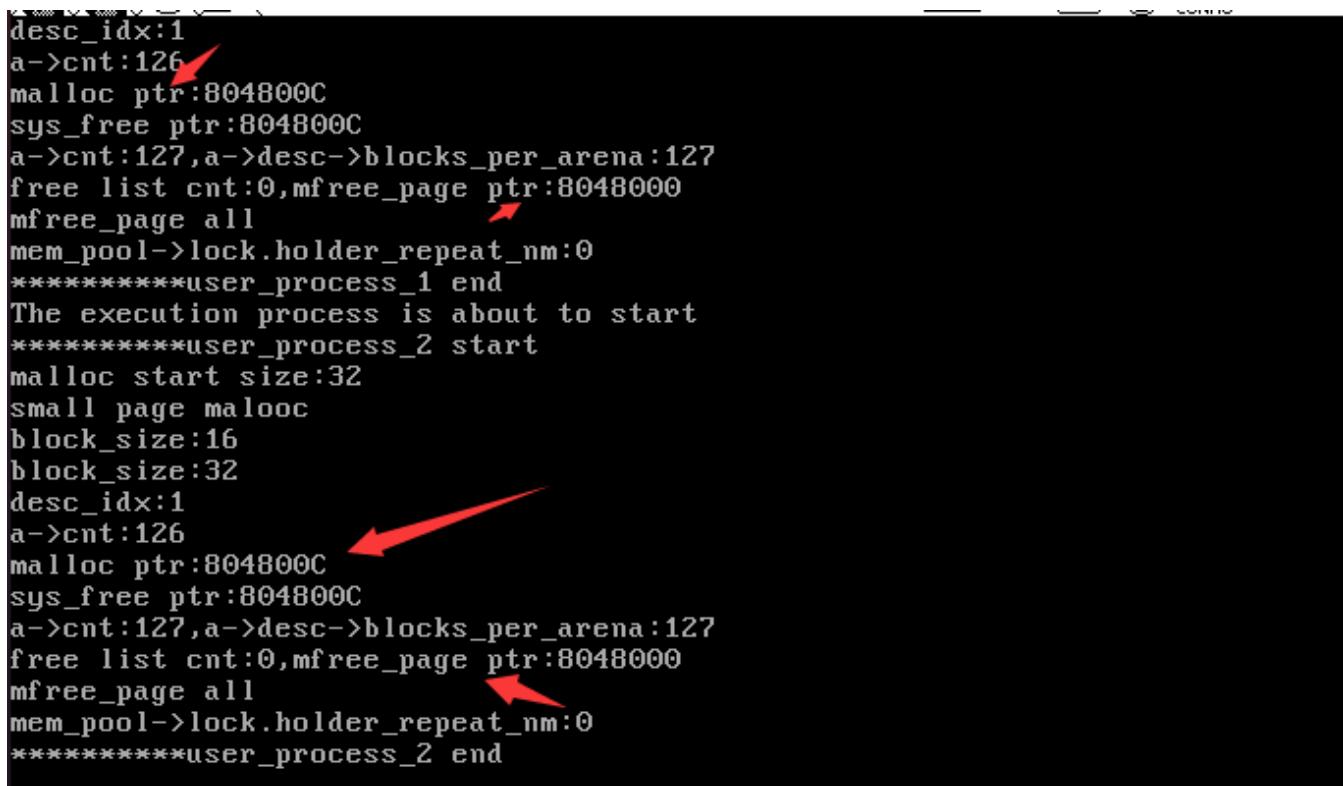
所以我们释放内存的步骤应为:

1. 还原虚拟内存池位图
2. 还原物理内存池位图
3. pte位p位置0

DOSBox x86 emulator, http://dosbox.sourceforge.net/



```
a->cnt:14
a->cnt:13
a->cnt:12
sys_malloc:C010200C,C010210C,C010220C
k_vaddr:7
u_vaddr:0
k_pool:7
u_pool:0
sys_free ptr:C010200C
a->cnt:13,a->desc->blocks_per_arena:15
mem_pool->lock.holder_repeat_nm:0
sys_free ptr:C010210C
a->cnt:14,a->desc->blocks_per_arena:15
mem_pool->lock.holder_repeat_nm:0
sys_free ptr:C010220C
a->cnt:15,a->desc->blocks_per_arena:15
free list cnt:0,mfree_page ptr:C0102000
mfree_page all
mem_pool->lock.holder_repeat_nm:0
k_vaddr:3
u_vaddr:0
k_pool:3
u_pool:0
*****thread2 end*****
```



```
desc_idx:1
a->cnt:126
malloc_ptr:804800C
sys_free_ptr:804800C
a->cnt:127,a->desc->blocks_per_arena:127
free list cnt:0,mfree_page_ptr:8048000
mfree_page all
mem_pool->lock.holder_repeat_nm:0
*****user_process_1 end
The execution process is about to start
*****user_process_2 start
malloc start size:32
small_page_malooc
block_size:16
block_size:32
desc_idx:1
a->cnt:126
malloc_ptr:804800C
sys_free_ptr:804800C
a->cnt:127,a->desc->blocks_per_arena:127
free list cnt:0,mfree_page_ptr:8048000
mfree_page all
mem_pool->lock.holder_repeat_nm:0
*****user_process_2 end
```


编写硬盘驱动程序

2022年7月8日 16:21

硬盘及分区表

2022年7月8日 16:21

盘片:

类似光盘中的一个圆盘，上面布满了磁性介质。

扇区:

扇区是硬盘读写的基本单位，它在磁道上均匀分布，与磁头和磁道不同(0, lba也从0开始寻址)，扇区从1开始编号。扇区的大小字节数 = 256*N, N为自然数。通常取N为2，因此扇区大小为512字节。

磁道:

盘片上一个个同心圆就是磁道，它是扇区的载体，每一个磁道由外向里从0开始编号。

磁头:

磁头用于读取盘片上的数据，盘片分为上下两个盘面。一个盘片也就拥有两个磁头，而一个硬盘不止有一个盘片，所以磁头数 = 盘片数*2

柱面:

同一磁道号的不同盘面乘以磁道的扇区数

分区:

连续的柱面，分区大小 = 柱面拥有的扇区数乘以柱面数

得出公式:

硬盘容量 = 单片容量*磁头数

单盘容量 = 磁道扇区数量*磁道数(=柱面数)*512字节

分区表:

在硬盘的MBR中有个64字节“固定大小”的数据结构(分区表)，分区表中的每一个项就是一个分区的“描述符”，表项大小是16字节。所以只能划分4个分区。

分区表必须位于引导扇区中，扇区前446字节是硬盘参数和应道程序，尾部64字节是分区表最后2字节是魔数 0x55aa

```
0E 16 00 75 BC 07 1F 66 61 C3 A1 F6 01 E8 09 00 ... u... fa Äj ö..  
A1 FA 01 E8 03 00 F4 EB FD 8B F0 AC 3C 00 74 09 jU.é..öý. ö<.é..  
B4 0E BB 07 00 CD 10 EB F2 C3 0D 0A 41 20 64 69 '»..Í.ëóÅ..A di  
73 6B 20 72 65 61 64 20 65 72 72 6F 72 20 6F 63 sk read error oc  
63 75 72 72 65 64 00 0D 0A 42 4F 54 4D 47 52 curred..BOOTMGR  
20 69 73 20 63 6F 6D 70 72 65 73 73 65 64 00 0D is compressed..  
0A 50 72 65 73 73 20 43 74 72 6C 2B 41 6C 74 2B .Press Ctrl+Alt+  
44 65 6C 20 74 6F 20 72 65 73 74 61 72 74 0D 0A Del to restart..  
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....  
00 00 00 00 00 00 8A 01 A7 01 BF 01 00 00 55 AA ..... §.ü..U®  
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
```

在这个描述符中有一个id属性，用于表示该分区在理论上可以再次划分出更多子分区(扩展分区，逻辑分区)。

使用fdisk分区:

```
fdisk -c=dos -u=cylinders hd80M.img
```

指定以柱面形式分区

```
Disk hd80M.img: 79.75 MiB, 83607552 bytes, 163296 sectors  
Geometry: 16 heads, 63 sectors/track, 162 cylinders  
Units: cylinders of 1008 * 512 = 516096 bytes  
Sector size (logical/physical): 512 bytes / 512 bytes  
I/O size (minimum/optimal): 512 bytes / 512 bytes  
Disklabel type: dos  
Disk identifier: 0x89062346  
  
Device Boot Start End Cylinders Size Id Type  
hd80M.img1 1 32 32 15.7M 83 Linux  
hd80M.img4 33 162 131 64M 5 Extended  
hd80M.img5 33 50 18 8.8M 66 unknown  
hd80M.img6 51 75 25 12.3M 66 unknown  
hd80M.img7 76 86 11 5.4M 66 unknown  
hd80M.img8 87 110 24 11.8M 66 unknown  
hd80M.img9 111 162 52 25.6M 66 unknown
```

磁盘分区表浅析:

磁盘分区表(DPT),包含了四个元素，每个元素代表了一个分区的元信息。记录了分区的起始地址和大小界限。

DPT位于MBR(主引导扇区中)。

MBR浅析:

它是一段引导程序，位于0盘0道1物理扇区(LBA0扇区),共512字节。由以下三部分构成:

1. 主引导记录MBR (0-0x1bd) 446byte.
2. 磁盘分区表DPT(0x1be-0x1fd) 64byte
3. 结束魔数55AA(0x1fe,0x1ff) 2byte, 标识当前扇区是主引导扇区

分区粒度:

每个分区都要占用独立的柱面，因为mbr占用了一个磁道所以其所在柱面将不会被划入分区，一般为0x3f大小。

扩展分区:

将扩展分区视为主扩展分区，将它划分为多个子扩展分区，把每个子扩展分区视为硬盘，每个子扩展分区就拥有自己的分区表，使用链式结构，将所有子扩展分区的分区表串在一起，形成一个单向链表。

此时在子扩展分区中的最开始的扇区为引导扇区(EBR)同MBR结构相同。

在EBR中的分区表，第一分区表项用来描述包含的逻辑分区的元信息，第二分区表项描述下一个子扩展分区的EBR引导扇区。位于EBR中的分区表相当于总扩展分区中的组建以追赶分不清有的分区表链表节点。

同时，第一分区表项指向该逻辑分区最开始的扇区，即操作系统引导扇区(OBR扇区)

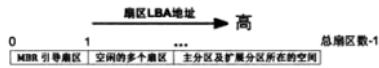
OBR扇区位于分区之内，EBR不位于扇区之内。

下图中的数据宽度为“字节”

表 13-1

分区表项结构

偏移量	数据宽度	描述
0	1	活动分区标记，此标记有两种取值，0x80 和 0。 0x80 表示活动分区，也就是此分区的引导扇区中包含引导程序，可引导。只要是可执行加载的程序都可作为引导程序，不要误以为引导程序一定得是内核提供的程序。尽管通常情况下都是内核加载器。 0 表示非活动分区，不可引导。 其他值非法
1	1	分区起始磁头号
2	1	分区起始扇区号
3	1	分区起始柱面号
4	1	文件系统类型 ID，如 0 表示不可识别的文件系统，1 表示 FAT32
5	1	分区结束磁头号
6	1	分区结束扇区号
7	1	分区结束柱面号
8	4	分区起始偏移扇区
12	4	分区容量扇区数



▲图 13-13 硬盘布局

主引导扇区中分区表部分元信息：

偏移	描述	数值
0x563EFFE00 (0x0)	Master Bootstrap Loader Code	00 00 00 00 00 00 00 00 00 00 ...
0x563EFFB8 (0x1B8)	MBR Disk Signature	46 23 06 89 (2298880838)
Partition Table Entry 1		
0x563EFFB8E (0x1BE)	Active Flag(00h or 80h)	00
0x563EFFBF (0x1BF)	Start Head	1 (0x01)
0x563EFFC0 (0x1C0)	Start Sector	1
0x563EFFC0 (0x1C0)	Start Cylinder	0
0x563EFFC2 (0x1C2)	Partition Type Indicator	83 (131)
0x563EFFC3 (0x1C3)	End Head	15 (0x0F)
0x563EFFC4 (0x1C4)	End Sector	63
0x563EFFC4 (0x1C4)	End Cylinder	31
0x563EFFC6 (0x1C6)	Sectors Preceding Partition	63 (0x0000003F)
0x563EFFCA (0x1CA)	Sectors In Partition	32193 (0x00007DC1)
Partition Table Entry 4		
0x563EFFEE (0x1EE)	Active Flag(00h or 80h)	00
0x563EFFEF (0x1EF)	Start Head	0 (0x00)
0x563EFFFO (0x1F0)	Start Sector	1
0x563EFFFO (0x1F0)	Start Cylinder	32
0x563EFFF2 (0x1F2)	Partition Type Indicator	05 (5)
0x563EFFF3 (0x1F3)	End Head	15 (0x0F)
0x563EFFF4 (0x1F4)	End Sector	63
0x563EFFF4 (0x1F4)	End Cylinder	161
0x563EFFF6 (0x1F6)	Sectors Preceding Partition	32256 (0x00007E00)
0x563EFFFA (0x1FA)	Sectors In Partition	131040 (0x0001FFE0)
0x563EFFFE (0x1FE)	Signature (55 AA)	55 AA (43605)

定位第一个扩展分区的EBR:

0x7e00是主扩展分区的起始扇区偏移.

