tips

> 要将os看做一个 `.so`

> BIOS 加载 bootloader, bootloader 再加载 kernel

> 因为操作系统内核本身就是通过自旋循环常驻内存的

> 中断是唯一进入内核方式

> 页表的创建与物理内存的分配是完全不同的两件事，建立页表建立的只是映射关系，与内存是否被分配无关

> 用户态进入内核态的开销是巨大的，是通过软中断进入的。中断与系统调用是都有可能进入的（**开销在于栈的切换与寄存器的保存与恢复**)。当执行完对 中断/异常/系统调用 打断的处理后，最后会执行一个 iret 指令。**在执行此指令之前，CPU的当前栈指针 esp 一定指向上次产生 中断/异常/系统调用 时 CPU 保存的被打断的指令地址 CS 和 EIP，iret 指令会根据 ESP 所指的保存的址 CS 和 EIP 恢复到上次被打断的地方继续执行**

1. int 80指令引发一个中断陷入内核
2. 当前用户态进程通过 TS 寄存器找到 TSS 任务状态段，进而确定内核栈的位置，内核栈这个时候一定是空白的
3. 开始将寄存器压栈（全部压在内核态的，**全部是中断前的寄存器值**，硬件负责压栈的部分在进入中断帧前就已经变成了内核态的寄存器值，如cs, ss等等，软件负责压栈的寄存器有软件负责修改一些），顺序如下
    - ss (用户态进入核态会压)
    - esp (用户态进入核态会压这个值)
    - eflags
    - cs
    - eip
    - errorCode
    - trapno
    - ds
    - es
    - fs
    - gs
4. 压栈完成之后，跳转到中断处理程序
5. 如果是中断进来的，那么可以有中断的嵌套，但是绝对没有进程的切换，所以中断服务例程处理完毕之后，开始弹栈，开始恢复寄存器，并且由内核态返回用户态（就根据保存在内核栈顶的ss，

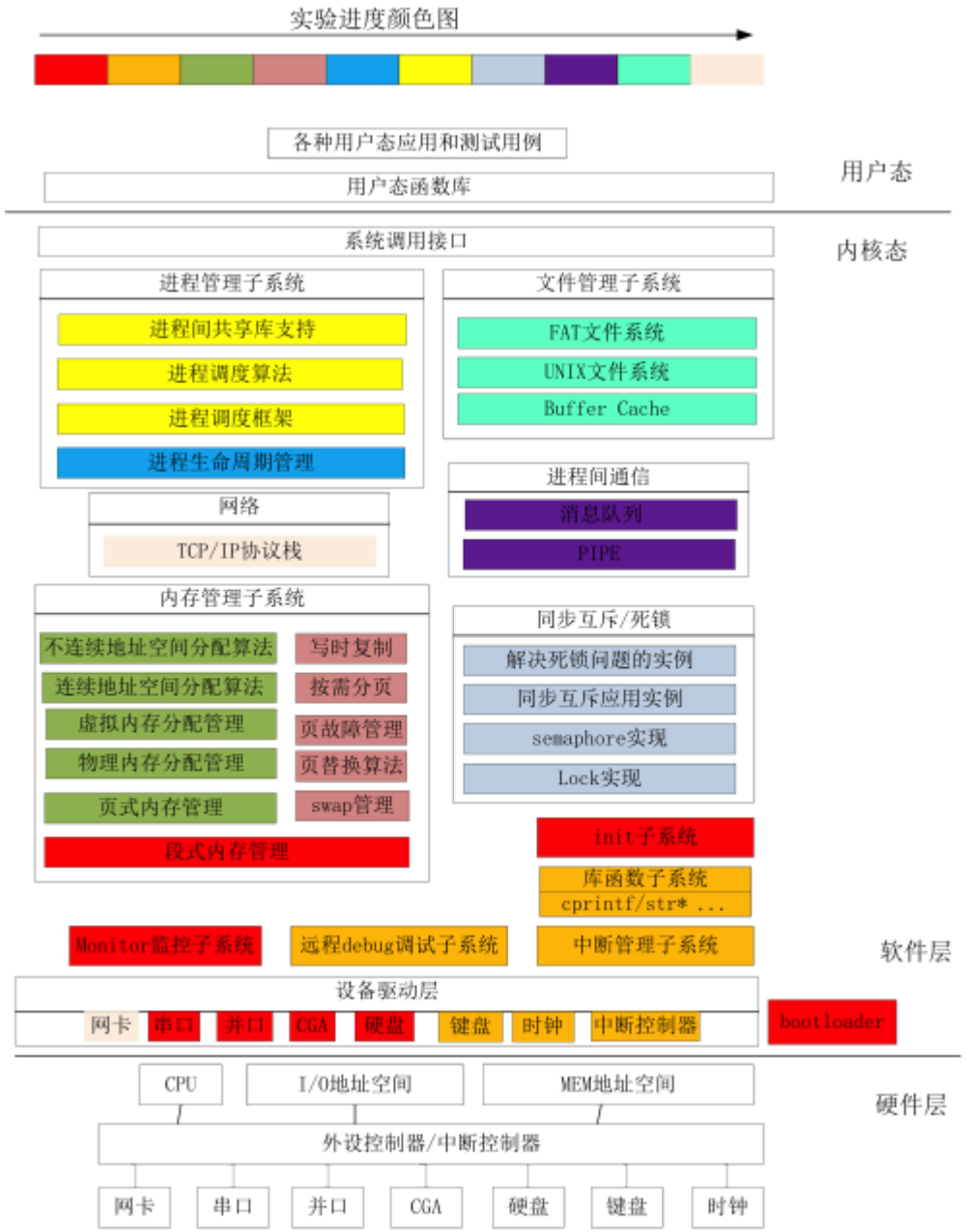
esp以及cs, eip, efalgs的值)

- **最后的 iret，将中断之前的 cs, eip, cflags 等抛出来去恢复对应的寄存器，并且跳过去。如果是用户态进入的，则 ss,esp 也要抛出去恢复**

6. 如果是系统调用进来的，内核可以抢占，且系统调用本身自己可以把自己 hang 起来，这个时候发生进程的切换，现场是被保留的，被调度回来依然可以在对应的内核栈上去执行对应的操作

> mmu 支持多种页转换的大小, 比如 4KB 2MB 1GB 等等

os实验



那我们准备如何一步一步来实现ucore呢？根据一个操作系统的设计实现过程，我们可以有如下的实验步骤：

1. 启动操作系统的 bootloader，用于了解操作系统启动前的状态和要做的准备工作，了解运行操作系统的硬件支持，操作系统如何加载到内存中，理解两类中断--**外设中断**，**陷阱中断**等
2. 物理内存管理子系统，用于理解x86分段/分页模式，了解操作系统如何管理物理内存
3. 虚拟内存管理子系统，通过页表机制和换入换出（swap）机制，以及中断-故障中断、缺页故障处理等，实现基于页的内存替换算法
4. 内核线程子系统，用于了解如何创建相对与用户进程更加简单的内核态线程，如果对内核线程进行动态管理等
5. 用户进程管理子系统，用于了解用户态进程创建、执行、切换和结束的动态管理过程，了解在用户态通过系统调用得到内核态的内核服务的过
6. 处理器调度子系统，用于理解操作系统的调度过程和调度算法
7. 同步互斥与进程间通信子系统，了解进程间如何进行信息交换和共享，并了解同步互斥的具体实现以及对系统性能的影响，研究死锁产生的原因，以及如何避免死锁
8. 文件系统，了解文件系统的具体实现，与进程管理等的关系，了解缓存对操作系统IO访问的性能改进，了解虚拟文件系统（VFS）、buffer cache和disk driver之间的关系

```
target: prerequisites
    command
```

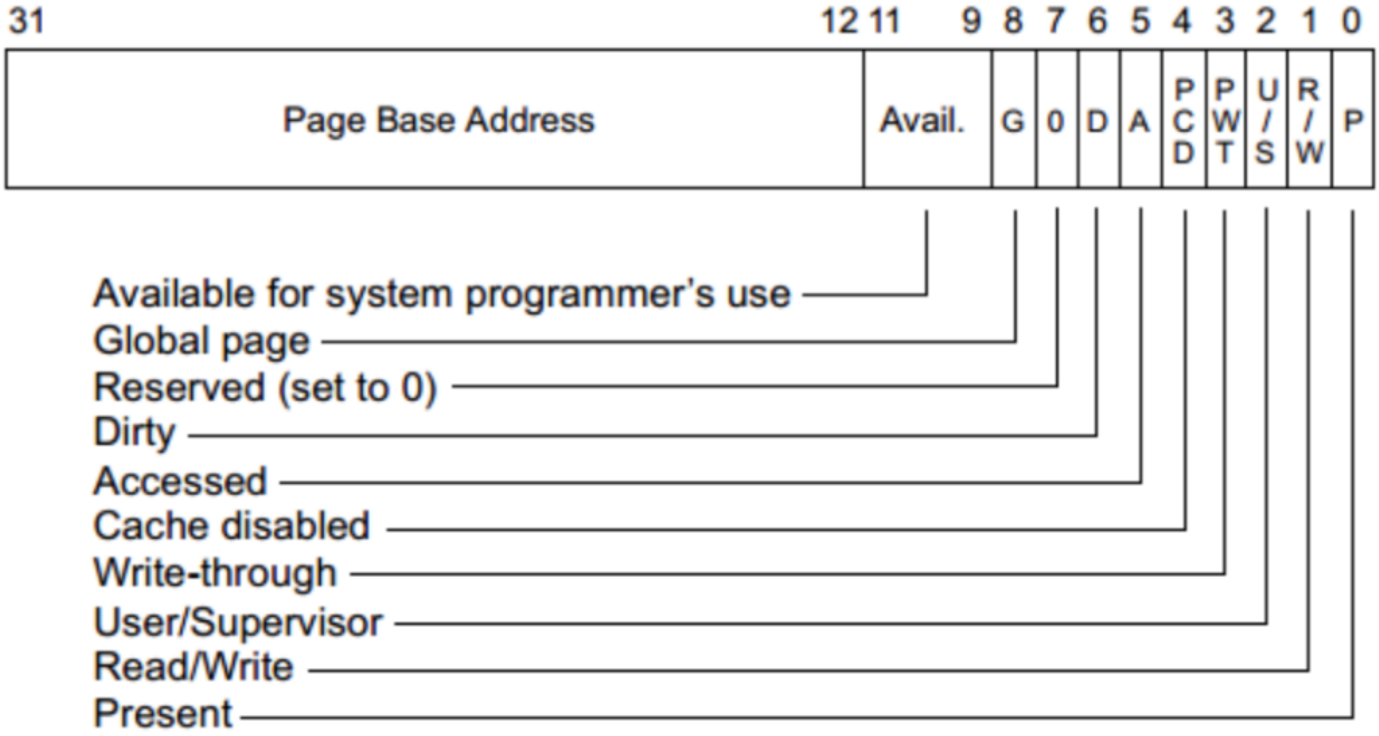
- target也就是一个目标文件，可以是object file，也可以是执行文件。还可以是一个标签（label）。prerequisites就是，要生成那个target所需要的文件或是目标。command也就是make需要执行的命令（任意的shell命令）。 这是一个文件的依赖关系，也就是说，target这一个或多个的目标文件依赖于prerequisites中的文件，其生成规则定义在 command中。如果prerequisites中有一个以上的文件比target文件要新，那么command所定义的命令就会被执行。这就是makefile的规则。也就是makefile中最核心的内容。
- Makefile编译c/c++的核心思路
    - 由顶层递归进入子目录，当进入到一个目录的最底层时，开始gcc，将该层的所有.o文件打包成build-in.o，返回上一层目录再递归进入子目录，最后开始编译顶层的.c文件，最后将顶层的.o文件和顶层每个子目录的build-in.o链接成我们的目标文件，**多叉树的前序遍历**

内核的初始化过程

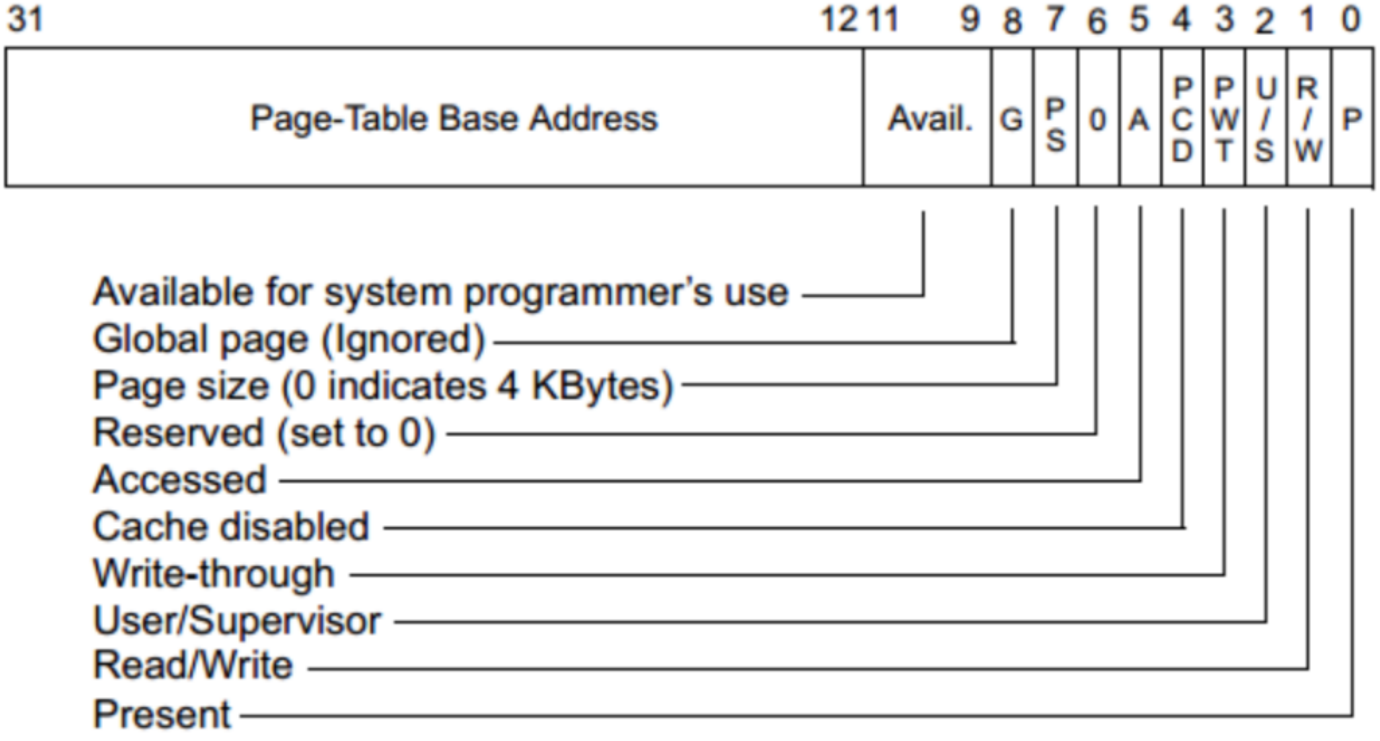
- 先看一下内核的elf文件格式

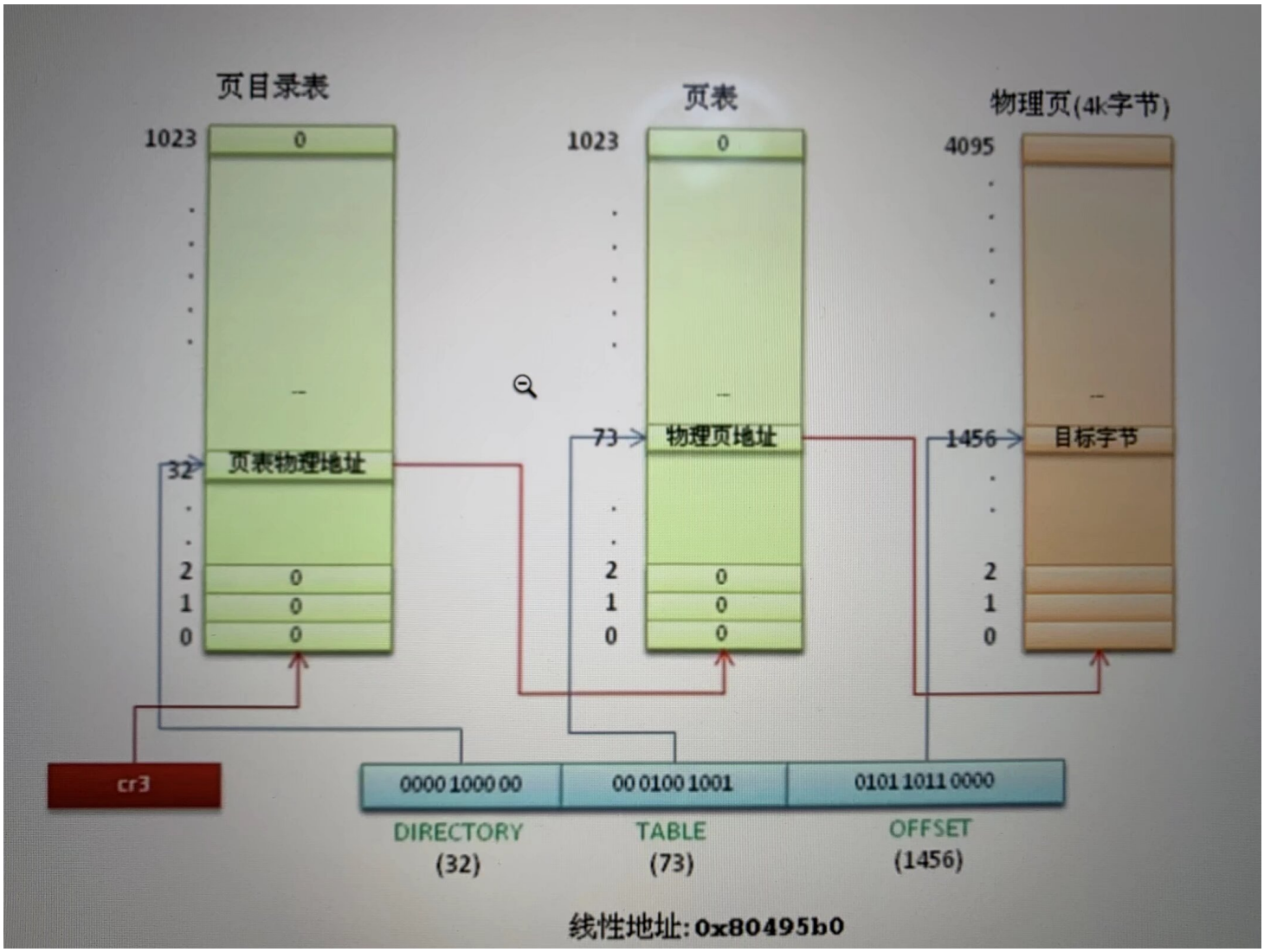
```
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's' complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           Intel 80386
  Version:                           0x1
  Entry point address:               0xc0100000  # kerne的入口会由这个地方开始执行，rip的位置是在这
  Start of program headers:          52 (bytes into file)
  Start of section headers:          122636 (bytes into file)
  Flags:                             0x0
  Size of this header:               52 (bytes)
  Size of program headers:           32 (bytes)
  Number of program headers:         3
  Size of section headers:           40 (bytes)
  Number of section headers:         12
  Section header string table index: 11

# Elf file type is EXEC (Executable file)
# Entry point 0xc0100000
# There are 3 program headers, starting at offset 52

# virtual这个是程序的虚拟地址，也叫做逻辑地址
Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  LOAD           0x001000 0xc0100000 0xc0100000 0x14a60 0x14a60 R E 0x1000
  LOAD           0x016000 0xc0115000 0xc0115000 0x05000 0x05f28 RW  0x1000
  GNU_STACK      0x000000 0x00000000 0x00000000 0x00000 0x00000 RWE 0x10

# 看一下段寄存器，这个只是程序员可见的部分
cs 0x8
ss 0x16
ds 0x16
es 0x16
fs 0x16
gs 0x16
```

Page-Table Entry (4-KByte Page)