0x7e00*512 byte

硬盘初始化:

硬盘上有两个ata通道,也称为IDE通道。第一个通道上的两个硬盘(主从)的中断信号挂在8259a的irq14上。使用硬盘控制器的device寄存器中的第4位(dev)区分主从信号
ide.h:

```
/*分区表结构*/
typedef struct _partition
{
    uint32_t start_lba; //起始扇区
    uint32_t sec_cnt; //扇区数
    disk* my_disk; //分区所属的硬盘
    list_elem part_tag; //队列中的标记
    char name[8]; //分区名称
    struct _super_block* sb; //本分区的超级块
    BitMap block_bitmap;//块位图
    BitMap inode_bitmap;//i节点位图
    list open_inodes; //本分区打开的i节点队列
}partition;
/*硬盘结构*/
typedef struct _disk
{
    char name[8]; //本硬盘的名称
    ide_channel* my_channel; //此块硬盘属于哪个ide通道
    uint8_t dev_no;//本硬盘是主盘(0)还是从盘(1)
    partition prim_parts[4];//一个硬盘只有最多四个主分区
    partition logic_parts[8];//逻辑分区,我们设置为最多8个,理论上无限多
}disk;
/*ata通道结构*/
```

```

typedef struct _ide_channel
{
    char name[8];//ata通道名称
    uint16_t port_base; //本通道的起始端口号
    uint8_t irq_no; //本通道的所用的中断号
    lock lock;//通道锁
    bool expecting_intr;//表示等待硬盘的中断是否是由上一次硬盘命令产生的中断，可用于执行接下来的操作
    semaphore disk_done;// 用于阻塞和唤醒驱动程序，硬盘在处理时，用于阻塞当前线程，硬盘工作结束后唤醒当前线程
    disk devices[2]; //一个通道上连接两个硬盘，一主一从
} ide_channel;

```

`port_base` 是本通道的端口基址，对于它要多解释两句，咱们这里只处理两个通道的主板，每个通道的端口范围是不一样的，通道 1（Primary 通道）的命令块寄存器端口范围是 0x1F0~0x1F7，控制块寄存器端口是 0x3F6，通道 2（Secondary 通道）命令块寄存器端口范围是 0x170~0x177，控制块寄存器端口是 0x376。通道 1 的端口可以以 0x1F0 为基数，其命令块寄存器端口在此基数上分别加上 0~7 就可以了，控制块寄存器端口在此基数上加上 0x206，同理，通道 2 的基数就是 0x170。

获取硬盘信息，扫描分区表：

identify命令(0xec)，用于获取硬盘参数，返回结果都是以字为单位。

表 13-8 identify 命令获得的返回信息（部分）

字 偏 移 量	描 述
10~19	硬盘序列号，长度为 20 的字符串
27~46	硬盘型号，长度为 40 的字符串
60~61	可供用户使用的扇区数，长度为 2 的整型

硬盘命名规则：

[x]d[y][n]

x表示硬盘分类, ID硬盘和SCSI硬盘。h代表IDE,s代表SCSI

y表示设备号，区分是第几个设备，其中a是第1个硬盘，b是第二个硬盘，依次类推。

n表示分区号，也就是硬盘上的第几个分区。分区从1开始，依次类推

实现效果：

```

ioqueue_init success
keyboard_init success
tss_init success
syscall_init success
disk sda
    info:SN:BXHD00011
    MODULE:Generic 1234
    SECTORS: 121968
    CAPACITY:59MB
disk sdb
    info:SN:BXHD00012
    MODULE:Generic 1234
    SECTORS: 163296
    CAPACITY:79MB

all partition info:
sdb1 start_lba:0x3F,sec_cnt:0x7DC1
sdb5 start_lba:0x7E3F,sec_cnt:0x46A1
sdb6 start_lba:0xC51F,sec_cnt:0x6231
sdb7 start_lba:0x1278F,sec_cnt:0x2B11
sdb8 start_lba:0x152DF,sec_cnt:0x5E41
sdb9 start_lba:0x1B15F,sec_cnt:0xCC81
ide_init success
kernel main thread pid:0x1

```

手动解析磁盘分区表所得结果一致。



文件系统

2022年7月8日 16:21

文件系统概念简介

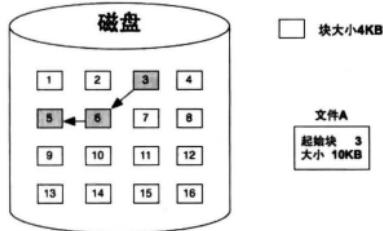
2022年7月8日 16:21

inode、简捷快索引表、文件控制块FCB简介:

块(在windows中称为簇(cu,四声)):

因为硬盘是低速设备，操作系统不会有了一扇区数据就去读写一次磁盘。而是等到数据积攒到一个“块”大小时才会一次性访问硬盘。硬盘的读写单位是扇区，因此一个块是由多个扇区组成，块的大小是扇区大小的整数倍。块是文件系统读写的基本单位。

FAT文件系统:



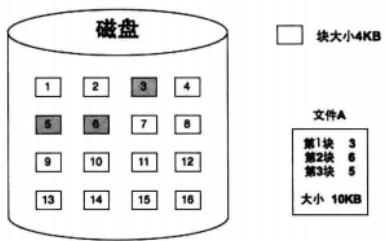
▲图 14-1 文件的链式组织结构

大于1个块的文件如何存储?

在FAT文件系统中，文件被拆分为多个块，块与块之间通过单向链表连接，缺点是如果想访问文件中的某个块，需要从链表头节点开始遍历文件块。

UNIX操作系统的文件系统:

将文件以索引结构来组织。文件系统为每个文件的所有块建立一个索引表，索引表就是包含块地址的数组，每个数组元素就是块的地址，数组元素下标就是文件块的索引，这样访问任意一个块的时候，只需要从缩影表中获得块地址就行了。包含此索引表的索引结构体称为inode(index node)，一个文件必须对应一个inode，磁盘中有多少文件就有多少inode。



▲图 14-2 文件的索引组织结构

索引结构如何存放大文件?

每个索引表共15个索引项，前12个为文件块的直接地址。如果文件大于12个块，那么建立一个一级间接块索引表，表中最多可容纳256个索引块。如果还是不够，则建立二级间接索引表，二级索引表可存放256个一级索引表，三级索引表存放二级索引表。以上三个索引表分别放置在索引表的13-15索引项。

inode:

inode不仅包含了索引表，还包括文件的一些其他属性。每个文件都有一个inode，inode本身也是要占用分区空间的。

所有文件的inode结构大小 + 所有文件的数据块大小 = 分区的容量

所有文件的inode结构大小 = 文件数量 * inode结构大小。

因为分区容量有限，所以必须固定其中一个变量的值，才能初始化文件系统

因为inode结构的大小可以固定(存放大文件时，则大?)，所以文件的数量也就能固定。即一个分区最大创建的文件数量是有限的。

inode_table:

inode的数量等于文件的数量，分区中所有的文件的inode通过一个大表哥来维护，此表格称为inode_table。此数组元素的下标就是文件inode的编号。

inode利用率和分区空间利用率:

inode利用率:

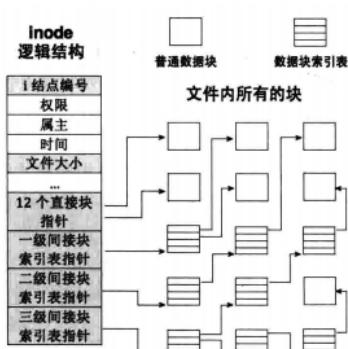
是我们预先判断的要存放的最大文件数量，如果一个超大文件占用了分区空间，则会降低inode利用率

linux命令:df -i

空间利用率:

磁盘空间利用率，也就是使用了多少磁盘空间。

linux命令:df



▲图 14-3 inode 结点结构与各级间接块索引表

目录项与目录简介:

什么是目录?

在linux中，目录和文件都用inode表示，目录也是文件，目录是包含文件的文件(目录文件),如何区目录文件和普通文件呢，通过inode数据块中是指向普通文件数据，还是指向目录文件的目录项。

什么是目录项:

不管文件是普通文件，还是目录文件，它总会存在于某个目录中，所有的文件都存在于根目录 '/' 之下。

```
root@admin1-virtual-machine:/home/admin1/bochs# ls -lai
total 92
546871 drwxrwxrwx 5 admin1 admin1 4096 Jul 20 09:44 .
524504 drwxrwxrwx 18 admin1 admin1 4096 May 24 10:07 bin
546872 drwxrwxrwx 8 admin1 admin1 4096 Jul 20 10:12 bin
548703 -rw-rw-r-- 1 admin1 admin1 16169 Jul 20 10:12 bochsout.txt
527489 drwxrwxr-x 2 admin1 admin1 4096 Jul 20 09:44 build
551576 -rwxr-xr-x 1 root root 47272 Jul 20 09:44 kernel.bin
527361 -rw-rw-r-- 1 admin1 admin1 3680 Jul 18 16:09 makefile
551156 -rw-rw-r-- 1 admin1 admin1 24 May 24 12:03 run
546875 drwxrwxrwx 5 admin1 admin1 4096 May 24 12:03 share
root@admin1-virtual-machine:/home/admin1/bochs#
```

其中第二列开头字母'd'表示是目录文件,'-'表示是一个文件。

这时候回过来看目录:

目录类似于一个包含目录项结构体的数组。(我的理解)

目录箱相当于个文件列表，每个文件在目录中都一个entry(条目，项),各个entry至少包含inode编号，文件名，文件类型。(书上的理解)

目录lib/下的目录项

inode 编号	文件名	文件类型
567272	.	目录
171460	"	目录
567273	php	目录
567320	php.ini	普通文件

▲图 14-6 lib 目录下的目录项

通过文件名来找到文件的实体数据:

1. 在目录中找到文件名所在的目录项
2. 从目录项中获取inode编号
3. 用inode编号作为inode数组的索引下标，找到inode。
4. 从该inode中获取数据块的地址，读取数据块。

目录项和inode都是用于管理文件相关信息的数据结构体，称为文件控制块。创建文件的本质就是创建了文件的文件控制块。

总结：

- (1) 每个文件都有自己单独的 inode，inode 是文件实体数据块在文件系统上的元信息。
- (2) 所有文件的 inode 集中管理，形成 inode 数组，每个 inode 的编号就是在该 inode 数组中的下标。
- (3) inode 中的前 12 个直接数据块指针和后 3 个间接块索引表用于指向文件的数据块实体。
- (4) 文件系统中并不存在具体称为“目录”的数据结构，同样也没有称为“普通文件”的数据结构，统一用同一种 inode 表示。inode 表示的文件是普通文件，还是目录文件，取决于 inode 所指向数据块中的实际内容是什么，即数据块中的内容要么是普通文件本身的数据，要么是目录中的目录项。
- (5) 目录项仅存在于 inode 指向的数据块中，有目录项的数据块就是目录，目录项所属的 inode 指向的所有数据块便是目录。
- (6) 目录项中记录的是文件名、文件 inode 的编号和文件类型，目录项起到的作用有两个，一是粘合文件名及 inode，使文件名和 inode 关联绑定，二是标识此 inode 所指向的数据块中的数据类型（比如是普通文件，还是目录，当然还有更多的类型）。
- (7) inode 是文件的“实质”，但它并不能直接引用，必须通过文件名找到文件名所在的目录项，然后从该目录项中获得 inode 的编号，然后用此编号到 inode 数组中去找相关的 inode，最终找到文件的数据块。

在上面通过文件名来找到文件过程中，如何找到目录文件所在的数据块呢？

通过每个分区的固定根目录，通过递归的方式找到。

超级块与文件系统布局

什么是超级块?