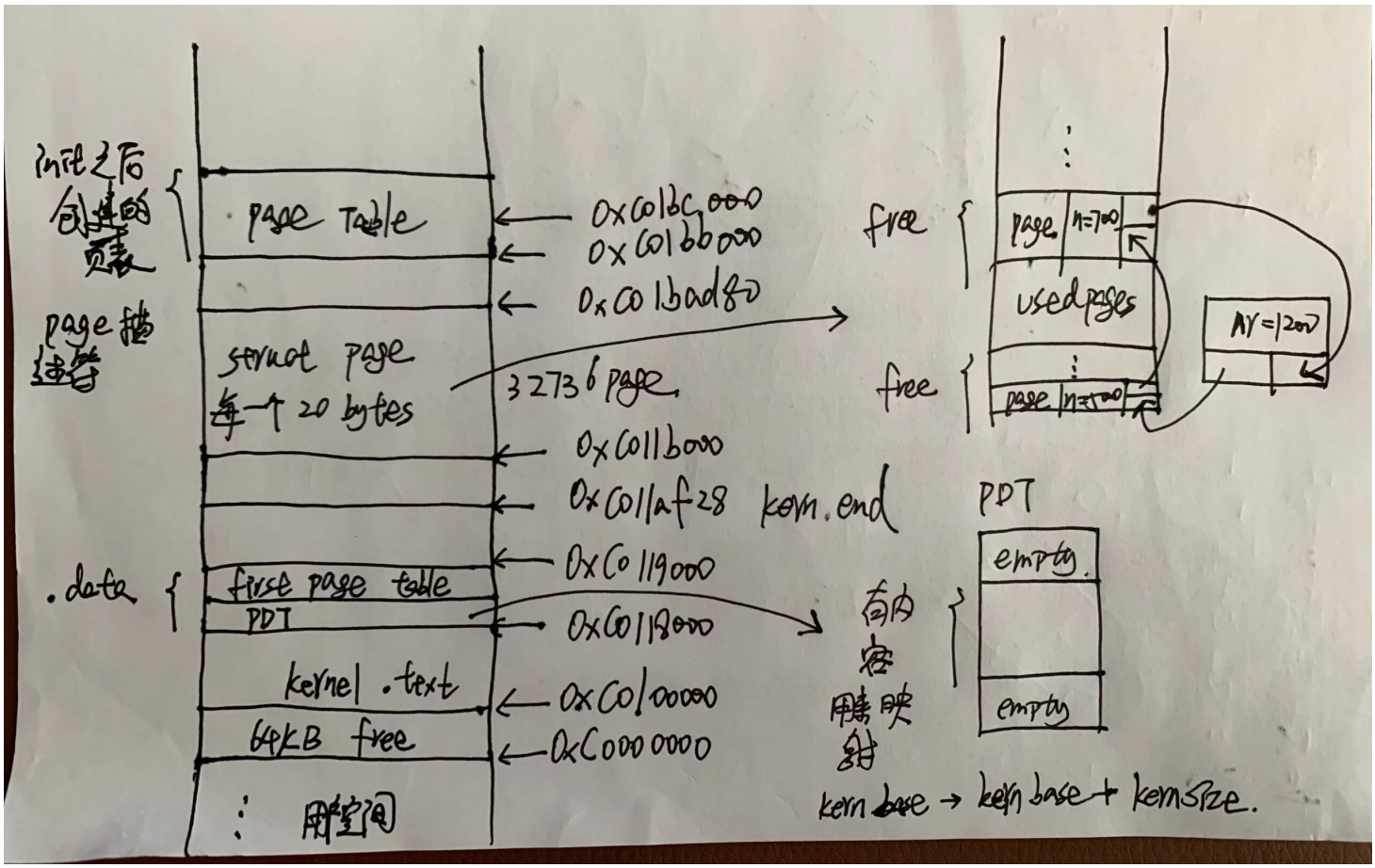


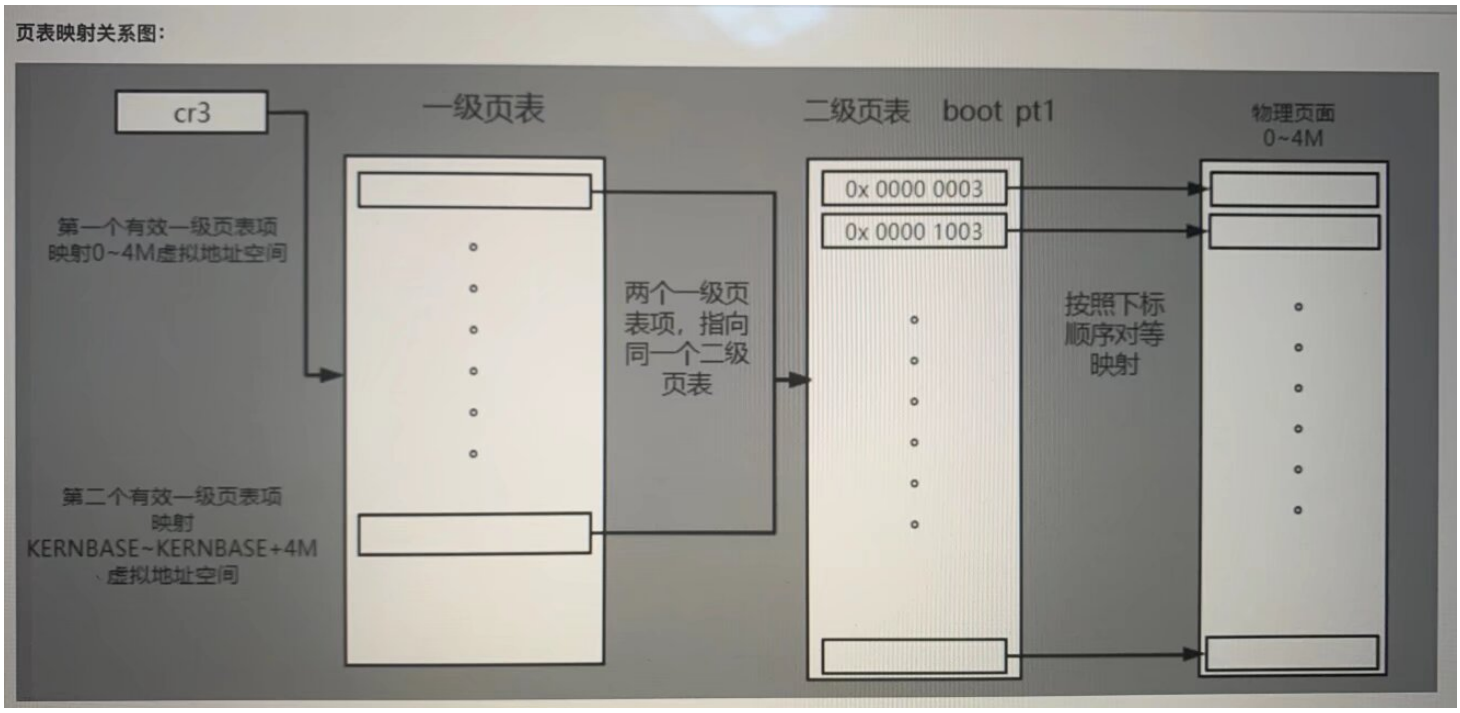
Page-Directory Entry (4-KByte Page Table)

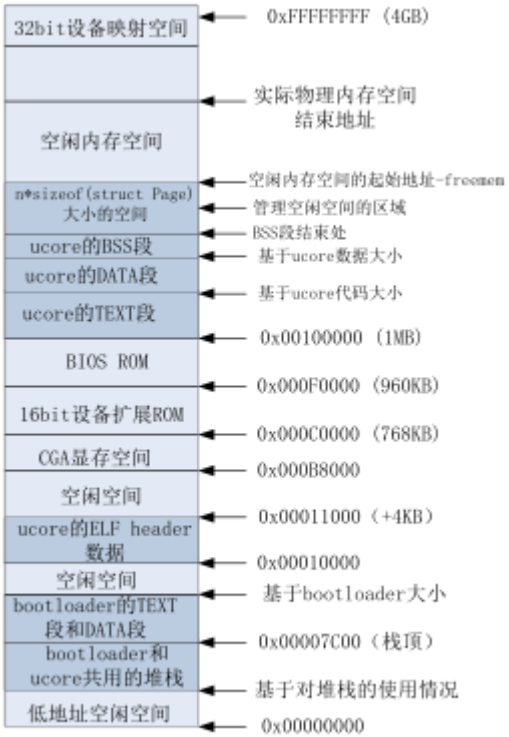




- cr3 包括页表，包括页目录表**存放的都是物理地址**


init之后创建的页表
page Table
0xc016c000
0xc0160000
0xc015bdf8
page描述符
struct page
每个20 bytes
32736 page.
0xc011b000
0xc011af28
kern.end
0xc0119000
.data
first page table
PDT
0xc0118000
kernel .text
0xc0100000
64KB free
0xc0000000
用空间
free
page n=700
used pages
NY=1207
free
page n=300
PDT
empty
empty
有内容映射
kern base → kern base + kern size.


页表映射关系图:
cr3
一级页表
二级页表 boot pt1
物理页面
0~4M
第一个有效一级页表项
映射0~4M虚拟地址空间
两个一级页表项，指向同一个二级页表
0x 0000 0003
0x 0000 1003
按照下标顺序对等映射
第二个有效一级页表项
映射
KERNBASE~KERNBASE+4M
虚拟地址空间