我们知道我们需要创建一个inode数组，但是数组在哪个扇区，数组的大小是多少。同时每个分区都有自己的根目录，根目录所在扇区是多少？以及一些其他的描述文件系统的元信息的元信息，应该有一个固定的位置用于获取，这个地方就是超级块。

提一下：文件系统是针对各个分区来进行管理的。

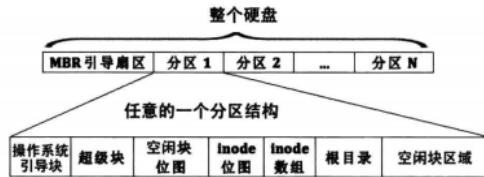
创建一个基本的文件系统，需要维护哪些信息:

超级块

魔数
数据块数量
inode 数量
分区起始扇区地址
空闲块位图地址
空闲块位图大小
inode 位图地址
inode 位图大小
inode 数组地址
inode 数组大小
根目录地址
根目录大小
空闲块起始地址
...

超级块的位置和大小应该是固定的，我们将他固定存储在各分区的第2个扇区，大小为一扇区。

我们计划的磁盘布局：



创建文件系统

2022年7月21日 9:08

在硬盘hd80M.img各分区上创建文件系统。

我们的基本规定：

一块 = 一扇区

ionde中的数据块只支持一级间接块

文件的最大支持数量(inode)为4096

创建文件系统:

- 首先创建一个超级块,根据我们基本规定填充超级块的信息。
 - 将超级块写入到磁盘分区的第二个扇区,第一个扇区为引导扇区(包含有分区信息)。
 - 申请一块堆空间(buf),在此buf中分别填充“空闲块位图信息”、“inode位图信息”、“inode数组”、“上级目录和当前目录的目录项”。写入到磁盘中保存。
 - 通过全局channels数组,获取通道遍历磁盘,遍历分区。检查每个分区是否具有我们超级块设置的魔数,如果不存在则写入文件系统。

Makefile:

创建文件系统：

第一次启动：

```
Bochs x86 emulator, http://bochs.sourceforge.net/
[File] [Edit] [View] [Run] [Break] [Registers] [Stack] [CPU] [Memory] [Disk]
[USB] [Copy] [Paste] [Clipboard] [Base] [Format] [CPU Control]
[CPU Control]

block_bitmap_sectors:0x5
inode_bitmap_lba:0x7E46
inode_bitmap_sectors:0x1
inode_table_lba:0x7E47
inode_table_sectors:0x260
data_start_lba:0x80A7
root_dir_lba:0x80A7
sdb5 format done
formatting sdb's partition sdb5 .....
sdb5 info:
    magic:0x10190498
    part_lba_base:0x7E3F
    all_sectors:0x46A1
    inode_cnt:0x1000
    block_bitmap_lba:0x7E41
    block_bitmap_sectors:0x5
    inode_bitmap_lba:0x7E46
    inode_bitmap_sectors:0x1
    inode_table_lba:0x7E47
    inode_table_sectors:0x260
    data_start_lba:0x80A7
root_dir_lba:0x80A7
sdb5 format done
kernel main thread pid:0x1
```

第二次启动:

```
all partition info:  
sdb1 start_lba:0x3F,sec_cnt:0x7DC1  
sdb5 start_lba:0x7E3F,sec_cnt:0x46A1  
sdb6 start_lba:0xC51F,sec_cnt:0x6231  
sdb7 start_lba:0x1278F,sec_cnt:0x2B11  
sdb8 start_lba:0x152DF,sec_cnt:0x5E41  
sdb9 start_lba:0xB15F,sec_cnt:0xCC81  
ide_init success  
sdb1 has filesystem  
sdb5 has filesystem  
sdb6 has filesystem  
sdb7 has filesystem  
sdb8 has filesystem  
sdb9 has filesystem  
kernel main thread pid:0x1
```

挂载分区:

1. 遍历分区链表，将要挂载的分区名传入遍历回调函数
 2. 在遍历回调函数中，分区名相同时执行挂载操作，如不存在则退出
 3. 从list_elem定位partition结构体，再定位到所属磁盘
 4. 从磁盘中读取第二个分区(超级块)，到堆内存
 5. 在堆中创建inode位图，block位图，因为磁盘的操作都是一个扇区为大小，所以创建位图的时候不要使用bitmap_len而是使用超级块中的block_bitmap_sects。
 6. 将硬盘中的inode位图和block位图复制到内存中。返回true结束链表遍历。

```
ide_init success
sdb1 has filesystem
sdb5 has filesystem
sdb6 has filesystem
sdb7 has filesystem
sdb8 has filesystem
sdb9 has filesystem
arg_name:sdb1,part->name:sdb1
mount sdb1 done!
kernel main thread pid:0x1
```





虽然什么都没做
但还是辛苦我自己了

文件描述符简介

2022年7月26日 15:07

文件描述符原理:

inode(给操作系统使用):

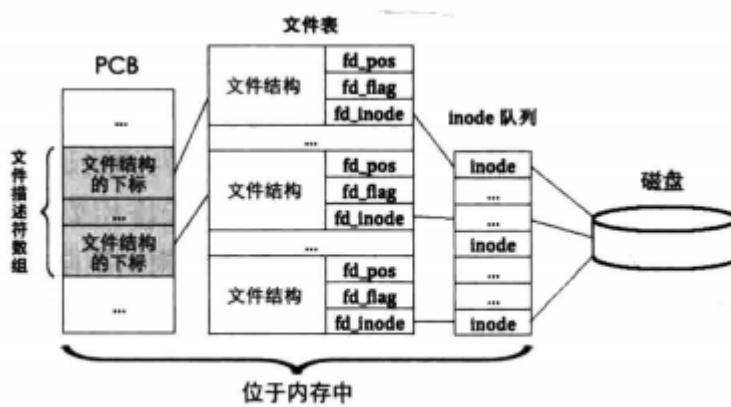
用于描述文件的物理存储位置、权限等信息。

文件描述符(给用户使用):

用于描述“文件被打开”后，文件读写偏移量等信息。

一个文件只有一个inode，一个inode可以有多个文件描述符

linux中的读写文件操作:



▲图 14-16 文件描述符与 inode 关联关系

通过open函数打开文件后返回一个整数(文件描述符数组中的下标)。如何通过此整数得到文件所在的数据块呢?

1. 通过此下标，在pcb中的文件描述符数组中得到文件表数组下标
2. 在全局文件表数组中得到文件结构，再取出fd_inode属性
3. 在已打开的inode链表中寻找是否存在此inode，如存在则得到文件扇区地址，如不存在则在硬盘inode_table中找到此inode并将其添加进inode队列，使其文件结构fd_inode元素指向此inode。这部分后面再写。

我们已经知道了文件读写的过程，那么如何添加一个文件描述符呢？

1. 在全局inode队列中新建一个inode，返回此inode的地址
2. 在全局文件表中，填充一个文件结构体，并使fd_inode指向上面返回的地址，返回此文件结构在文件表中的下标。
3. 在pcb中的文件描述符数组中添加文件结构体下标，最后返回文件描述符项在文件描述符数组中的下标。

文件描述符的实现:

task_struct:

```
/*
进程线程的pcb
*/
typedef struct _task_struct
{
    uint32_t* self_kstack; //内核栈
    int16_t pid;//pid
    enum task_status status;
    uint8_t priority; //优先级
    uint8_t ticks;//每次运行的滴答数
    char name[16];//进程或线程的名称
    uint32_t elapsed_ticks;//已运行的滴答数总和
    int32_t fd_table[MAX_FILES_OPEN_PER_PROC]; //文件描述符数组
    list_elem general_tag;//线程就绪队列的节点
    list_elem all_list_tag;//所有线程队列的节点
    uint32_t* pgdir; //虚拟页目录表地址(CR3,线程为NULL)
    struct _virtual_pool userprog_vaddr; //用户进程的虚拟地址池,用于管理进程堆
    mem_block_desc u_block_desc[DESC_CNT];//用户进程内存块描述符数组
    uint32_t stack_magic;//魔数,检测栈溢出
}task_struct,*ptask_struct;
```

init_thread:

```
105     /*标准输入输出流*/
106     thread_pcb->fd_table[0] = 0;
107     thread_pcb->fd_table[1] = 1;
108     thread_pcb->fd_table[2] = 2;
109     /*初始化其他文件描述符为 -1 */
110     uint8_t id_x = 3;
111     while (id_x<MAX_FILES_OPEN_PER_PROC)
112     {
113         thread_pcb->fd_table[id_x] = -1;
114     }
```

文件操作相关的基础函数

2022年7月26日 16:01

```
/*
 * @brief
 * 将最上层路径名解析出来
 * pathname:完整路径
 * name_store: 保存最上层路径
 * 路径为空返回NULL, 其余返回pathname
 */
static char* path_parse(char* pathname, char* name_store);
```

例子:

/a/b/c/d

思路:

1. 判断 pathname 第一个字符是不是'/'，直到将 pathname 指针移至非'/'的位置。

2. 复制 pathname 指针处的字符至直到 name_store 遇见下一个'/'或者字符串结尾 '0'。这里我们是没有复制末尾的0，所以 name_store 缓冲区必须初始化为0。

3. 此时 pathname 指向类似于'/b/c/d'的字符串。

4. 返回 pathname，以便调用函数遍历路径。

```
/*
 * @brief
 * 搜索文件, 成功返回其inode, 失败返回-1
 * pathname: 文件完整路径(从根目录开始)
 */
static int32_t search_file(char* pathname, path_seach_record* searched_record);
```

例子:

/a/c/d

思路:

1. pathname 是从根目录出发的全路径，searched 是用于保存搜索结果的结构体

2. 判断 pathname 是否是根目录，如果是则直接填充 searched 结构体后返回。

3. 检查路径格式 '/x/x/x'，并取出第一层路径。

4. 使用 search_dir_entry 判断是否存在此路径或文件。

5. 如果是文件，则直接填充 searched.file_type = 文件。因为

6. 如果是目录则关闭当前上次打开的目录，并打开当前的目录，填充 searched.parent_dir

7. 还有一些琐碎的细节，这里不说了

创建文件:

```
mount sdb1 done!
kernel main thread pid:0x1
search_file: no file or directory
start to create file file1
ide write lba:0x2A0,buf:0xC010490C
ide write lba:0x40,buf:0xC010490C
ide write lba:0x40,buf:0xC010490C
ide write lba:0x49,buf:0xC010146C
start_lba:602
open file id:3
```

```
mount sdb1 done!
kernel main thread pid:0x1
create file failed,file already exists
open file id:255
```