32bit设备映射空间
0xFFFFFFFF (4GB)
实际物理内存空间
结束地址
空闲内存空间
空闲内存空间的起始地址-freemem
n*sizeof(struct Page)
大小的空间
管理空闲空间的区域
BSS段结束处
ucore的BSS段
基于ucore数据大小
ucore的DATA段
基于ucore代码大小
ucore的TEXT段
0x00100000 (1MB)
BIOS ROM
0x000F0000 (960KB)
16bit设备扩展ROM
0x000C0000 (768KB)
CGA显存空间
0x000B8000
空闲空间
0x00011000 (+4KB)
ucore的ELF header
数据
0x00010000
空闲空间
基于bootloader大小
bootloader的TEXT
段和DATA段
0x00007C00 (栈顶)
bootloader和
ucore共用的堆栈
基于对堆栈的使用情况
低地址空闲空间
0x00000000

```
# 首先是BootLoader的一个汇编代码部分

# Start the CPU: switch to 32-bit protected mode, jump into C
# The BIOS loads this code from the first sector of the hard disk into
# memory at physical address 0x7c00 and starts executing in real mode
# with %cs=0 %eip=0x7c00

# 这四句话相当于定义变量，理解为define
.set PROT_MODE_CSEG,        0x8                     # kernel code segment selector
.set PROT_MODE_DSEG,        0x10                    # kernel data segment selector
.set CR0_PE_ON,             0x1                     # protected mode enable flag
.set SMAP,                  0x534d4150

# start address should be 0:0x7c00, in real mode, the beginning address of the running bootloader
.globl start  # start in 0x7c00
start:
.code16                                             # Assemble for 16-bit mode，16位模式下运行的指令
    cli                                             # Disable interrupts，关中断
    cld                                             # String operations increment，以自增的方式去运行指令

    # Set up the important data segment registers (DS, ES, SS). 段寄存器清零
    xorw %ax, %ax                                   # Segment number zero
    movw %ax, %ds                                   # -> Data Segment
    movw %ax, %es                                   # -> Extra Segment
    movw %ax, %ss                                   # -> Stack Segment

    ...     # Enable A20:  # 32位地址线可以用了

    ...     # probe_memory，嗅探内存，e820会被映射到物理地址0x8000上，实地址模式的时候被映射过去的，实际上是保存了一个物

    # GDT表长的是这个样子，基地址均为0，是对等映射，计算机的问题都可以加一层抽象来处理，GDT表其实就是这样的一层
        SEG_NULLASM  # null seg，第一个是空的
        SEG_ASM(STA_X|STA_R, 0x0, 0xffffffff)  # code seg for bootloader and kernel,代码段，可以读可以执行，段的基址
        SEG_ASM(STA_W, 0x0, 0xffffffff)  # data seg for bootloader and kernel，数据段，有写权限

    # Switch from real to protected mode, using a bootstrap GDT
    # and segment translation that makes virtual addresses
    # identical to physical addresses, so that the
    # effective memory map does not change during the switch.
    lgdt gdtdesc            # 加载gdt表，实际上在设置GDTR寄存器
    movl %cr0, %eax
    orl $CR0_PE_ON, %eax    # 将CR0寄存器的PE位置1代表进入了32位的保护模式
    movl %eax, %cr0

    # Jump to next instruction, but in 32-bit code segment.
    # Switches processor into 32-bit mode.
    ljmp $PROT_MODE_CSEG, $protcseg

.code32                                             # Assemble for 32-bit mode，正式进入32位模式
protcseg:
    # Set up the protected-mode data segment registers
```

```
    # CS寄存器设置为0x8，其他段寄存器设置为0x10，其它都是与数据相关的段寄存器
    movw $PROT_MODE_DSEG, %ax                       # Our data segment selector
    movw %ax, %ds                                   # -> DS: Data Segment
    movw %ax, %es                                   # -> ES: Extra Segment
    movw %ax, %fs                                   # -> FS
    movw %ax, %gs                                   # -> GS
    movw %ax, %ss                                   # -> SS: Stack Segment

    # Set up the stack pointer and call into C. The stack region is from 0--start(0x7c00)
    # 为bootmain函数的执行设置堆栈
    movl $0x0, %ebp
    movl $start, %esp
    call bootmain  # 在调用函数bootmain的时候，第一件事是push %rbp，mov %rbp，%rsp，会将start的值给到ebp寄存器中，然后

    # If bootmain returns (it shouldn't), loop
spin:
    jmp spin

.data
# Bootstrap GDT
.p2align 2                                          # force 4 byte alignment
gdt:
    SEG_NULLASM                                     # null seg
    SEG_ASM(STA_X|STA_R, 0x0, 0xffffffff)           # code seg for bootloader and kernel
    SEG_ASM(STA_W, 0x0, 0xffffffff)                 # data seg for bootloader and kernel

# bootmain执行完毕之后呢（加载磁盘数据的过程是同步且阻塞的过程，反正也没其它事情做的说）
# 需要加载 kernel，但是kernel的入口是entry.S的汇编代码部分
# 即在entry.S中，BootLoader执行完毕之后跳转到了kern_entry中执行，而不是之前的kern_init
# 在这里，它会加载一个写死的页表映射关系，进入这里的时候
# 需要倒着看，有些变量的定义要看清楚的

    # .space size, fill
        # 这个指令保留size个字节的空间，每个字节值为fill，如果逗号和fill被省略，则fill默认为0
        # .space 1024 表示保留1024个字节的空间，并且每个字节的值为0

    # .long expression
        # 这个表达式表示任意一节中的一个或多个表达式，例如
            # gdt_descr:
            # .long 10
            # 变量gdt_descr的值置为10

#define REALLOC(x) (x - KERNBASE)  # x-0xc0000000

.text  # 这里是只读的指令
.globl kern_entry  # 在kernel.ld中被设置成了0xc0100000，即虚拟地址的高位置
kern_entry:
    # 所以.S中的所有变量与符号都已经相当于是映射到地址高位去了
    # 但是实际上，目前分页机制还没有开启，指令寻址依然是按照段式的对等映射，所以要把0xc0000000的偏移减去
    # cr3中保存的是页目录的起始的物理地址
```

```
    # load pa of boot pgdir, 加载一个现成的设计好了的，页表包括一个Pag directory table与一个page table
    movl $REALLOC(__boot_pgdir), %eax
    movl %eax, %cr3
    # 将PDT的物理地址给到 cr3 寄存器中

    # enable paging
    ...
    # CPU进入了分页模式

    # update eip, now, eip = 0x1.....
    # 因为有两个页表的映射机制，所以这里能够正常工作
    leal next, %eax
    # set eip = KERNBASE + 0x1.....  # 0xc0000000 + 0x1...
    jmp *%eax
next:

    # unmap va 0 ~ 4M, it's temporary mapping
    xorl %eax, %eax
    movl %eax, __boot_pgdir

    # set ebp, esp
    # 建立内核栈
    movl $0x0, %ebp
    # the kernel stack region is from bootstack -- bootstacktop,
    # the kernel stack size is KSTACKSIZE (8KB)defined in memlayout.h
    movl $bootstacktop, %esp
    # now kernel stack is ready , call the first C function
    call kern_init  # 这个时候ebp的值一直是0，call到kern_init中之后，会为kern_init函数建立stack，rbp就是现在rsp的值

# 数据段在下面，上面是代码段，可能就是顺序与平时编程不太一样所造成的不太好理解
.data  # 这里是初始化好了的数据段
.align PGSIZE
    .globl bootstack  # global定义了全局符号，其它模块也可以引用这个变量的说，初始化好的栈的起始位置
bootstack:
    .space KSTACKSIZE 8KB 预留8KB的地址空间，bootstack这个变量就是8K预留空间的起始地址
    .globl bootstacktop
bootstacktop:

# 自旋死循环(如果内核实现正确，kern_init函数将永远不会返回并执行至此。因为操作系统内核本身就是通过自旋循环常驻内存的)
spin:
    jmp spin

# kernel builtin pgdir
# an initial page directory (Page Directory Table, PDT)
# These page directory table and page table can be reused!
.section .data.pgdir
.align PGSIZE
__boot_pgdir:
.globl __boot_pgdir
    # PDT也是一个page,也是一个数组，有1024个目录项
    # map va 0 ~ 4M to pa 0 ~ 4M (temporary)
```

```
    .long REALLOC(__boot_pt1) + (PTE_P | PTE_U | PTE_W)  # PDT的第一个值就是页表的物理地址，负责映射0-4M的虚拟地址
    # 继续填充768个空的entry
    .space (KERNBASE >> PGSHIFT >> 10 << 2) - (. - __boot_pgdir) # pad to PDE of KERNBASE
    # map va KERNBASE + (0 ~ 4M) to pa 0 ~ 4M，再加一个目录项用来映射
    .long REALLOC(__boot_pt1) + (PTE_P | PTE_U | PTE_W)
    .space PGSIZE - (. - __boot_pgdir) # pad to PGSIZE

# 两个一级页表项都指向下边的二级页表项，这个页表就是指向0-4MB的物理内存的
# 相当于这里有写死的页表，需要被加载到内存中，加载位置不知道，我日，这里可以类比段表的加载
.set i, 0
__boot_pt1:
.rept 1024
    # __boot_pt1是一个存在1024个32位long数据的数组，当将其作为页表时其中每一项都代表着一个物理地址映射项
    # i为下标，每个页表项的内容为i*1024作为映射的物理页面基址并加上一些低位的属性位(PTE_P代表存在，PTE_W代表可写)
    .long i * PGSIZE + (PTE_P | PTE_W)  # *1024就是相当于左移了12位，低12位用于存放flag等数据，高20位就是物理页框号
    .set i, i + 1
.endr
```

```
# 通过调试来打印一下页表的结果
__boot_pgdir 0x118000 虚拟地址为0xc0118000  gdb: p/x boot_pgdir
0xc0118000 - 0xc118fff 4KB页目录表

当前应该有一个是存在有效数据的，那就是 0xc0118000 + 786*4B=预估在0xc0118c00处
0xc0118c00 处的数据是0x00119027 这个是__boot_pt1的物理地址（当然是前20位才是） # gdb: x/4096 0xc0118

__boot_pt1: 0xc0119027 其中高20位是 0x119000，这也是目前唯一的一个页表的物理地址，虚拟地址位0xc011900

# 来看一下页表中的项     gdb: x/4096 0xc0119000
# 其中的数据就是页表项，并且是符合预期的，如下所示
0xc0119000:     0x00000003      0x00001003      0x00002003      0x00003003
0xc0119010:     0x00004003      0x00005003      0x00006003      0x00007003
0xc0119020:     0x00008003      0x00009003      0x0000a003      0x0000b003
0xc0119030:     0x0000c003      0x0000d003      0x0000e003      0x0000f003
0xc0119040:     0x00010003      0x00011003      0x00012003      0x00013003
0xc0119050:     0x00014003      0x00015003      0x00016003      0x00017003
0xc0119060:     0x00018003      0x00019003      0x0001a003      0x0001b003
0xc0119070:     0x0001c003      0x0001d003      0x0001e003      0x0001f003
0xc0119080:     0x00020003      0x00021003      0x00022003      0x00023003
0xc0119090:     0x00024003      0x00025003      0x00026003      0x00027003
0xc01190a0:     0x00028003      0x00029003      0x0002a003      0x0002b003
0xc01190b0:     0x0002c003      0x0002d003      0x0002e003      0x0002f003
0xc01190c0:     0x00030003      0x00031003      0x00032003      0x00033003

# 以下均为在页机制下的虚拟地址，减去0xc0000000，就是已经存在的物理地址
# 内核本身一共占据了108KB的内存页
# special kernel symbols
    entry 0xc0100036  # kern_init的入口地址
    etext 0xc0105a75  # kernel text代码段的地址
    edata 0xc011a000  # kernel data数据段的地址
    end   0xc011af28  # kernel 结束位置的地址
    kernel executable memory footprint: 108KB, (end - kern_init + 1023)/1024

# 内存嗅探的结果，内存嗅探得到的地址是物理地址
# memmap->map[i].size, begin, end - 1, memmap->map[i].type
# e820map:
  memory: 0009fc00, [00000000, 0009fbff], type = 1. 连续可用物理内存 1
  memory: 00000400, [0009fc00, 0009ffff], type = 2. 16^2*4=1KB reserved
  memory: 00010000, [000f0000, 000fffff], type = 2. 64KB reserved
  memory: 07ee0000, [00100000, 07fdffff], type = 1. 连续可用物理内存 2
  memory: 00020000, [07fe0000, 07ffffff], type = 2. 128KB reserved
  memory: 00040000, [fffc0000, ffffffff], type = 2. 256KB reserved
```

```
// typedef可以声名新的类型来代替已有的类型名字
// foo_t这个符号就代表了struct foo
// 双向循环list
typedef struct foo {
    ElemType data;
    struct foo *prev;
    struct foo *next;
} foo_t;

// 还有一个比较重要的内容 TSS
// 初始化内存管理完成之后会重新加载一次GDT表，把用户态的代码段与数据段都加入到其中去，以及初始化TSS
static struct taskstate ts = {0};  # TSS是一个全局静态的变量目前
/* *
 * Global Descriptor Table:
 *
 * 用户的段与kernel的段是不一样的
 * The kernel and user segments are identical (except for the DPL). To load
 * the %ss register, the CPL must equal the DPL. Thus, we must duplicate the
 * segments for the user and the kernel. Defined as follows:
 *   - 0x0 :  unused (always faults -- for trapping NULL far pointers)
 *   - 0x8 :  kernel code segment
 *   - 0x10:  kernel data segment
 *   - 0x18:  user code segment
 *   - 0x20:  user data segment
 *   - 0x28:  defined for tss, initialized in gdt_init
 * */
static struct segdesc gdt[] = {
    SEG_NULL,
    [SEG_KTEXT] = SEG(STA_X | STA_R, 0x0, 0xFFFFFFFF, DPL_KERNEL),
    [SEG_KDATA] = SEG(STA_W, 0x0, 0xFFFFFFFF, DPL_KERNEL),
    [SEG_UTEXT] = SEG(STA_X | STA_R, 0x0, 0xFFFFFFFF, DPL_USER),
    [SEG_UDATA] = SEG(STA_W, 0x0, 0xFFFFFFFF, DPL_USER),
    [SEG_TSS]    = SEG_NULL,
};

/* temporary kernel stack */
uint8_t stack0[1024];

/* gdt_init - initialize the default GDT and TSS */
static void
gdt_init(void) {
    ts.ts_esp0 = (uint32_t)&stack0 + sizeof(stack0);
    ts.ts_ss0 = KERNEL_DS;

    gdt[SEG_TSS] = SEG16(STS_T32A, (uint32_t)&ts, sizeof(ts), DPL_KERNEL);
    gdt[SEG_TSS].sd_s = 0;  // 也要通过GDT表去定位TSS

    // reload all segment registers
    lgdt(&gdt_pd);
    // load the TSS
    ltr(GD_TSS);
```

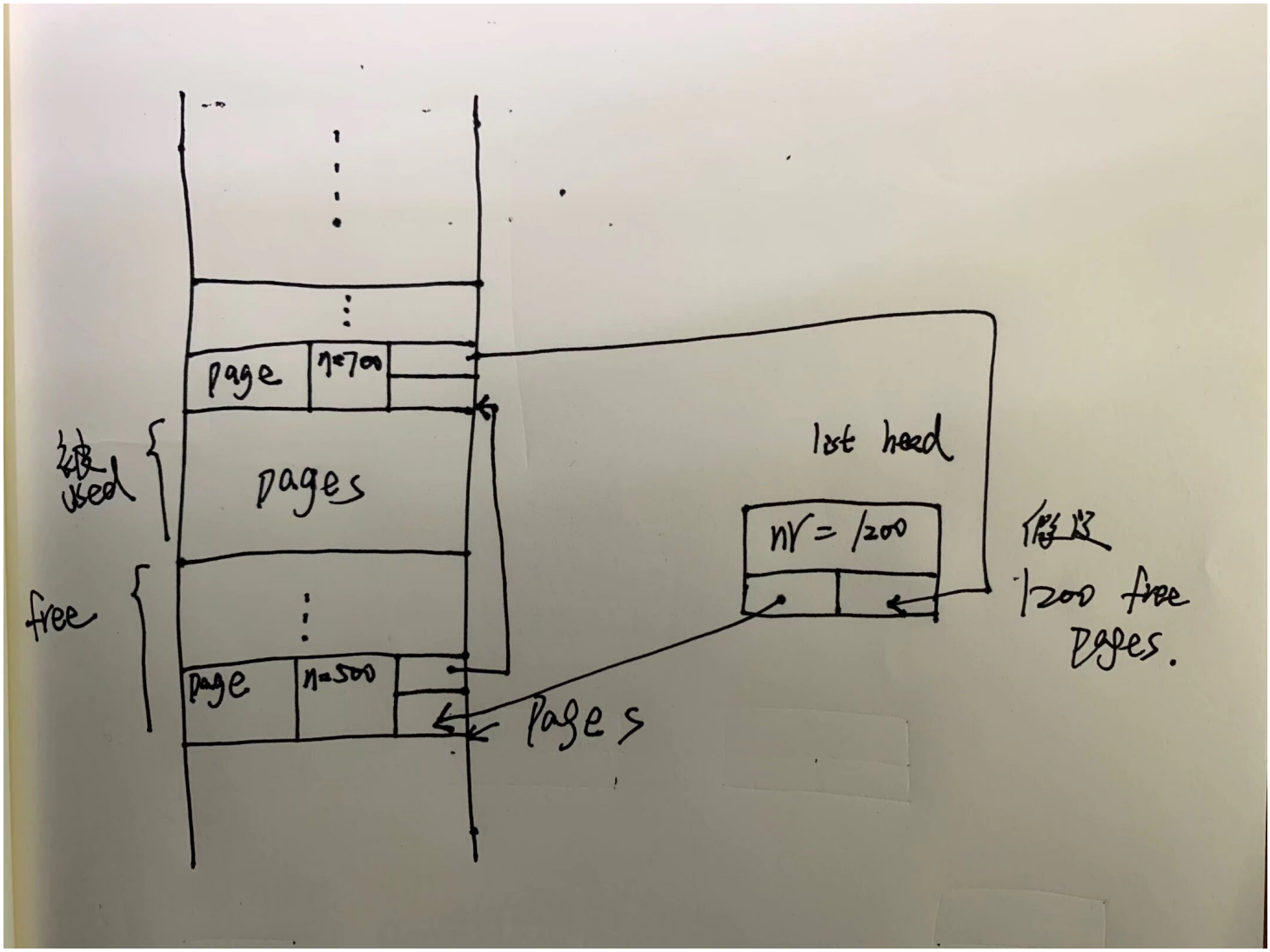
}
- 任务状态段
  - TSS是一段内存
    - 有保存内核栈的起始位置，供用户态进入内核时使用
  - 在GDT表中，task寄存器保存了当前TSS在GDT表中的下标
  - 找到GDT中的TSS描述符之后，读取段选择子 + offset就能够找到TSS的地址了