关闭文件:

```

all partition info:
sdb1 start lba:0x3F,sec_cnt:0x7DC1
sdb5 start lba:0x7E3F,sec_cnt:0x46A1
sdb6 start lba:0xC51F,sec_cnt:0x6231
sdb7 start lba:0x127BF,sec_cnt:0x6231
sdb8 start lba:0x189FF,sec_cnt:0x6231
sdb9 start lba:0x1EC6F,sec_cnt:0x9171
ide_init success
sdb1 has filesystem
sdb5 has filesystem
sdb6 has filesystem
sdb7 has filesystem
sdb8 has filesystem
sdb9 has filesystem
part_name:sdb1,part->name:sdb1
mount sdb1 done!
kernel main thread pid:0x1
open file id:3
close 1
cur_part start_lba:682
cur_thread file_table virtual address:0xC009E024
global file table virtual address:0xC0011BAA
EIP:0xC00007760
IPS: 49,601M

```

```

Next at t=162876499
(0) [0x000000007760] 0008:c0007760 (unk. ctxt): mov dword ptr ss:[esp+4], e
ax ; 89442404
<bochs:> x /10wx 0xc0011b0
[bochs]:
0xc0011b0 <bogus+ 0:> 0x00000000 0x00000000 0x00000000
0x00000000
0xc0011b0 <bogus+ 16:> 0x00000000 0x00000000 0x00000000
0x00000000
0xc0011b0 <bogus+ 32:> 0x00000000 0x00000000 0x00000000
<bochs:> x /10wx 0xc009e024
[bochs]:
0xc009e024 <bogus+ 0:> 0x00000000 0x00000001 0x00000002
0xfffffff
0xc009e034 <bogus+ 16:> 0xfffffff 0xfffffff 0xfffffff
0xc009e044 <bogus+ 32:> 0xc0011a30 0xc0011a38
<bochs:> c

```

可以看到在全局文件表中和线程文件表中都被恢复了默认值也就是关闭成功了。

文件写入:

小于12个直接块的写入测试没问题。

```

Activities Bochs ▾
Bochs x86 emulator, http://bochs.sourceforge.net
sdb7 start_lba:0x127BF,sec_cnt:0x6231
Thunderbird Mail lba:0x1EC6F,sec_cnt:0x9171
ide_init success
sdb1 has filesystem
sdb5 has filesystem
sdb6 has filesystem
sdb7 has filesystem
sdb8 has filesystem
sdb9 has filesystem
part_name:sdb1,part->name:sdb1
mount sdb1 done!
kernel main thread pid:0x1
open file id:3
ide write lba:0x2AC,buf:0xC010160C
file write at lba:0x2AC
file write lba:0x4A,buf:0xC010440C
inode sync lba:0x4A,inode_pos.off_size:0x4C
write size:25
close 1
cur_part start_lba:682
cur_thread file_table virtual address:0xC009E024
global file table virtual address:0xC0011BAA
EIP:0xC00007760
IPS: 66,388M

```

文件读取:

```

Bochs x86 emulator, http://bochs.sourceforge.net/
char buf[25] = "write buf byte 123456789";
buf[24] = 0;
int32_t fd_id = sys_open("/file1", O_WRITE | O_READ);
printf("open file id:%d\n", fd_id);
if(fd_id == -1){
    //文件打开或创建失败
    while(1);
}
int32_t write_size = sys_write(fd_id, buf, 25);
printf("write size:%d\n", write_size);
write_size = sys_read(fd_id, buf, 25);
printf("read size:%d,buf ptr:0x%08X,print buf:%s\n", write_size, buf, buf);
bool satatus = sys_close(fd_id);
if(satatus){
    printf("close true\n");
} else {
    printf("close error\n");
}
show_fs();

```

修改文件指针(SEEK_SET +1):

```

*****filesystems_init end*****
kernel main thread pid:0x1
open file id:3
file_read: read file size:25
read size:25,buf ptr:0xC009EFB3,print buf:rite buf byte 123456789
close true
cur_part start_lba:682
cur_thread file_table virtual address:0xC009E024
global file table virtual address:0xC0011BAA
EIP:0xC000086C7
*****show_fs end*****

```

删除文件:

包括以下操作:

inode操作:

1. 回收inode位图
2. inode_table
3. inode中的直接块和间接块

父目录操作:

1. 回收目录项
2. 修改目录inode的i_bsize

创建目录:

- (1) 确认待创建的新目录在文件系统上不存在。
- (2) 为新目录创建 inode。
- (3) 为新目录分配 1 个块存储该目录中的目录项。
- (4) 在新目录中创建两个目录项“.” 和 “..”，这是每个目录都必须存在的两个目录项。
- (5) 在新目录的父目录中添加新目录的目录项。
- (6) 将以上资源的变更同步到硬盘。

下面我们在 `mkdir` 的内核部分——`sys_mkdir` 中实现以上功能，见代码 14-40。

磁盘数据展示

根目录:

```
5:5400h: 2E 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ..  
5:5410h: 00 00 00 00 02 00 00 00 2E 2E 00 00 00 00 00 00 ..  
5:5420h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ..  
5:5430h: 66 69 6C 65 31 00 00 00 00 00 00 00 00 00 00 00 file1..  
5:5440h: 01 00 00 00 01 00 00 00 64 69 72 31 00 00 00 00 .. dir1..  
5:5450h: 00 00 00 00 00 00 00 00 02 00 00 00 00 00 00 00 ..  
5:5460h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ..  
5:5470h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ..  
5:5480h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ..  
5:5490h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ..
```

dir1目录:

```
5:5800h: 2E 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ..  
5:5810h: 02 00 00 00 02 00 00 00 2E 2E 00 00 00 00 00 00 ..  
5:5820h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ..  
5:5830h: 64 69 72 32 00 00 00 00 00 00 00 00 00 00 00 00 dir2..  
5:5840h: 03 00 00 00 02 00 00 00 00 00 00 00 00 00 00 00 ..  
5:5850h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ..  
5:5860h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ..  
5:5870h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ..  
5:5880h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ..
```

dir2目录:

```
5:5A00h: 2E 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ..  
5:5A10h: 03 00 00 00 02 00 00 00 2E 2E 00 00 00 00 00 00 ..  
5:5A20h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ..  
5:5A30h: 66 69 6C 65 32 00 00 00 00 00 00 00 00 00 00 00 file2..  
5:5A40h: 04 00 00 00 01 00 00 00 00 00 00 00 00 00 00 00 ..  
5:5A50h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ..
```

```
Bochs x86 emulator, http://bochs.sourceforge.net/  
USER Sync Page Suspend Reset suspend Power  
inode sync lba:0x10,inode_pos.off_size:0x0  
open file id:3  
search_file: no file or directory  
block bitmap value:3  
block_idx:0x2  
sync_dir_entry lba:0x2AA  
***** sync *****  
inode sync lba:0x40,inode_pos.off_size:0x0  
inode sync lba:0x40,inode_pos.off_size:0x98  
***** sync end*****  
search_file: no file or directory  
block bitmap value:  
block_idx:0x3  
sync_dir_entry lba:0x2AC  
***** sync *****  
inode sync lba:0x40,inode_pos.off_size:0x98  
inode sync lba:0x40,inode_pos.off_size:0xE4  
***** sync end*****  
sys_mkdir true  
search_file: no file or directory  
start to create file file2  
sync_dir_entry lba:0x2AD  
inode sync lba:0x40,inode_pos.off_size:0xE4  
inode sync lba:0x40,inode_pos.off_size:0x30  
IP: 127.912m
```

Main:

```
satatus = sys_mkdir("/dir1");  
satatus = sys_mkdir("/dir1/dir2");  
if(satatus){  
    printf("sys_mkdir true\n");  
}else{  
    printf("sys_mkdir error\n");  
}  
  
int32_t fd_id2 = sys_open("/dir1/dir2/file2",O_CREAT);  
printf("open file id:%d\n",fd_id2);  
if(fd_id2<-1){  
    //文件打开或创建失败  
    while(1);  
}  
  
satatus = sys_clos(fd_id);  
if(satatus){  
    printf("close true\n");  
}else{  
    printf("close error\n");  
}
```

遍历目录:

遍历目录前要打开目录，遍历结束后要关闭目录。

先实现这两个函数

```
68  
69     dir* pdir = sys_opendir("/dir1/dir2");  
70     if(pdir==NULL){  
71         dbg_printf("open dir failed\n");  
72     }else{  
73         dbg_printf("dir open sucess\n");  
74     }  
75     satatus = sys_closedir(pdir);  
76     if(satatus){  
77         dbg_printf("close dir true\n");  
78     }else{  
79         dbg_printf("close dir false\n");  
80     }  
81  
82  
83  
84
```

```
all partition info:  
sdb1 start_lba:0x3F,sec_cnt:0x7DC1  
sdb5 start_lba:0x73F,sec_cnt:0x46A1  
sdb6 start_lba:0xC51F,sec_cnt:0x6231  
sdb7 start_lba:0x1278F,sec_cnt:0x6231  
sdb8 start_lba:0x1B9FF,sec_cnt:0x6231  
sdb9 start_lba:0x1EC6F,sec_cnt:0x9171  
lde init success  
sdb1 has filesystem  
sdb5 has filesystem  
sdb6 has filesystem  
sdb7 has filesystem  
sdb8 has filesystem  
sdb9 has filesystem  
mount_partition run  
part_name:sdb1,part->name:sdb1  
blk0k bitmap value:0  
mount sdb1 done!  
blk0k bitmap value:0  
blk0k bitmap value:0  
*****filesystems_init end*****  
kernel main thread pid:0x1  
dir open success  
close dir true  
  
IPS: 46,994H
```

```
遍历目录: 9.10修改bug后再次截图
sdb0 has filesystem
sdb9 has filesystem
mount_partition run
part_name:sdb1,part->name:sdb1
block bitmap value:F
mount sdb1 done!
block bitmap value:F
block bitmap value:F
block bitmap value:F
block bitmap value:F
*****filesystem_init end*****
kernel main thread pid:0x1
*****sys_mkdir*****
sys_mkdir:file or directory /dir1 exist
*****sys_mkdir*****
sys_mkdir:file or directory /dir1/dir2 exist
sys_mkdir error
dir open success
inode_no fileType fileName
1 Z .
6 Z ..
7 Z dir2
*****test*****
block bitmap lba:0x41
*****test*****
close dir true

IPS: 139.058M          R: NUM  QOPS SGRL HD:0-MHD:0-S
```

删除空目录:

```
删除dir2:  
sd87 has filesystem  
sd88 has filesystem  
sd89 has filesystem  
mount_partition run  
part_name:sdb1,part->name:sdb1  
block bitmap value:F  
mount sdb1 done!  
block bitmap value:F  
block bitmap value:F  
*****filesystems_init end*****  
kernel main thread pid:0x1  
*****sys_mkdir*****  
sys_mkdir:file or directory /dir1 exist  
*****sys_mkdir*****  
sys_mkdir:file or directory /dir1/dir2 exist  
sys_mkdir error  
dir open success  
inode_no      fileType      fileName  
1             2              .  
0             2              ..  
*****test*****  
block bitmap lba:0x41  
*****test*****  
close dir true
```

显示当前工作目录:

```
Bochs x86 emulator, http://bochs.sourceforge.net/
[Icons] [File] [Edit] [View] [Run] [Break] [Registers] [Stack] [CPU] [Memory] [Disk] [PCI] [USB] [Serial] [Parallel] [Video] [Help] [About] [Exit]

block idx:0x2
sync_dir_entry lba:0x2AA
make_dir ok: dir info:
dir_name:/dir1,dir_isize:48,dir_no:1
*****sys_end*****
*****sys_mkdir*****
NO sync memory block bitmap value:7
block idx:0x3
sync_dir_entry lba:0x2AC
make_dir ok: dir info:
dir_name:/dir1/dir2,dir_isize:48,dir_no:2
*****sys_end*****
sys_mkdir true
dir open success
inode_no fileType fileName
0 Z .
0 Z ..
1 Z dir1
*****test*****
block bitmap lba:0x41
*****test*****
close dir true
cwd:/
cwd now:/dir1/dir2

IPS: 136,098M | 0: 0MHz | CPU0 | CPU1 | IO-DMA | INT-DMA | 0: 0MHz | 1: 0MHz | 2: 0MHz | 3: 0MHz | 4: 0MHz | 5: 0MHz | 6: 0MHz | 7: 0MHz | 8: 0MHz | 9: 0MHz | 10: 0MHz | 11: 0MHz | 12: 0MHz | 13: 0MHz | 14: 0MHz | 15: 0MHz | 16: 0MHz | 17: 0MHz | 18: 0MHz | 19: 0MHz | 20: 0MHz | 21: 0MHz | 22: 0MHz | 23: 0MHz | 24: 0MHz | 25: 0MHz | 26: 0MHz | 27: 0MHz | 28: 0MHz | 29: 0MHz | 30: 0MHz | 31: 0MHz | 32: 0MHz | 33: 0MHz | 34: 0MHz | 35: 0MHz | 36: 0MHz | 37: 0MHz | 38: 0MHz | 39: 0MHz | 40: 0MHz | 41: 0MHz | 42: 0MHz | 43: 0MHz | 44: 0MHz | 45: 0MHz | 46: 0MHz | 47: 0MHz | 48: 0MHz | 49: 0MHz | 50: 0MHz | 51: 0MHz | 52: 0MHz | 53: 0MHz | 54: 0MHz | 55: 0MHz | 56: 0MHz | 57: 0MHz | 58: 0MHz | 59: 0MHz | 60: 0MHz | 61: 0MHz | 62: 0MHz | 63: 0MHz | 64: 0MHz | 65: 0MHz | 66: 0MHz | 67: 0MHz | 68: 0MHz | 69: 0MHz | 70: 0MHz | 71: 0MHz | 72: 0MHz | 73: 0MHz | 74: 0MHz | 75: 0MHz | 76: 0MHz | 77: 0MHz | 78: 0MHz | 79: 0MHz | 80: 0MHz | 81: 0MHz | 82: 0MHz | 83: 0MHz | 84: 0MHz | 85: 0MHz | 86: 0MHz | 87: 0MHz | 88: 0MHz | 89: 0MHz | 90: 0MHz | 91: 0MHz | 92: 0MHz | 93: 0MHz | 94: 0MHz | 95: 0MHz | 96: 0MHz | 97: 0MHz | 98: 0MHz | 99: 0MHz | 100: 0MHz | 101: 0MHz | 102: 0MHz | 103: 0MHz | 104: 0MHz | 105: 0MHz | 106: 0MHz | 107: 0MHz | 108: 0MHz | 109: 0MHz | 110: 0MHz | 111: 0MHz | 112: 0MHz | 113: 0MHz | 114: 0MHz | 115: 0MHz | 116: 0MHz | 117: 0MHz | 118: 0MHz | 119: 0MHz | 120: 0MHz | 121: 0MHz | 122: 0MHz | 123: 0MHz | 124: 0MHz | 125: 0MHz | 126: 0MHz | 127: 0MHz | 128: 0MHz | 129: 0MHz | 130: 0MHz | 131: 0MHz | 132: 0MHz | 133: 0MHz | 134: 0MHz | 135: 0MHz | 136: 0MHz | 137: 0MHz | 138: 0MHz | 139: 0MHz | 140: 0MHz | 141: 0MHz | 142: 0MHz | 143: 0MHz | 144: 0MHz | 145: 0MHz | 146: 0MHz | 147: 0MHz | 148: 0MHz | 149: 0MHz | 150: 0MHz | 151: 0MHz | 152: 0MHz | 153: 0MHz | 154: 0MHz | 155: 0MHz | 156: 0MHz | 157: 0MHz | 158: 0MHz | 159: 0MHz | 160: 0MHz | 161: 0MHz | 162: 0MHz | 163: 0MHz | 164: 0MHz | 165: 0MHz | 166: 0MHz | 167: 0MHz | 168: 0MHz | 169: 0MHz | 170: 0MHz | 171: 0MHz | 172: 0MHz | 173: 0MHz | 174: 0MHz | 175: 0MHz | 176: 0MHz | 177: 0MHz | 178: 0MHz | 179: 0MHz | 180: 0MHz | 181: 0MHz | 182: 0MHz | 183: 0MHz | 184: 0MHz | 185: 0MHz | 186: 0MHz | 187: 0MHz | 188: 0MHz | 189: 0MHz | 190: 0MHz | 191: 0MHz | 192: 0MHz | 193: 0MHz | 194: 0MHz | 195: 0MHz | 196: 0MHz | 197: 0MHz | 198: 0MHz | 199: 0MHz | 200: 0MHz | 201: 0MHz | 202: 0MHz | 203: 0MHz | 204: 0MHz | 205: 0MHz | 206: 0MHz | 207: 0MHz | 208: 0MHz | 209: 0MHz | 210: 0MHz | 211: 0MHz | 212: 0MHz | 213: 0MHz | 214: 0MHz | 215: 0MHz | 216: 0MHz | 217: 0MHz | 218: 0MHz | 219: 0MHz | 220: 0MHz | 221: 0MHz | 222: 0MHz | 223: 0MHz | 224: 0MHz | 225: 0MHz | 226: 0MHz | 227: 0MHz | 228: 0MHz | 229: 0MHz | 230: 0MHz | 231: 0MHz | 232: 0MHz | 233: 0MHz | 234: 0MHz | 235: 0MHz | 236: 0MHz | 237: 0MHz | 238: 0MHz | 239: 0MHz | 240: 0MHz | 241: 0MHz | 242: 0MHz | 243: 0MHz | 244: 0MHz | 245: 0MHz | 246: 0MHz | 247: 0MHz | 248: 0MHz | 249: 0MHz | 250: 0MHz | 251: 0MHz | 252: 0MHz | 253: 0MHz | 254: 0MHz | 255: 0MHz
```

```
105     char cwd_buf[32] = {0};  
106     sys_getcwd(cwd_buf,32);  
107     dbg_printf("( cwd:%s\n", cwd_buf);  
108     sys_chdir((char*)"/dir1\\dir2");  
109     sys_getcwd(cwd_buf,32);  
110     dbg_printf(" cwd now:%s\n", cwd_buf);
```

获取文件属性:

```
block bitmap 1ba:0x41
*****test*****
close dir true
cwd:/
cwd now:/dir1/dir2
file_type    inode_no      inode_size
2             0              72
2             1              72
IPS: 139,479H
next_d1_l=0
(0) [0x0000fffff0] f000:fff0 (unk. ctxt): jmpf 0xf000:e05b      ; ea5be00
(0)
```

```
stat obj_stat;
stat obj_stati;
sys_stat("/",&obj_stat);
sys_stat("/dir1",&obj_stati);
dbg_printf("file_type      inode_no      inode_size\n");
dbg_printf("%d      %d      %d\n",obj_stat.st_file_type,obj_stat.st_ino,obj_stat.st_size);
dbg_printf("%d      %d      %d\n",obj_stati.st_file_type,obj_stati.st_ino,obj_stati.st_size);
while (1){
}
```

系统交互

2022年09月12日 16:35

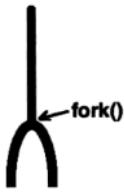
fork

2022年09月12日 16:35

什么是fork:

fork的作用是克隆进程，相当于同一个程序多次加载执行。

执行流:



▲图 15-3 fork 叉子

实现fork:

克隆一个进程就要克隆此进程的所有资源。

1. 进程的pcb
2. 程序体
3. 用户栈
4. 内核栈
5. 虚拟地址池
6. 页表

实现结果正常输出:

但是有一个疑问谁占用的3的pid? ? ? ?

9.20回答回答 (之前创建了一个懒线程, 用于在其他所有线程休息时运行。)

```
#include "fork.h"
#include "../thread/thread.h"
#include "../fs/file.h"
#include "../lib/kernel/memory.h"
#include "../lib/kernel/string.h"
#include "../lib/kernel/global.h"
#include "../lib/kernel/debug.h"
#include "../lib/kernel/stdio-kernel.h"
#include "../lib/kernel/interrupt.h"
#include "process.h"
extern void intr_exit(void);
/** 
 * @brief
 * 将父进程的pcb和内核栈拷给子进程
 */
static int32_t copy_pcb_vaddrbitmap_stack0(task_struct* child_thread,
task_struct* parent_thread)
{
    /* 复制pcb整个页, 里面包含进程pcb信息和0级的栈
    memcpy(child_thread, parent_thread, PG_SIZE);
    child_thread->pid = fork_pid();
    child_thread->elapsed_ticks = 0;
    child_thread->status = TASK_READY;
    child_thread->ticks = child_thread->priority;
    child_thread->parent_pid = parent_thread->pid;
    child_thread->general_tag.next = child_thread->general_tag.prev = NULL;
    child_thread->all_list_tag.prev = child_thread->all_list_tag.next = NULL;
    // 复制父进程的虚拟地址位图
    uint32_t bitmap_pg_cnt = DIV_ROUND_UP((0xc0000000 - USER_VADDR_START) / PG_SIZE / 8, PG_SIZE);
    void* vaddr_btmp = get_kernel_pages(bitmap_pg_cnt);
    memcpy(vaddr_btmp, child_thread->userprog_vaddr.bitmap_vaddr.bits, bitmap_pg_cnt * PG_SIZE);
    child_thread->userprog_vaddr.bitmap_vaddr.bits = vaddr_btmp;
    // debug
    assert(strlen(child_thread->name)<11);
    strcat(child_thread->name, "_fork");
    return 0;
}
/** 
 * @brief
 * 复制子进程的进程体(代码和数据)和用户栈
 */
static void copy_body_stack3(task_struct* child_thread,
task_struct* parent_thread, void* buf_page)
{
    uint8_t* vaddr_btmp = parent_thread->userprog_vaddr.bitmap_vaddr.bits;
    uint32_t btmp_bytes_len = parent_thread->userprog_vaddr.bitmap_vaddr.bitmap_byte_len;
    uint32_t vaddr_start = parent_thread->userprog_vaddr.vaddr_start;
    uint32_t vaddr_end = parent_thread->userprog_vaddr.vaddr_end;
    dbg_printf("vaddr_btmp:0x%x, btmp_bytes_len:0x%x, vaddr_start:0x%x\n", vaddr_btmp, btmp_bytes_len, vaddr_start);
    dbg_printf("bitmap is empty:%d\n", bitmap_isempty(&parent_thread->userprog_vaddr.bitmap_vaddr));
    while(idx_byte<btmp_bytes_len)
    {
        if(vaddr_btmp[idx_byte])
        {
            idx_bit = 0;
            while(idx_bit<8)
            {
                if((BITMAP_MASK << idx_bit) & vaddr_btmp[idx_byte])
                {
                    // 父进程被使用的虚拟地址页
                    prog_vaddr = (idx_byte * 8+ idx_bit) * PG_SIZE + vaddr_start;
                    dbg_printf("copy virtual mem addr:0x%llx, idx_bit:%d, idx_byte:%d\n", prog_vaddr, idx_bit, idx_byte);
                    memcpy(buf_page, (void*)prog_vaddr, PG_SIZE);
                }
                // 切换为子进程的cr3
                page_dir_activate(child_thread);
                get_a_page_without_opvaddrbitmap(PF_USER, prog_vaddr);
                memcpy(prog_vaddr, buf_page, PG_SIZE);
                // 恢复父进程cr3
                page_dir_activate(parent_thread);
            }
            idx_bit++;
        }
        idx_byte++;
    }
}
/** 
 * 为子进程构建thread_stack和修改返回值
 */
static int32_t build_child_stack(task_struct* child_thread)
{
    intr_stack* intr_0_stack = (intr_stack*)((uint32_t)child_thread + PG_SIZE - sizeof(intr_stack));
    intr_0_stack->eax = 0;
    // 这里需要重新看一下代码
    // 通过修改内核栈指针, 指向中断栈-5的位置
    // 为什么是-5?
    // 因为在switch_to中, 通过4个pop+一个ret切换任务。我们这里构建好ret时的返回位置为中断返回
    // (返回后的子进程eip也就是父进程进入中断后保存的eip)
    uint32_t* ebp_ptr_in_thread_stack = (uint32_t*)intr_0_stack - 5;
    child_thread->self_kstack = ebp_ptr_in_thread_stack;
    uint32_t* ret_addr_in_thread_stack = (uint32_t*)intr_0_stack - 1;
    *ret_addr_in_thread_stack = (uint32_t)intr_exit;
    return 0;
}
/** 
 * 更新inode打开数
 */
static void update_inode_open_ctns(task_struct* thread)
{
    int32_t local_fd = 3, global_fd = 0;
    while(local_fd < MAX_FILES_OPEN_PER_PROC) // 此处和书中不一致
        global_fd = thread->fd_table[local_fd];
        if(global_fd != -1)
            file_table[global_fd].fd_inode->i_open_ctns += 1;
        local_fd++;
}
/**
```

```

* @brief
* 拷贝父进程本身所占资源给子进程
*/
static int32_t copy_process(task_struct* child_thread, task_struct* parent_thread) {
    /* 中转缓冲区 */
    void* buf_page = get_kernel_pages(1);
    if(buf_page == NULL) {
        return -1;
    }
    /* 复制pcb, 内核栈信息, 虚拟地址位图信息 (大小为1页)
     * 如果copy_pcb_vaddrbitmap_stack0(child_thread, parent_thread) == -1 {
     *     return -1;
     * }
     * 为子进程创建页表
     * child_thread->pgdir = create_page_dir();
     * if(child_thread->pgdir == NULL) {
     *     return -1;
     * }
     * 复制父进程3环所有内存信息
     * copy_body_stack3(child_thread, parent_thread, buf_page);
     * 构建子进程thread_stack和修改返回值pid
     * build_child_stack(child_thread);
     * 更新文件inode的打开次数
     * update_inode_open_cnts(child_thread);
     * mfree_page(PF_KERNEL, buf_page, 1);
     *
     * return 0;
    */
    /**
     * @brief
     * 克隆用户进程
     * 成功返回子进程pid, 失败返回-1
     */
    pid_t sys_fork(void) {
        task_struct* parent_thread = get_running_thread_pcb();
        task_struct* child_thread = get_kernel_pages(1);
        // 构建子进程pcb
        if(child_thread == NULL) {
            return -1;
        }
        // 保证是在关中断, 且是在3环进程中使用
        assert(INTR_OFF == get_intr_status() && (parent_thread->pgdir != NULL));
        if(copy_process(child_thread, parent_thread) == -1) return -1;
        // 添加子进程到就绪列表和所有线程链表
        assert(!list_elem_find(&thread_ready_list, &child_thread->general_tag));
        list_append(&thread_ready_list, &child_thread->general_tag);
        assert(!list_elem_find(&thread_all_list, &child_thread->all_list_tag));
        list_append(&thread_all_list, &child_thread->all_list_tag);
        return child_thread->pid;
    }
}