简单梳理一下这个过程吧

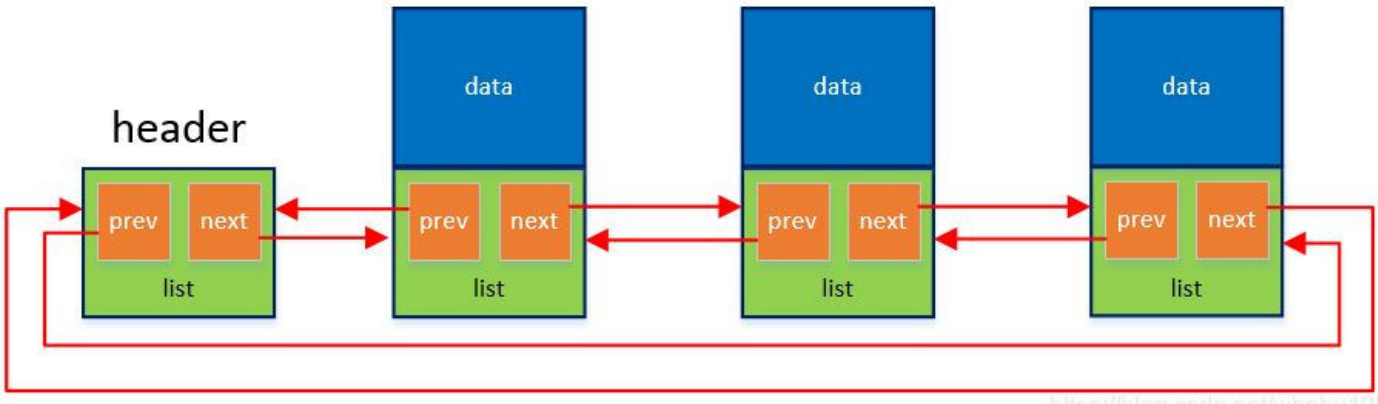
1. 实模式启动
   - 嗅探内存
     - e820会被映射到物理地址0x8000上，实地址模式的时候被映射过去的,实际上是保存了一个物理内存的布局
     - `struct e820map *memmap = (struct e820map *)(0x8000 + KERNBASE);`
   - 加载段表
2. 段表加载完毕后，cr0 寄存器 pe 位置1进入保护模式
3. 设置段寄存器并建立stack以执行BootLoader
4. BootLoader 加载kernl的img, 代码段以及数据段到内存中，并跳转到entry.S的入口处
5. kernel 的数据段中有一个页目录表，也有一个页表，固定位置写好了的
   - 实现的是将0-4M的虚拟地址映射到0-4M的物理地址，以及将kernelBase-kernelBase+4M的虚拟地址映射到0-4M的物理地址
6. 内核跳转到高地址执行
7. 初始化内存管理
   - 先管理物理内存，链表的形式。分配空间作为page管理的元数据
   - 填充页表，能够映射内核
     - ucore的物理内存明确不能超过kernsize，即不能超过896M，保证所有的物理地址空间都映射到虚拟的内核地址空间中
       - 将页目录表项补充完成（从0~4M扩充到0~KMEMSIZE），涉及到物理内存的分配，但是这里还是连续分配的
         - `virt addr = linear addr = phy addr + 0xC0000000`
         - **内核是不会缺页的**，这里物理内存是全部映射到内核的虚拟地址空间中的，**页表的映射与内存的分配完全是两会事**
           - 分配页框之后返回的页框对应的虚地址，这样访问一定是正常的
           - 在栈上分配数据，栈一开始就是设置好的
     - 有可能根本就没有对应的二级页表的情况，所以二级页表不必要一开始就分配，而是等到需要的时候再添加对应的二级页表。如果在查找二级页表项时，发现对应的二级页表不存在，则需要根据create参数的值来处理是否创建新的二级页表
       - 下面去看缺页中断的处理，就能理解这个地方了

- 重新加载GDT表
  - 使用了一个新的段表。这个新段表除了包括内核态的代码段和数据段描述符，还包括用户态的代码段和数据段描述符以及TSS（段）的描述符

8. 中断初始化
9. 自旋死循环在内核中



- 空闲页的管理如上图所示，也是最简单的管理方式



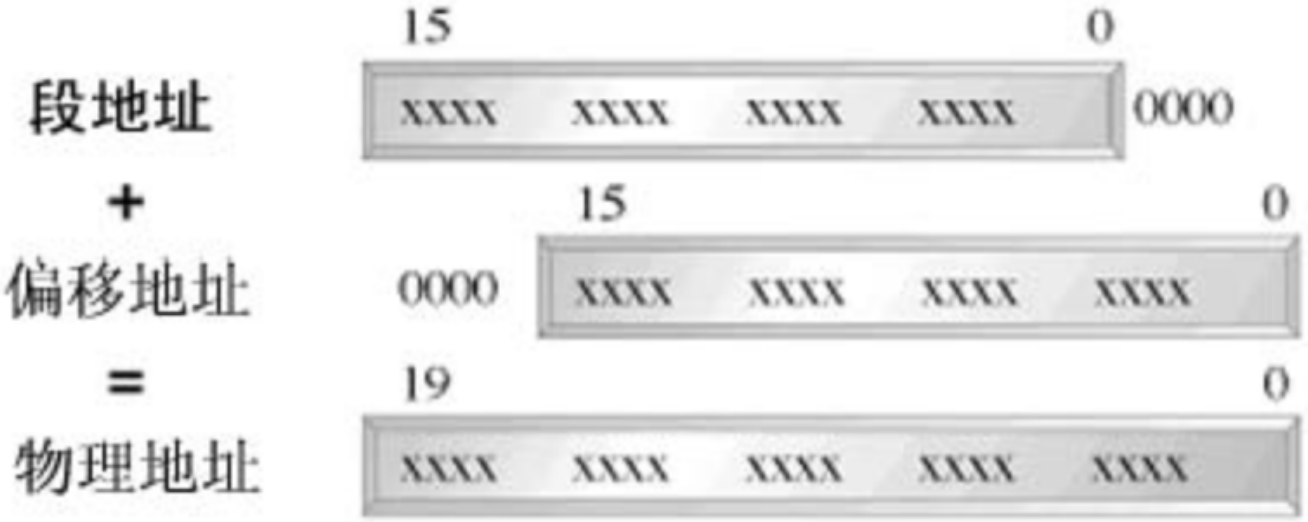
- 循环链表的示意图

上述部分一些细节的理解与说明

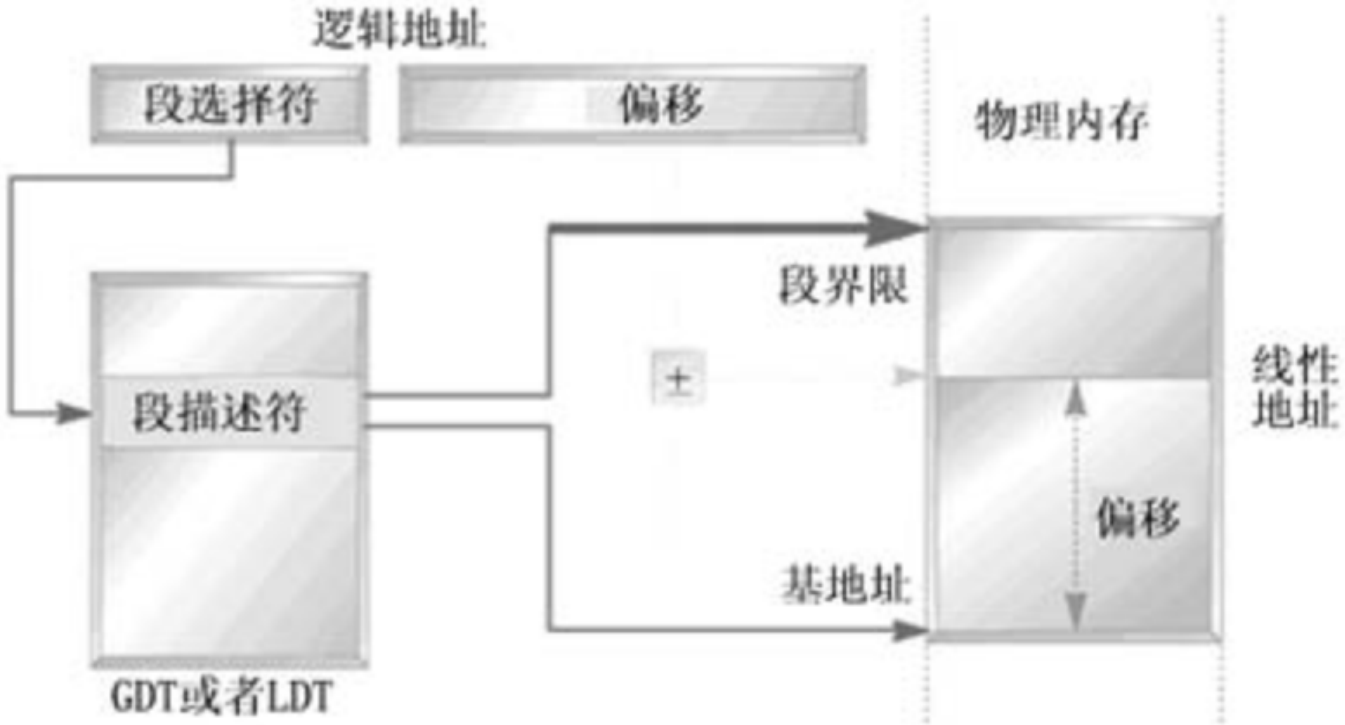
- 与磁盘交换数据的接口
  - **在boot阶段就是简单的向磁盘发读命令，等待，然后从磁盘读kernel镜像到内存**

```
waitdisk();  // 等待磁盘可用
outb(0x1F2, 1); // 向端口（这里是磁盘）发指令，告诉磁盘我要读什么，outb是向指定的端口号写数据
waitdisk();  // wait for disk to be ready，这个就是通常所说的那个耗时的状态，数据需要由磁盘存储介质达
// read a sector
insl(0x1F0, dst, SECTSIZE/4); // 由指定的端口号读取数据到地址dst，dst是内存中的地址，称为端口 IO，端口
```