```

实现一个简单的Shell

2022年09月12日 16:35

- 添加read调用，获取键盘输入

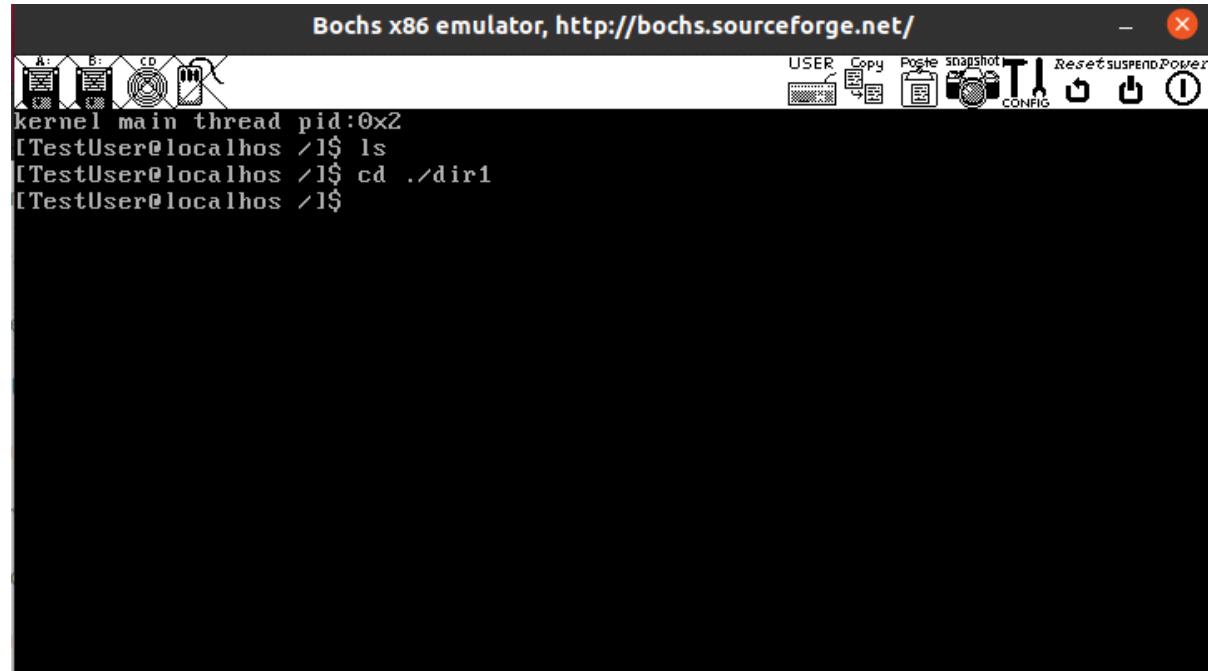
```
/**  
 * @brief  
 * 从文件描述符fd中读取count个字节到buf, 支持标准输入流  
 * 成功返回读取的字节数, 失败返回-1  
 */  
int32_t sys_read(int32_t fd, void* buf, uint32_t count) {  
    int32_t ret = -1;  
    if(fd<0) {  
        dbg_printf("sys_read: fd error\n");  
        return -1;  
    }  
    assert(buf!=NULL);  
    if(fd == stdin_no) {  
        char* buffer = buf;  
        int32_t bytes_read = 0;  
        while(bytes_read<(int32_t)count) {  
            *buffer = ioq_getchar(&keyboard_iobuffer);  
            buffer++;  
            bytes_read++;  
        }  
        ret = (bytes_read==0)?-1:bytes_read;  
    } else {  
        uint32_t _fd = fd_idx2global_ftable_idx(fd);  
        ret = file_read(&file_table[_fd], buf, count);  
    }  
    return ret;  
}
```

- 添加clear调用

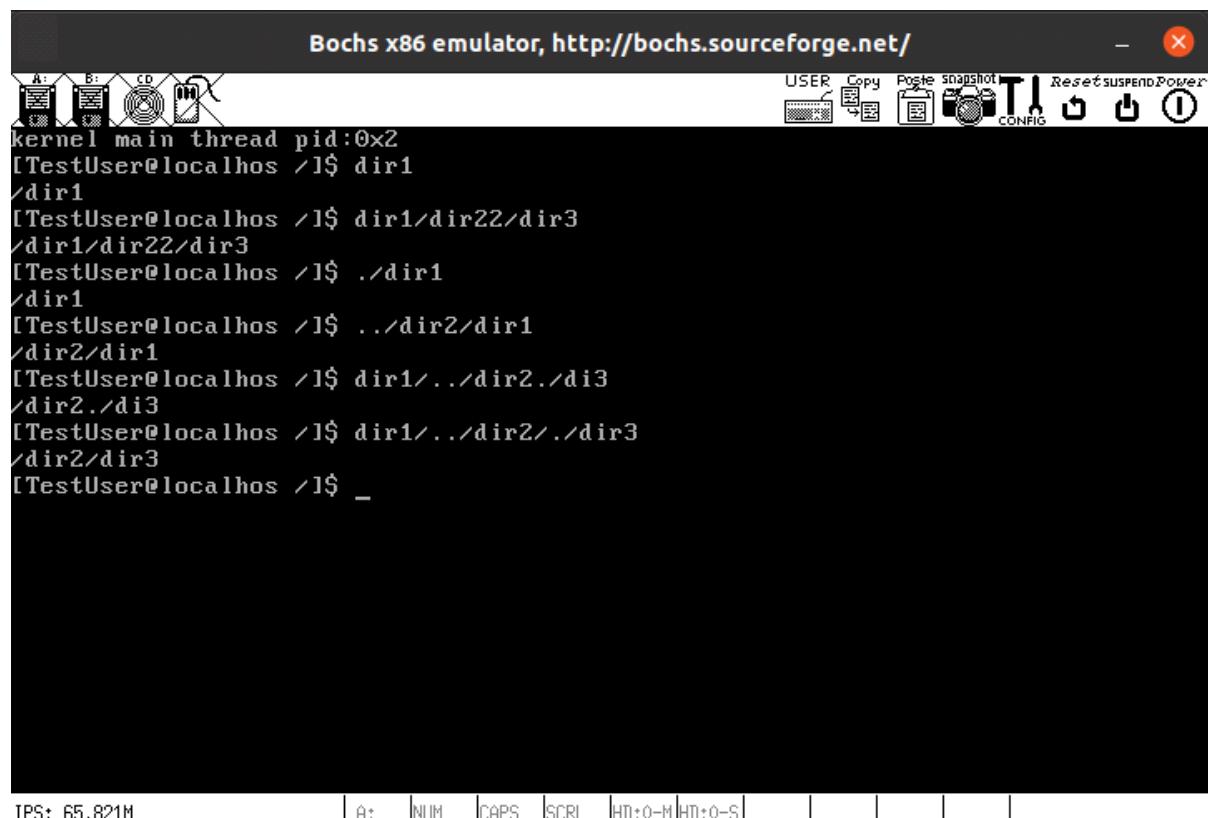
```
;清除屏幕内容  
cls_screen:  
    pushad  
    ;给gs赋值, 用户进程提权后gs=0  
    mov ax, SELECTOR_VIDEO  
    mov gs, ax  
  
    mov ebx, 0  
    mov ecx, 80*25  
  
    ;使用黑底白字覆盖全屏幕  
.cls:  
    mov word [gs:ebx], 0x0720  
    add ebx, 2  
    loop .cls  
    mov ebx, 0  
    call .set_cursor  
  
    popad  
    ret
```

- 实现Shell

简单shell,只有输出没有解析功能:



- 实现路径解析：



实现系统调用命令：

Bochs x86 emulator, http://bochs.sourceforge.net/

```
[TestUser@localhos /]$ ls -l
total: 72
d 0 72 .
d 0 72 ..
d 1 72 dir1
[TestUser@localhos /]$ mkdir dir2
*****sys_mkdir*****
NO sync memory block bitmap value:F
block idx:0x4
sync_dir_entry lba:0x2AA
make dir ok: dir inofo:
dir_name:/dir2,dir_isize:48,dir_no:3
*****sys_end*****
/dir2
[TestUser@localhos /]$ ls -l
total: 96
d 0 96 .
d 0 96 ..
d 1 72 dir1
d 3 48 dir2
[TestUser@localhos /]$ rmdir dir2
/dir2
[TestUser@localhos /]$ ls
. .. dir1
[TestUser@localhos /]$
```