实模式地址



保护模式地址

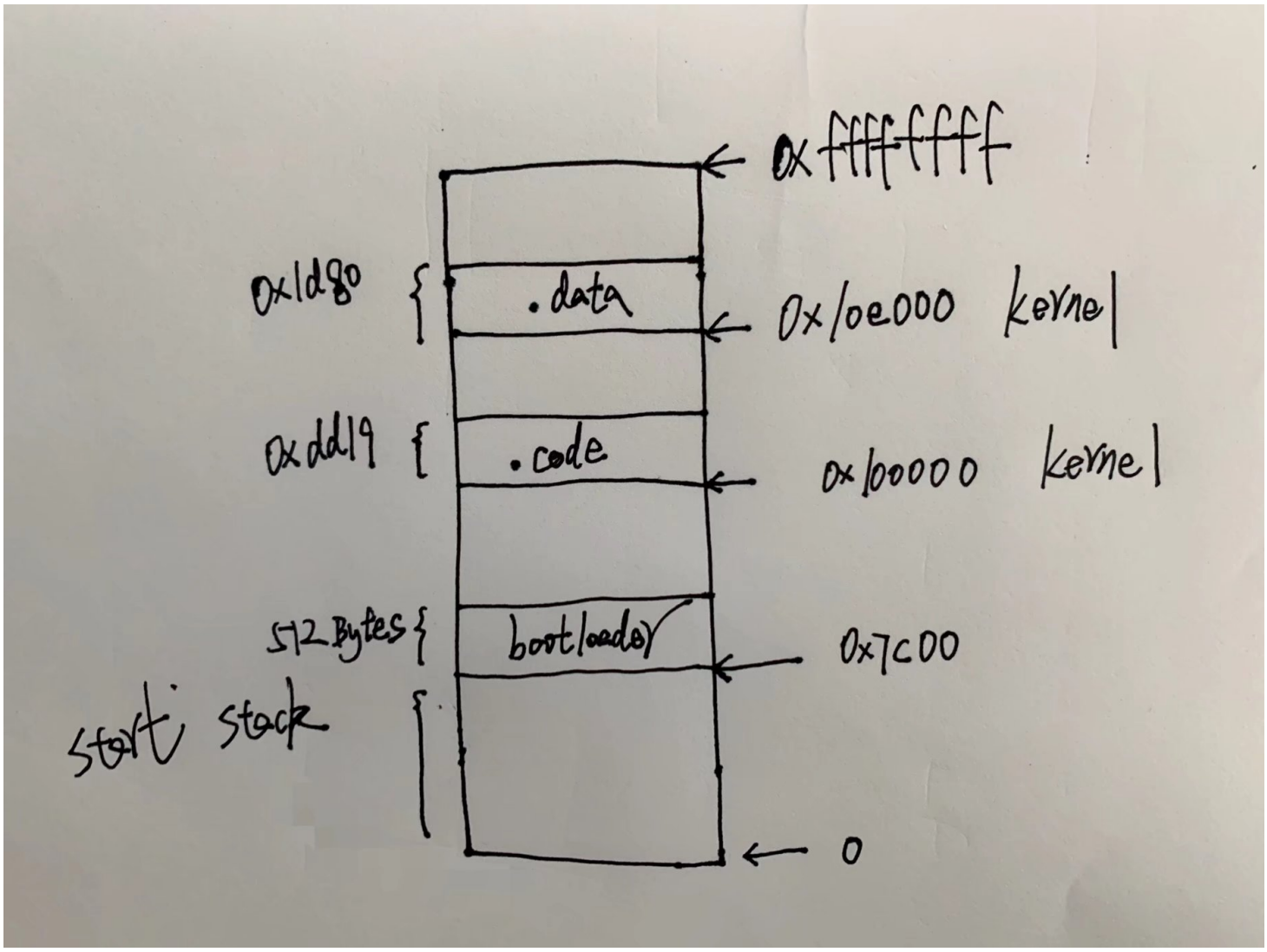


实模式与保护模式

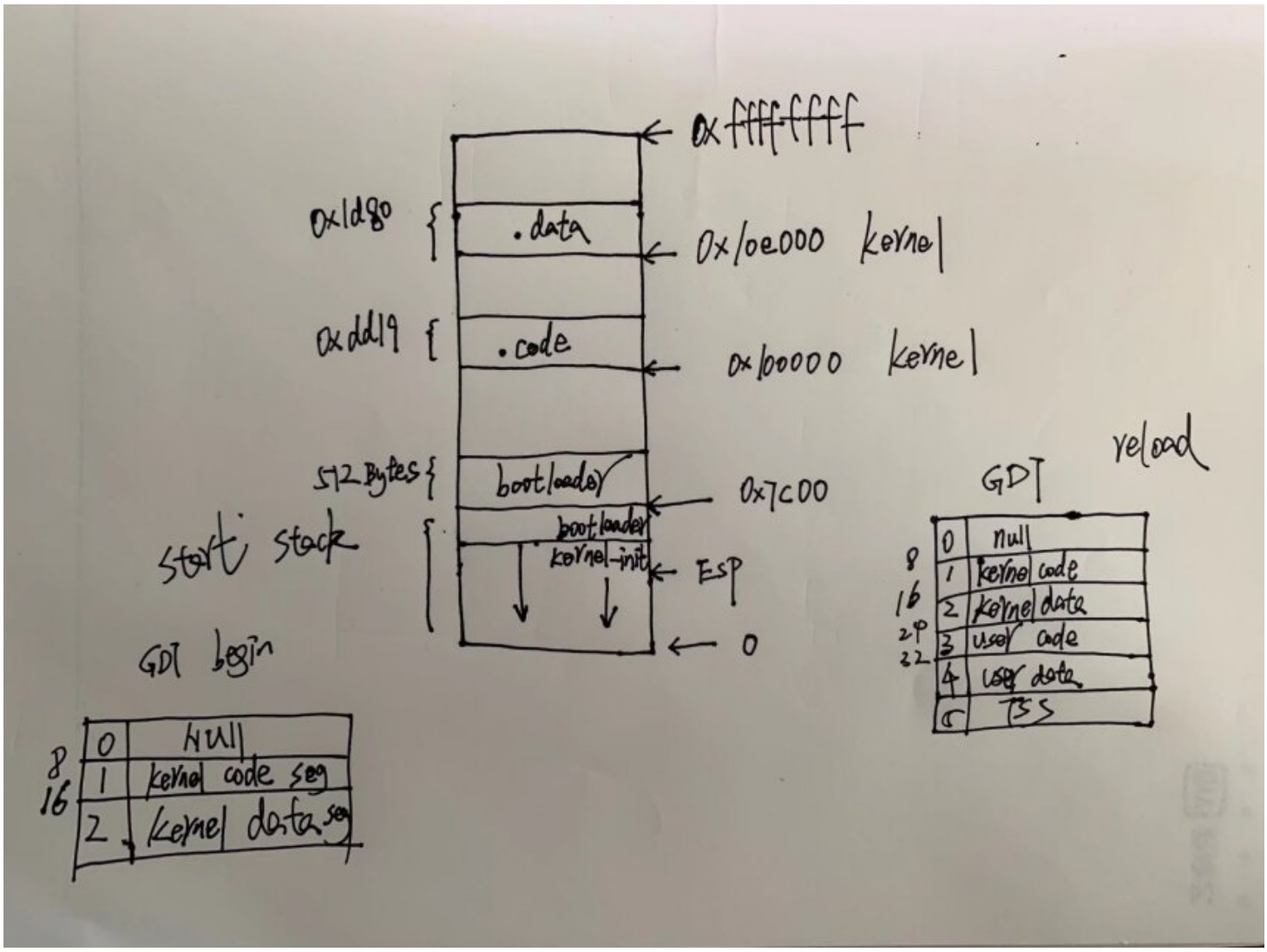
- 只确定一件事，代码中的地址应该如何用
  - **直接用程序中地址去访问物理内存**
  - **通过段页式的转换去访问物理内存呢**
- **实模式->保护模式**转变的**核心**就是
  1. 加载 GDT 表，GDT 表的基地址保存在 GDTR 中

2. **CR0寄存器的PE位置1**

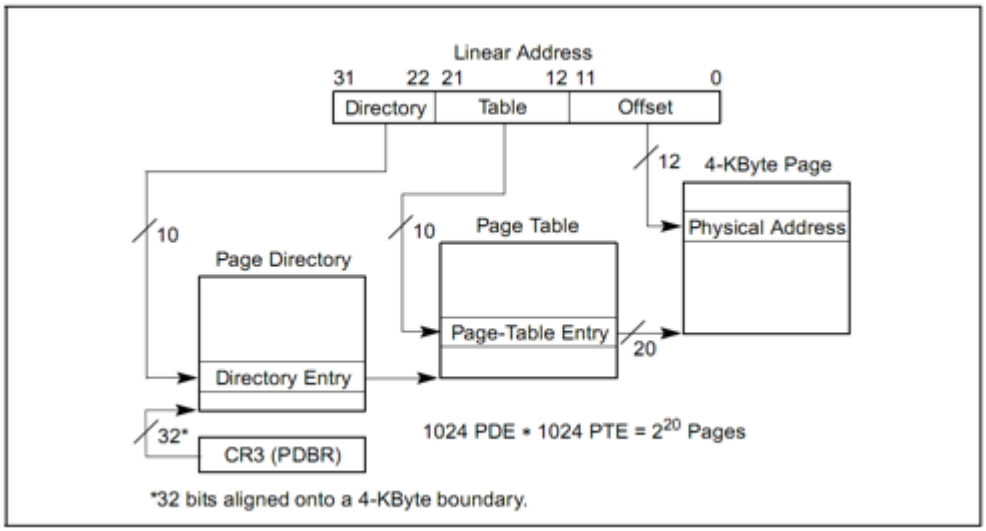
- 两种模式的部分细节
  - 实模式
    - 8086的模式，**只能访问不超过1MB的物理内存**
    - 因为8086只有20根地址线，16位的寄存器，所以单个的16位寄存器是无法访问到1M的地址空间的，只有64KB。所以就加入段寄存器，段寄存器出一点，其它寄存器（例如通用寄存器）出16位的段内偏移，就可以寻址到1MB了
      - 段寄存器也是16位的，左移4位，能够访问的是1MB地址空间的任意位置，这个就是段的**基址**
      - 段的基址+16位寄存器中的偏移，就能够访对应的物理地址了
    - 80386虽然可以访问4GB的空间了，所以兼容8086，加电的时候其实最开始也是**实模式**
  - 保护模式
    - 分段机制将逻辑地址（虚拟地址）转换为线性地址
      - 没有引入分页机制的时候，线性地址就是物理地址
    - CS(16-bit)指向一个段描述符，会有一些描述，加上offset就成为了物理地址
      - CS中的值就是GDT表中数组的下标
      - GDT表的基地址由GDTR寄存器提供
        - **非进程上下文**
    - 80386的地址空间。80386是32位的处理器，即可以寻址的物理内存地址空间为2^32=4G字节。为更好理解面向80386处理器的ucore操作系统，需要用到三个地址空间的概念：
    - 物理地址
      - 物理内存地址空间是处理器提交到**总线**上用于访问计算机系统中的**内存**和**外设**的最终地址。一个计算机系统中只有一个物理地址空间
    - 线性地址
      - 编程的时候的那个地址是虚拟地址，仅仅是代码段或数据段中的一个偏移，所以要和段寄存器中的基址加起来得到的地址就是线性地址
    - 逻辑地址（虚拟地址）
      - 每一个进程都有独立的虚拟地址空间，看起来好像独占系统的全部内存空间，虚拟地址空间的大小与可寻址范围有关
        - 32位 4GB
        - 64位 2^64
    - **所以有虚拟地址经过段表转为线性地址，再经过页表转为物理地址**
- 只有段模式下的内存布局，比较初期的结果