IPS: 68.100M A: NUM CAPS SCRL HD:0-M HD:0-S

Bochs x86 emulator, http://bochs.sourceforge.net/

```
dir_name:/dir2,dir_isize:48,dir_no:3
*****sys_end*****
/dir2
[TestUser@localhos /]$ ls -l
total: 96
d 0 96 .
d 0 96 ..
d 1 72 dir1
d 3 48 dir2
[TestUser@localhos /]$ rmdir dir2
/dir2
[TestUser@localhos /]$ ls
. .. dir1
[TestUser@localhos /]$ cd /dir1
[TestUser@localhos /dir1]$ ls
. .. dir2
[TestUser@localhos /dir1]$ pwd
[dir1]
[TestUser@localhos /dir1]$ ps
PID      PPID      STAT      TICKS      COMMAND
1        NULL      READY      0xA90      init
2        NULL      READY      0xA4C      main_thread
3        NULL      BLOCKED   0x0        idle
4        1         RUNNING   0x0        init_fork
[TestUser@localhos /dir1]$
```

IPS: 66.029M A: NUM CAPS SCRL HD:0-M HD:0-S

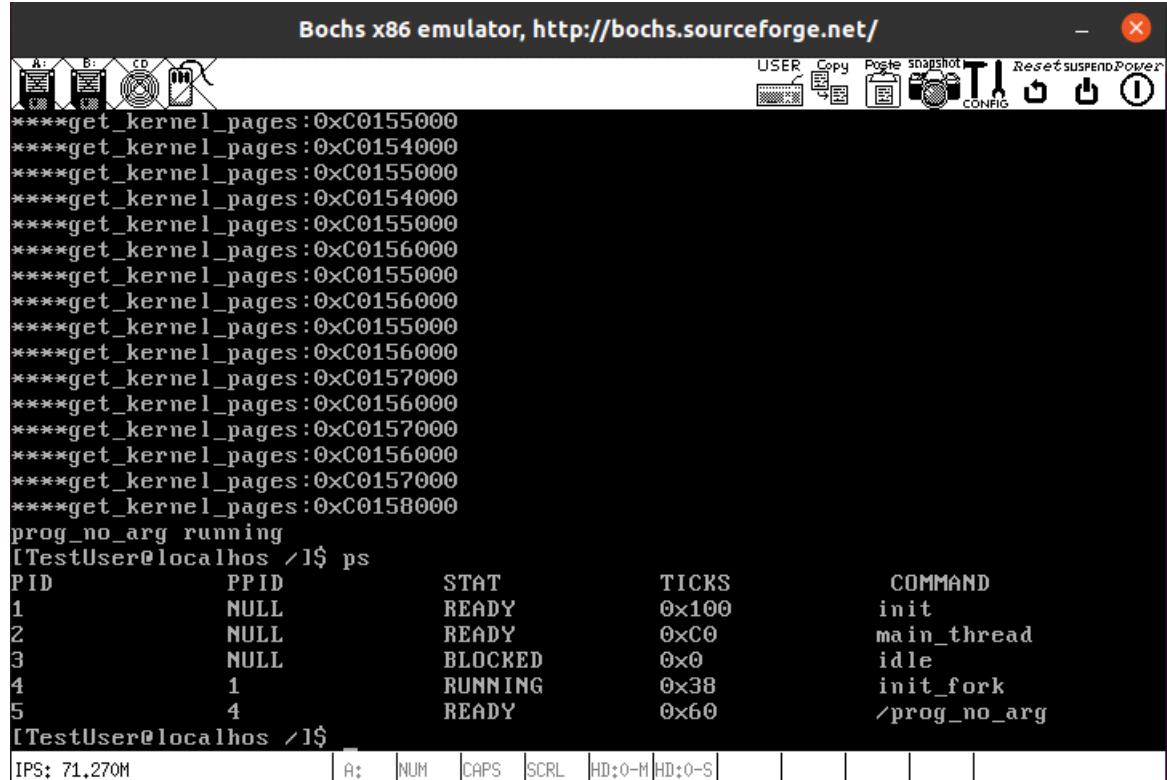
实现exec

2022年9月19日 16:34

实现无参的加载磁盘上的进程：

1. 使用fork创建一个子进程
2. 从磁盘中读取elf文件到内存，进行elf文件特征值比较
3. 比较成功后，将elf中可加载的段，载入到内核内存中一定要载入到内核不能像书中一样直接覆盖到虚拟地址
4. 将内核内存中的值复制到elf文件指定的虚拟地址（通过判断目标虚拟地址的pde和pte是否有效，来决定是否需要申请内存）
5. 构建一个中断返回栈，使eip指向程序入口点，最后从中断返回。
6. 注意：9.28再次修正

成功截图(屏蔽了一些函数的细节输出):



The screenshot shows the Bochs x86 emulator interface. The terminal window displays the following output:

```
Bochs x86 emulator, http://bochs.sourceforge.net/
A: B: CD: 
*****get_kernel_pages:0xC0155000
*****get_kernel_pages:0xC0154000
*****get_kernel_pages:0xC0155000
*****get_kernel_pages:0xC0154000
*****get_kernel_pages:0xC0155000
*****get_kernel_pages:0xC0156000
*****get_kernel_pages:0xC0155000
*****get_kernel_pages:0xC0156000
*****get_kernel_pages:0xC0155000
*****get_kernel_pages:0xC0156000
*****get_kernel_pages:0xC0157000
*****get_kernel_pages:0xC0156000
*****get_kernel_pages:0xC0157000
*****get_kernel_pages:0xC0158000
prog_no_arg running
[TestUser@localhos /]$ ps
 PID      PPID      STAT      TICKS      COMMAND
 1        NULL      READY      0x100      init
 2        NULL      READY      0xC0      main_thread
 3        NULL      BLOCKED    0x0       idle
 4        1         RUNNING    0x38      init_fork
 5        4         READY      0x60      /prog_no_arg
[TestUser@localhos /]$
IPS: 71,270M
```

The terminal prompt is [TestUser@localhos /]\$. The command ps is run to list processes. The output shows five processes: init (PID 1), main_thread (PID 2), idle (PID 3), init_fork (PID 4), and /prog_no_arg (PID 5). The /prog_no_arg process is currently running.

实现有参的加载磁盘上的进程：

Bochs x86 emulator, http://bochs.sourceforge.net/

```
[TestUser@localhos /]$ ls
. .. prog_arg
[TestUser@localhos /]$ prog_arg hello
prog_no_arg running
argv[0]:/prog_arg
argv[1]:hello
father process,child pid:6
shou dao chan shu:hello
next arg://hello
open file failed,file does not exists
[TestUser@localhos /]$ _
```

IPS: 69.167M

A: NUM CAPS SCRL HD:0-M HD:0-S

实现wait和exit

2022年10月8日 10:55

exit:

exit是由子进程调用的，表面上功能是使子进程结束运行并传递返回值给内核，本质上是内核在幕后会将进程除pcb以外的所有资源都回收。

wait:

wait是由父进程调用的，表面上功能是使附近阻塞自己，直到子进程调用exit结束运行，然后获得子进程的返回值。本质上是内核在幕后将子进程的返回值传递给父进程并会唤醒父进程，并将子进程的pcb回收

孤儿进程：

子进程未执行完毕，父进程先执行完毕并退出，这种情况将由init进程成为新的父进程负责回收子进程的pcb资源。

僵尸进程：

子进程执行完毕，但是父进程没有调用wait，导致pcb不能回收。

因为是子功能所以，没有截图，只附上代码

```
#include "wait_exit.h"
#include "../thread/thread.h"
#include "../lib/kernel/memory.h"
#include "../fs/fs.h"
#include "../lib/stdio.h"
#include "../lib/kernel/stdio-kernel.h"
/***
 * @brief
 * 释放用户进程资源
 * 包括以下资源
 * 页表的物理页
 * 虚拟内存池的物理页
 * 关闭打开的文件
 */
static void release_prog_resource(task_struct* release_thread) {
    uint32_t* pgdir_vaddr = release_thread->pgdir;
    uint16_t user_pde_nr = 768, pde_idx = 0;
    uint32_t pde = 0;
    uint32_t* v_pde_ptr = NULL;
    uint16_t user_pte_nr = 1024, pte_idx = 0;
    uint32_t pte = 0;
    uint32_t* v_pte_ptr = NULL;
    uint32_t* first_pte_vaddr_in_pde = NULL;
    uint32_t pg_phys_addr = 0;
    //回收页表中用户空间的页框
    while(pde_idx < user_pde_nr) {
        v_pde_ptr = pgdir_vaddr + pde_idx;
        pde = *v_pde_ptr;
        if(pde & 1) {
            first_pte_vaddr_in_pde = get_pte_ptr(pde_idx * 0x400000);
            pte_idx = 0;
            while(pte_idx < user_pte_nr) {
                v_pte_ptr = first_pte_vaddr_in_pde + pte_idx;
                pte = *v_pte_ptr;
                if(pte & 1) {
                    pg_phys_addr = pte & 0xfffff000;
                    free_a_phys_page(pg_phys_addr);
                }
                pte_idx++;
            }
            pg_phys_addr = pde & 0xfffff000;
            free_a_phys_page(pg_phys_addr);
        }
    }
}
```

```

        pde_idx++;
    }
    //回收用户虚拟地址池所占的物理内存
    uint32_t bitmap_pg_cnt = release_thread->userprog_vaddr.bitmap_vaddr.bitmap_byte_len /PG_SIZE;
    uint8_t* user_vaddr_pool_bitmap = release_thread->userprog_vaddr.bitmap_vaddr.bits;
    mfree_page(PF_KERNEL, user_vaddr_pool_bitmap, bitmap_pg_cnt);

    //关闭进程打开的文件
    uint8_t fd_idx = 3;
    while(fd_idx < MAX_FILES_OPEN_PER_PROC) {
        if(release_thread->fd_table[fd_idx] != -1) {
            sys_close(fd_idx);
        }
        fd_idx++;
    }
}
/** 
 * @brief
 * list回调
 * 查找pelem的pid是否是ppid
 * 成功返回true, 失败返回false
 */
static bool find_child(list_elem* pelem, int32_t ppid) {
    task_struct* pthread = struct_entry(pelem, task_struct, all_list_tag);
    if(ppid == pthread->pid) return true;
    return false;
}
/** 
 * @brief
 * list回调
 * 查找父进程pid为ppid并且状态为挂起的任务
 */
static bool find_hanging_child(list_elem* pelem, int32_t ppid) {
    task_struct* pthread = struct_entry(pelem, task_struct, all_list_tag);
    if(pthread->parent_pid == ppid && pthread->status == TASK_HANGING) {
        return true;
    }
    return false;
}
/** 
 * @brief
 * list回调
 * 所有父进程为pid的全部过继给init任务
 */
static bool init_adopt_a_child(list_elem* pelem, int32_t pid) {
    task_struct* pthread = struct_entry(pelem, task_struct, all_list_tag);
    if(pthread->parent_pid == pid) {
        pthread->parent_pid = 1;
    }
    return false;
}
/** 
 * @brief
 * 子进程结束时调用, status退出状态
 */
void sys_exit(int32_t status) {
    task_struct* child_thread = get_running_thread_pcb();
    child_thread->exit_status = status;
    if(child_thread->parent_pid == -1) {
        printf("current task no parent task\n");
    }
    // 1. 将当前进程的所有子进程过继给init
    list_traversal(&thread_all_list, (functionc*)init_adopt_a_child, (void*)child_thread->pid);
    // 2. 回收进程child_thread的资源
    release_prog_resource(child_thread);
    // 3. 唤醒正在等待的父进程
    task_struct* parent_thread = pid2thread(child_thread->parent_pid);
    if(parent_thread!=NULL && (parent_thread->status == TASK_WAITING)) {
        thread_unlock(parent_thread);
    }
}