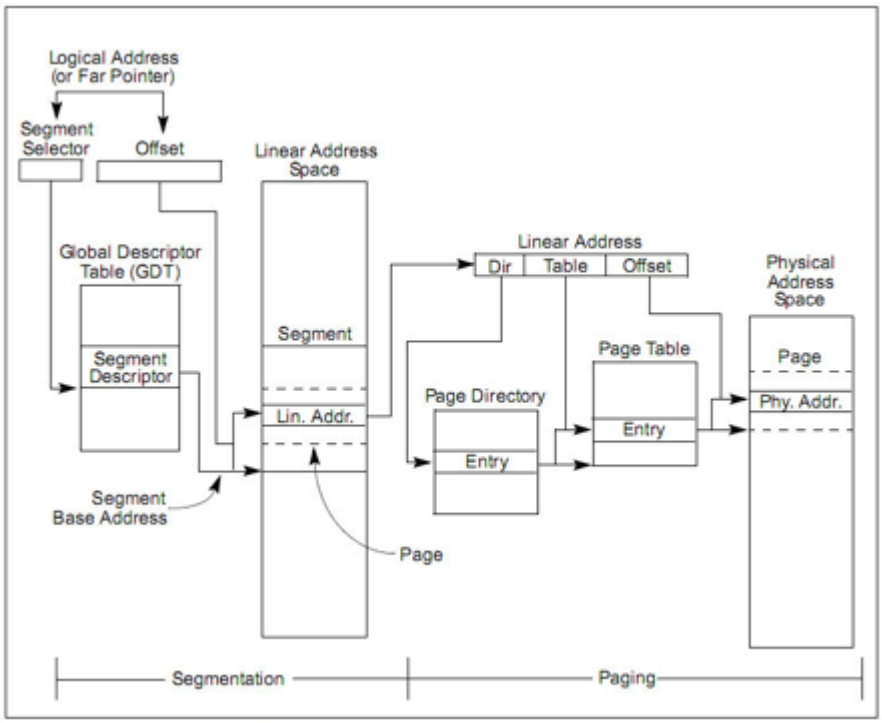


- 段页式管理
  - kernel执行前加载初始页表，并且完成一个4M的映射
    - kernel部分的执行就完全ok了现在
  - tips
    - 段页机制是有功能上的重叠的
    - 段的大小不确定，页的大小是确定的
    - 段表只有一个，页表可以有很多个
      - 每个进程都有一个页表，内核有一个页表
    - 尽量去弱化段的管理
    - **物理页的分配，返回的是虚拟地址，尽管代码内部可能会涉及物理地址的转换，但是呈现给程序员的一定就是虚拟地址**



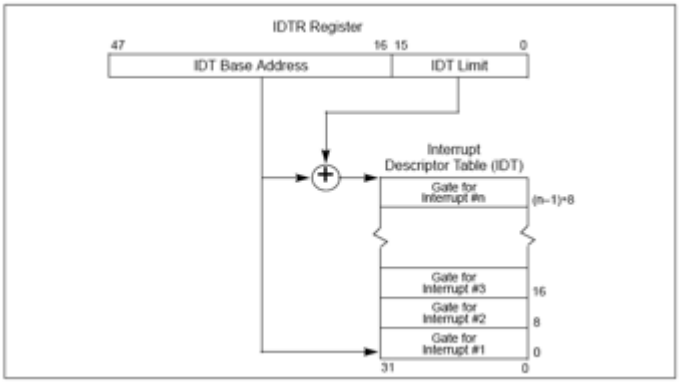
- 比较完善的段表
  - 现在cpu访问每一个地址都要访问两次，解决方案 cache
    - 实际上，对于CS段寄存器，实际上只有段选择子是程序员可见的，选择子去访问GDT实际上是为了得到一个base address，**这个base address是可以cache在CS寄存器中的**
      - 省了一次内存访问

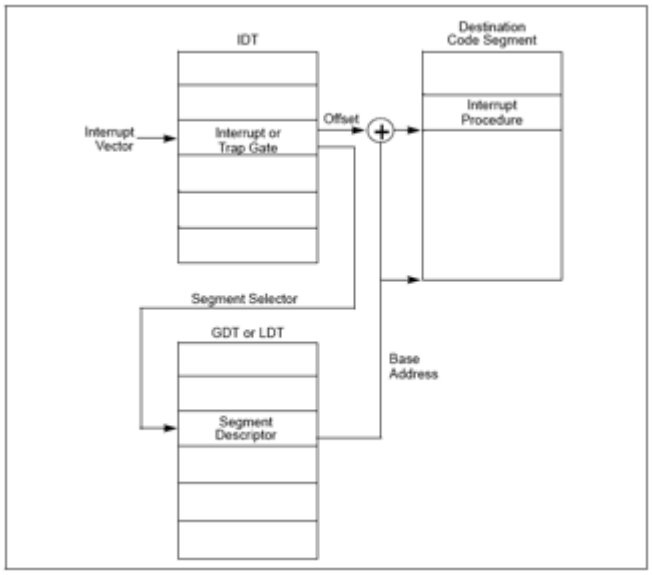




- CR3是页目录表的基址，是物理地址
- 也有cache，否则要访问非常多次的内存哦，必须缓存CPU中，即TLB
  - 开销其实是不大的，但是要避免TLB的shootdown问题

来好好聊一聊中断以及用户态与核态的switch





- 中断机制给操作系统提供了处理意外情况的能力，同时它也是实现进程/线程抢占式调度的一个重要基石
- 在操作系统中，有三种特殊的中断事件
  - 由CPU外部设备引起的外部事件如I/O中断、时钟中断、控制台中断等是异步产生的（即产生的时刻不确定），与CPU的执行无关，我们称之为异步中断(asynchronous interrupt)也称外部中断,简称**中断(interrupt)**
  - 而把在CPU执行指令期间检测到不正常的或非法的条件(如除零错、地址访问越界)所引起的内部事件称作同步中断(synchronous interrupt)，也称内部中断，简称**异常(exception)**
  - 在程序中使用请求系统服务的系统调用而引发的事件，称作陷入中断(trap interrupt)，也称软中断(soft interrupt)，系统调用(system call)**简称trap**
- 中断向量和中断服务例程的对应关系主要是由IDT（中断描述符表）负责。操作系统在IDT中设置好各种中断向量对应的中断描述符，留待CPU在产生中断后查询对应中断服务例程的起始地址。而IDT本身的起始地址保存在idtr寄存器中
  - 中断描述符表（Interrupt Descriptor Table）
    - 中断描述符表把每个中断或异常编号和一个指向中断服务例程的描述符联系起来。同GDT一样，IDT是一个8字节的描述符数组，但IDT的第一项可以包含一个描述符。CPU把中断（异常）号乘以8做为IDT的索引。IDT可以位于内存的任意位置，CPU通过IDT寄存器（IDTR）的内容来寻址IDT的起始地址
      - 在保护模式下，最多会存在256个Interrupt/Exception Vectors。范围[0，31]内的32个向量被异常Exception和NMI使用，但当前并非所有这32个向量都已经被使用，有几个当前没有被使用的，请不要擅自使用它们，它们被保留，以备将来可能增加新的Exception。范围[32，255]内的向量被保留给用户定义的Interrupts。Intel没有定义，也没有保留这些Interrupts。用户可以将它们用作外部I/O设备中断（8259A IRQ），或者系统调用（System Call 、Software Interrupts）
  - IDT gate descriptors

        - Interrupts/Exceptions应该使用Interrupt Gate和Trap Gate，它们之间的唯一区别就是：当调用Interrupt Gate时，**Interrupt会被CPU自动禁止**；**而调用Trap Gate时，CPU则不会去禁止或打开中断**，而是保留它原来的样子
        - 所谓自动禁止指的是CPU跳转到interrupt gate里的地址时，在将EFLAGS保存到栈上之后，清除EFLAGS里的IF位，以避免重复触发中断。在中断处理例程里，操作系统可以将EFLAGS里的IF设上,从而允许嵌套中断
            - 关中断也是通过修改寄存器中的值去实现的
    - 中断处理中硬件负责完成的工作，中断服务例程包括具体负责处理中断（异常）的代码是操作系统的重要组成部分。需要注意区别的是，**有两个过程由硬件完成**
        - 进入
            1. CPU在执行完当前程序的每一条指令后，都会去确认在执行刚才的指令过程中中断控制器（如：8259A）是否发送中断请求过来，如果有那么CPU就会在相应的时钟脉冲到来时从总线上读取中断请求对应的中断向量（**中断号的生成是硬件来做的**）
            2. CPU根据得到的中断向量（以此为索引）到IDT中找到该向量对应的中断描述符，中断描述符里保存着中断服务例程的段选择子
            3. CPU使用IDT查到的中断服务例程的段选择子从GDT中取得相应的段描述符，段描述符里保存了中断服务例程的段基址和属性信息，此时CPU就得到了中断服务例程的起始地址，**并跳转到该地址**
            4. CPU会根据CPL（**当前特权级**）和中断服务例程的段描述符的DPL（**目标的特权级**）信息确认是否发生了特权级的转换
                - 比如当前程序正运行在用户态，而中断程序是运行在内核态的，则意味着发生了特权级的转换
                    - **这时CPU会从当前程序的TSS信息（该信息在内存中的起始地址存在TR寄存器中）里取得该程序的内核栈地址**，即包括内核态的ss和esp的值，并立即将系统当前使用的栈切换成新的内核栈
                    - 这个栈就是即将运行的中断服务程序要使用的栈
                    - 紧接着就将当前程序使用的用户态的ss和esp压到新的内核栈中保存起来
                - **用户态到中断与内核态处理中断是不一样的**
                    - 但是一定都是在对应的内核栈中被处理的
                - 中断服务例程（上半部）是不可能被打断的，即使发生中断的嵌套
                    - **用户态进程由于中断进入核心态之后，中断处理完成之后返回用户态一定是之前的进程，不会被调度的**
            5. CPU需要开始保存当前被打断的程序的现场（即一些寄存器的值），以便于将来恢复被打断的程序继续执行。这需要利用内核栈来保存相关现场信息，即依次压入当前被打断程序使用的eflags，cs，eip，errorCode（如果是有错误码的异常）信息
            6. CPU利用中断服务例程的段描述符将其第一条指令的地址加载到cs和eip寄存器中，**开始执行中断服务例程**。这意味着先前的程序被暂停执行，中断服务程序正式开始工作

        - 你能看到中断相关的内容被执行的时候，就意味着软件的工作已经开始了
    - 返回，每个中断服务例程在有中断处理工作完成后需要通过iret（或iretd）指令恢复被打断的程序的执行。CPU执行IRET指令的具体过程如下
        1. 程序执行这条iret指令时，首先会从内核栈里弹出先前保存的被打断的程序的现场信息，即eflags，cs，eip重新开始执行
        2. 如果存在特权级转换（从内核态转换到用户态），则还需要从内核栈中弹出用户态栈的ss和esp，这样也意味着栈也被切换回原先使用的用户态的栈了
        3. 如果此次处理的是带有错误码（errorCode）的异常，CPU在恢复先前程序的现场时，并不会弹出errorCode。这一步需要通过软件完成，即要求相关的中断服务例程在调用iret返回之前添加出栈代码主动弹出errorCode