```

```

        }

    // 4. 挂起自身, 等待父进程回收pcb
    thread_block(TASK_HANGING);
}

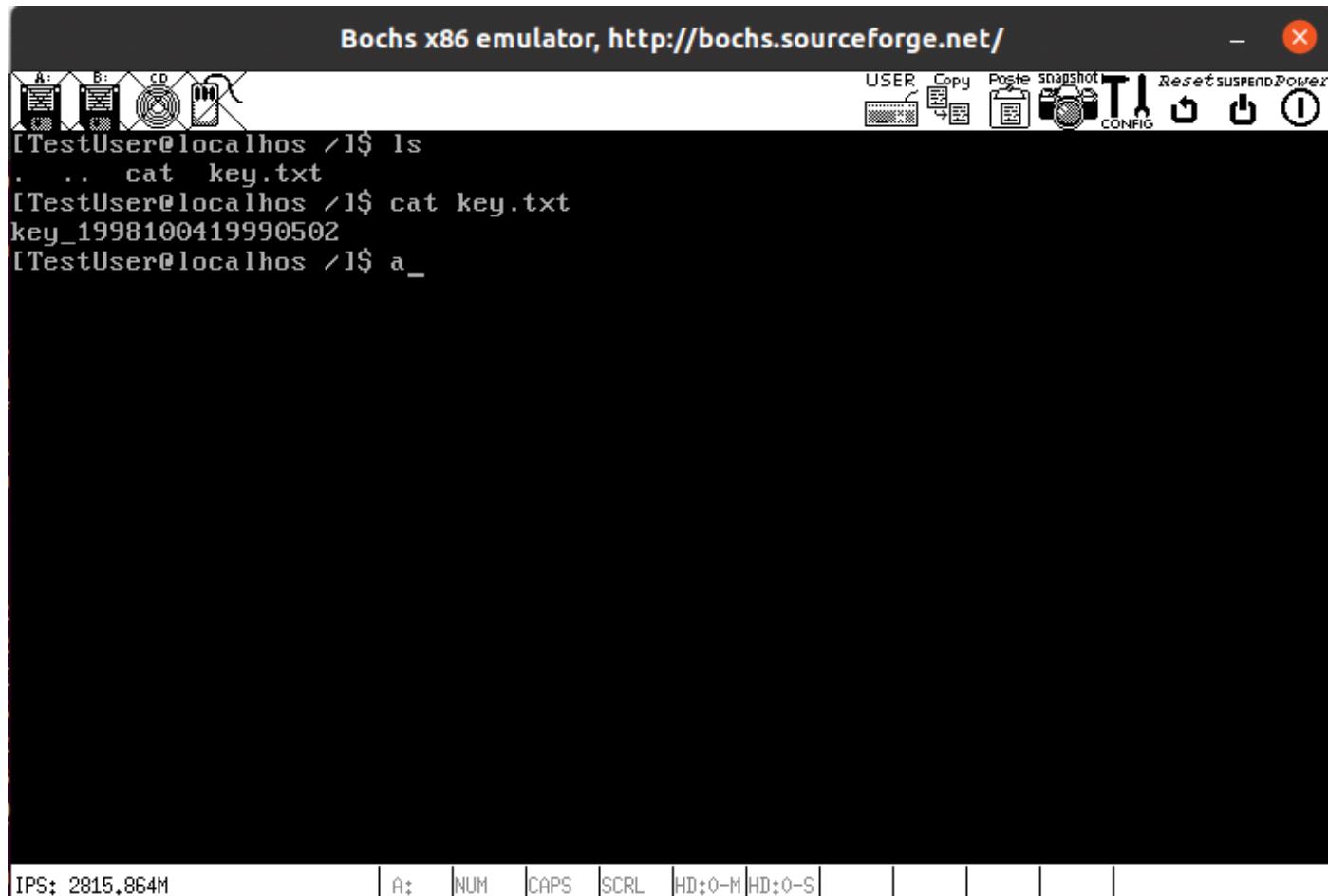
/***
 * @brief
 * 等待子进程调用exit, 将子进程的退出状态保存到status中, status可以为NULL
 * 成功返回子进程的pid, 失败返回-1
 */
pid_t sys_wait(int32_t* status) {
    task_struct* parnet_thread = get_running_thread_pcb();
    while(true) {
        // 1. 子进程调用exit会将自身挂起, 所以直接查找挂起的任务
        list_elem* child_elem = list_traversal(&thread_all_list, (functionc*)find_hanging_child, (void*)parnet_thread->pid);
        if(child_elem!= NULL) {
            task_struct* child_thread = struct_entry(child_elem, task_struct, all_list_tag);
            //保存子进程退出状态到参数
            if(status!=NULL) {
                *status = child_thread->exit_status;
            }
            uint16_t child_pid = child_thread->pid;
            // 2. 回收进程剩余资源
            thread_exit(child_thread, false);
            //dbg_printf("child thread exit, pid %d\n", child_pid);
            return child_pid;
        }
        // 3. 检查是否存在还在执行的子进程
        child_elem = list_traversal(&thread_all_list, (functionc*)find_child, (void*)parnet_thread->pid);
        //即不存在子进程, 那么wait返回失败
        if(child_elem == NULL) return -1;
        thread_block(TASK_WAITING);
    }
}

```

实现cat

2022年10月10日 13:41

1. 更新c运行库模块
2. 组合wait和exit功能到cat中
3. 读取磁盘文件
4. 通过write输出到屏幕上



The screenshot shows a terminal window in the Bochs x86 emulator. The title bar reads "Bochs x86 emulator, http://bochs.sourceforge.net/". The terminal window displays the following commands and output:

```
[TestUser@localhos /]$ ls
. . . cat key.txt
[TestUser@localhos /]$ cat key.txt
key_1998100419990502
[TestUser@localhos /]$ a_
```

The bottom status bar shows memory usage: "IPS: 2815.864M".

实现管道

2022年10月10日 14:04

最后一章了，加油冲冲冲

管道的原理：

管道通常被多个进程共享，而且存在于内存之中，因此将其安排在内核内存中较为合适。

使用环形缓冲区为管道提供合适的缓冲区大小。

管道有两段，一端从管道中读入数据，另一端往管道中写入数据。通过文件描述符的方式进行操作，因此创建管道就是内核为其返回了用于读取管道缓冲区的文件描述符，一个读一个写。通常为一个长度2的文件描述符数组。

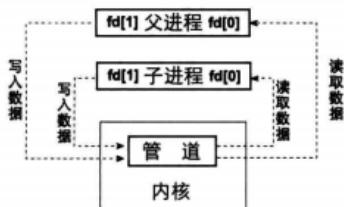
管道分为两种：

匿名管道：

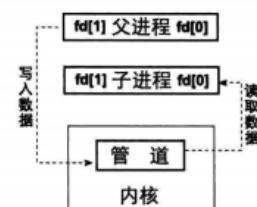
只能通过内核为其返回的文件描述符来访问，只对父子进程可见。

命令管道：

提供了一个路径名与之关联，以FIFO文件的形式存储于文件系统中，能够实现任何两个进程之间通信



▲图 15-18 父子进程与管道



▲图 15-19 父子进程间通常的管道操作

管道的设计：

部分效仿LINUX的VFS操作方式

将管道的文件描述放置在全局文件表中，这样各个进程都可以通过文件描述符打开管道，实现进程间的读写操作。

文件结构体：

```
typedef struct _file{  
    uint32_t fd_pos; //定义为管道的打开次数  
    uint32_t fd_flag;//专属于管道的flags(0xFFFF)  
    inode* fd_inode;//内核缓冲区  
}file;
```

管道的实现：

两次操作管道，申请和读取写入正常，全局文件表释放正常。（有错误单词，请别太在意）

The screenshot shows a terminal window in the Bochs x86 emulator. The output displays the creation of pipes, sending messages between processes, and exiting the program. The terminal window has a title bar 'Bochs x86 emulator, http://bochs.sourceforge.net/' and a menu bar with options like USER, Copy, Paste, snapshot, CONFIG, Reset, Suspend, Power.

```
Bochs x86 emulator, http://bochs.sourceforge.net/  
A: B: CD:  
ioqueue_init success  
pipe create local fd:3  
pipe create local fd:4  
local_fd:3,global_fd:3,file_table base:0xC0018EE0  
local_fd:4,global_fd:3,file_table base:0xC0018EE0  
father,my pid is 2  
child,my pid 5  
child,fathed send message:A message from the parent process  
  
father exit  
pipe mfree  
[TestUser@localhost ~]$ prog_pipe  
pipe create global fd:3  
ioqueue_init success  
pipe create local fd:3  
pipe create local fd:4  
local_fd:3,global_fd:3,file_table base:0xC0018EE0  
local_fd:4,global_fd:3,file_table base:0xC0018EE0  
father,my pid is 2  
child,my pid 5  
child,fathed send message:A message from the parent process  
  
father exit  
pipe mfree  
[TestUser@localhost ~]$
```

```
CH0114: Father->Send message to message from the parent process
```

```
father exit  
pipe mfree  
[TestUser@localhos /]$\nIPS: 2938.367M
```

A: NUM CAPS SCRL HD:0-M HD:0-S

在shell中支持管道:

遇到的bug还是挺多的

已解决的bug:

sys_write函数会持续追加数据到文件末尾，导致环形缓冲区挂起shell线程。

cat函数先后逻辑错误

管道释放后导致shell后面的内存空间也被释放，其实不是释放的问题，而是不应该在shell循环中使用break，导致进程

成功截图:

Bochs x86 emulator, http://bochs.sourceforge.net/

```
[TestUser@localhos /]$\nA: B: CD\nUSER Copy Paste snapshot T1 Reset Suspend Power\n[TestUser@localhos /]$\nls\n..\ncat key.txt\n[TestUser@localhos /]$\ncat key.txt|cat\npipe command\npipe create global fd:3\npipe buffer:0xC013C000\nioqueue_init success\npipe create local fd:3\npipe create local fd:4\nlocal_fd:3,global_fd:3,file_table base:0xC0018EE0\nlocal_fd:4,global_fd:3,file_table base:0xC0018EE0\nlocal_fd:3,global_fd:3,file_table base:0xC0018EE0\nlocal_fd:4,global_fd:3,file_table base:0xC0018EE0\nkey_1998100419990502global_fd:3,pipe mfree buffer 0xC013C000\n[TestUser@localhos /]$\nIPS: 2961.712M
```

A: NUM CAPS SCRL HD:0-M HD:0-S

help命令:

Bochs x86 emulator, http://bochs.sourceforge.net/

```
kernel main thread pid:0x2\n[TestUser@localhos /]$\nhelp\nbuildin commands:\n    ls: Displays files and folders in the current directory\n    cd: Changing the current directory\n    mkdir:Create a directory\n    rmdir:Remove a directory\n    pwd:Display current directory\n    ps:Displaying Process Information\n    clear:Clear the screen\n    exit:Exit the Shell\nshortcuts key:\n    ctrl+l:Clear the screen\n    ctrl+u:Clear input\nhint:\n    Commands and arguments are separated by Spaces\n    Pipe separator for "!"\n[TestUser@localhos /]$\nIPS: 3011.951M
```

A: NUM CAPS SCRL HD:0-M HD:0-S

内存布局(实时更新)

2022年4月27日 10:18

到目前为止的内存布局情况:

总共是4MB的大小=4096KB	
EBDA	0x9fc00-0x9ffff size:1K
kernel esp	0x9f000
可用空间	0x7e00-0x9fbff
	0x10_00000 1MB
kernel_file_image	0x70000
Mbr	0x7c00-0x7DFF size :512B
	0x7000
Kernel.c	0xc0001500 = 物理地址 0x1500
Kernel Load	0x900
	0x500 - 0x8ff
BIOS数据区	400-4ff
中断向量表	0x0-3ff size:1K