时钟中断初始化的例子

```
// set 8253 timer-chip
void clock_init(void) {
    outb(TIMER_MODE, TIMER_SEL0 | TIMER_RATEGEN | TIMER_16BIT);  # 向端口写数据，理解为set
    outb(IO_TIMER1, TIMER_DIV(100) % 256);
    outb(IO_TIMER1, TIMER_DIV(100) / 256);
    ...
    pic_enable(IRQ_TIMER);  // 使能中断
}
```

- 本质上是在set**时钟芯片**的属性
    - 每秒中断100次

IDT以及ISR的初始化过程

```
# handler
.text
.globl __alltraps  # 全局变量
.globl vector0  # 全局变量
vector0:
  pushl $0  # error num
  pushl $0  # 中断号
  jmp __alltraps
.globl vector1  # 全局变量
vector1:
  pushl $0  # 压入两个必要的参数后转到__alltraps处理
  pushl $1  # 两个参数均为硬件所产生的
  jmp __alltraps
...

# vector table
.data
.globl __vectors  # 全局变量
__vectors:
  .long vector0
  .long vector1
  .long vector2
  ...
```

- 首先中断服务例程（Interrupt Service Routine）是写死在Vector.S中的
  - 分别存在于内核 `.so` 的代码段与数据段
    - 数据段是向量表本身（**数组，值是对应的服务例程的地址**）
    - 代码段中保存的是服务例程，即对应的处理函数
- `static struct gatedesc idt[256] = {{0}};`
  - 这个是 `trap.c` 中的代码，所以，**IDT表本身也是写死分配好的**，在内核的数据段，256项，初始化为0

```
static struct gatedesc idt[256] = {{0}};
...
void
idt_init(void) {
    extern uintptr_t __vectors[];  // 这个是中断服务例程的数组，在内存中，在内核地址空间中
    int i;
    for (i = 0; i < sizeof(idt) / sizeof(struct gatedesc); i++) {
        // 参数0代表这个是中断门，不是trap门
        // GD_KTEXT代表中断处理程序是位于GD_KTEXT中的，handler在内核中
        // __vectors[i] 直接指向对应中断号的处理例程
        // DPL_KERNEL 代表描述符的特权级别，表示什么态能够触发这个中断，比如说DPL_KERNEL代表只有内核态
        // 比如时钟中断，磁盘网卡的中断等等
        SETGATE(idt[i], 0, GD_KTEXT, __vectors[i], DPL_KERNEL);  // 初始化IDT表格中的每一项，因为最
    }
        // set for switch from user to kernel
    // 允许用户态的代码到内核态，处理例程依然位于内核中，是用户态触发的
    // 允许在用户态通过一条指令生成一个软中断，否则用户态无法通过中断进入到内核态，确保这是允许的
    SETGATE(idt[T_SWITCH_TOK], 0, GD_KTEXT, __vectors[T_SWITCH_TOK], DPL_USER);
        // load the IDT
    lidt(&idt_pd);  // 告诉CPU IDT表在哪里，把地址给到寄存器IDTR中 全局唯一的
}
###### 解释一下IDT表中的entry，也就是门描述符每一项
/* *
 * Set up a normal interrupt/trap gate descriptor
 *   - istrap: 1 for a trap (= exception) gate, 0 for an interrupt gate
 *   - sel: Code segment selector for interrupt/trap handler
 *   - off: Offset in code segment for interrupt/trap handler
 *   - dpl: Descriptor Privilege Level - the privilege level required
 *          for software to invoke this interrupt/trap gate explicitly
 *          using an int instruction.
 * */
#define SETGATE(gate, istrap, sel, off, dpl) {
    (gate).gd_off_15_0 = (uint32_t)(off) & 0xffff;
    (gate).gd_ss = (sel);
    (gate).gd_args = 0;
    (gate).gd_rsv1 = 0;
    (gate).gd_type = (istrap) ? STS_TG32 : STS_IG32;
    (gate).gd_s = 0;
    (gate).gd_dpl = (dpl);
    (gate).gd_p = 1;
    (gate).gd_off_31_16 = (uint32_t)(off) >> 16;
}
```

- 在内核的init函数中（**中断服务例程以及中断描述符门都已经加载到内存中了**）
  - 现在要做的就是填充中断描述符表，并且将其基址写入到IDTR寄存器中
    - 利用函数 `SETGATE` 填充256个门描述符，保证通过中断号以及IDT表能够找到对应的中断例程

- 门描述符填充好之后，如果发生了中断，中断服务例程在将中断号（硬件给的）压栈之后到达 `__alltraps`
    - **在到达这里之前硬件已经压栈了部分寄存器了**

```
/* 中断发生之后，在内核栈上压栈的内容可以用以下的结构体来描述
 * 发生中断后，硬件与软件需要保存的内容，即trapframe
 * 说白了就是用来保存寄存器的
 * 在实际的内核中（5.15），struct trapframe 对应的数据结构是 struct pt_regs
 */
struct trapframe {
    # 软件需要去保存的内容
    uint16_t tf_gs;  # 若干需要被压栈的寄存器，均为发生中断时的寄存器状态
    uint16_t tf_fs;
    uint16_t tf_es;
    uint16_t tf_ds;
    uint32_t tf_trapno;  # 中断编号
    # 下面的数据发生中断的时候由硬件压栈
    uint32_t tf_err;  # error num
    uintptr_t tf_eip; # 发生中断时的 eip 寄存器，中断完成之后要恢复到该指令处继续取指
    uint16_t tf_cs;  # 发生中断时的 cs 寄存器
    ...
    uint32_t tf_eflags;
    # 下面的两个数据在特权级发生转变的时候回被压栈，例如用户态由于响应中断进入内核态
    uintptr_t tf_esp; # 用户态的esp
    uint16_t tf_ss;  # 如果是用户态进入到内核态，则用户态的ss寄存器是栈顶的第一个值
    ...
} __attribute__((packed));
```

```
# 研究一下软件的中断例程处理，所有中断都是一样的，不同的服务例程只是将不同的中断号压栈了
# vectors.S sends all traps here 所有的中断都被发到这里来
.text  # 位于内核的代码段
.globl __alltraps  # 这是一个全局符号
__alltraps:
    # push registers to build a trap frame
    # 此时栈上以及有的结构 error num, trap_no
    pushl %ds  # 这些都是发生在中断时的寄存器现场，如果是由用户态进入到内核态，则ds指向用户态的数据段
    pushl %es
    pushl %fs
    pushl %gs
    pushal
    # 压栈完成，现场保存完毕

    # 将内核态的代码段与数据段给到对应的段寄存器中，中断处理程序一定是内核代码
    # 其它cs，ss寄存器硬件会变
    movl $GD_KDATA, %eax
    movw %ax, %ds
    movw %ax, %es

    # 现在栈上已经有了一个比较完整的trapframe的结构，起始地址就是当前的rsp寄存器
    # 最后压入 esp，作为 trap 函数参数(struct trapframe * tf) 并跳转到中断处理函数 trap 处
    pushl %esp  # esp现在保存的就是tf的地址
    # call函数返回后执行下一句指令 popl %esp以及__trapret
    call trap  # 具体的中断处理例程在下面来说

    # 中断服务执行完毕之后，执行到这里来
    # 将栈顶元素pop到esp中，相当于esp恢复原状
    popl %esp
.globl __trapret
__trapret:
    # 一顿恢复
    # restore registers from stack
    popal
    # restore %ds, %es, %fs and %gs
    popl %gs  # 栈顶元素pop出，赋值给 %gs 寄存器
    popl %fs
    popl %es
    popl %ds
    # get rid of the trap number and error code
    addl $0x8, %esp  # 现在esp指向的位置是中断之前的eip的值

    # 现在中断帧中保存的寄存器包括
    # eip
    # cs
    # eflags
    # esp 如果是用户态进入的话
    # ss 如果是用户态进入的话

    # iret指令返回到相同的CPU保护级别的时候，会弹出eip cs eflags，并且跳转到eip处执行
```

```
# iret指令返回到不同的CPU保护级别的时候，会弹出eip cs eflags，esp，ss，跳转到eip执行
iret
```

- 下面看一下具体的中断处理函数

```
void
trap(struct trapframe *tf) {
    // dispatch based on what type of trap occurred
    trap_dispatch(tf);
}

# 用户态与核心态的转化，这里仅仅是一个模拟操作哈
static void
lab1_switch_to_user(void) {
        asm volatile (
            "sub $0x8, %%esp \n"  // rsp寄存器的值减去0x8，即16
            "int %0 \n"  // int一个中断号，指令是 int T_SWITCH_TOU，这个地方开始压栈了，压trapframe
            "movl %%ebp, %%esp"  // 中断处理完成之后，eip执行到这里来，这相当于之前esp相对于ebp增长的
            :
            : "i"(T_SWITCH_TOU)
        );
}

static void
lab1_switch_to_kernel(void) {
        asm volatile (
            "int %0 \n"  # int T_SWITCH_TOK
            "movl %%ebp, %%esp \n"  // 中断处理完成之后处理的指令，直接恢复esp与ebp一致
            :
            : "i"(T_SWITCH_TOK)
        );
}

// 完成对具体中断的处理
static void
trap_dispatch(struct trapframe *tf) {
    ...
    switch (tf->tf_trapno) {
    case IRQ_OFFSET + IRQ_TIMER:  // 时钟中断
        ticks ++;
        if (ticks % TICK_NUM == 0) {print_ticks();}
        break;
    // 通过软中断到用户态 to user
    case T_SWITCH_TOU:
        if (tf->tf_cs != USER_CS) {
            // 这里要首先明确，中断处理完成之后，是要返回到中断指令的后一条指令去执行的，这里全部都只是这
            // 一般来说，内核态到用户态的前提是用户态先进入了内核态，并且将用户态的cs,ds,esp寄存器压栈后
            // 这里的实验室由内核态直接进入用户态的 trapframe中的某些数据是状态转换前的，所以要set一下t
            switchk2u = *tf;  // 是trapframe的底，即低地址处
            switchk2u.tf_cs = USER_CS;  // 用户态的代码段，这个是在内存初始化的时候，重新加载GDT表的这
            switchk2u.tf_ds = switchk2u.tf_es = switchk2u.tf_ss = USER_DS;  // 用户态的数据段
            // 用户态的栈指针，最开始esp的值沿着栈的生长方向走了16个字节，等下其实就是继续在这个栈上去执
            switchk2u.tf_esp = (uint32_t)tf + sizeof(struct trapframe) - 8;  // iret之后这个esp的

                    // 允许用户态可以执行I/O操作
            // set eflags, make sure ucore can use io under user mode.
```

```
            // if CPL > IOPL, then cpu will generate a general protection.
            switchk2u.tf_eflags |= FL_IOPL_MASK;

            // set temporary stack
            // then iret will jump to the right stack
            *((uint32_t *)tf - 1) = (uint32_t)&switchk2u;

            // 执行完毕后，开始pop栈上的内容，并且部分pop出的值给到对应的寄存器中
            // iret将esp的值给到寄存器esp中，并且跳转到eip的执行位置，即int指令的下一条指令处
            // 这个时候栈的地址实际上是没有变化的，但是段寄存器（代码段与数据段都成为了用户态的，cpu的特
            // 所以这个时候执行的代码虽然还是内核中的print函数，但是实际上已经是用户态了
        }
        break;
    // 用户态到内核态 通过系统调用进入内核态
    case T_SWITCH_TOK:
        if (tf->tf_cs != KERNEL_CS) {
            // 细节不看了，说白了这里就是一个压栈与弹栈的过程，压栈后把trapframe中的某些寄存器值设置为内
            // 弹栈后，恢复寄存器到内核态的现场，实际不是这样的，实际上是TSS段保存了用户态到核态的ss与e
            tf->tf_cs = KERNEL_CS;
            tf->tf_ds = tf->tf_es = KERNEL_DS;
            tf->tf_eflags &= ~FL_IOPL_MASK;
            switchu2k = (struct trapframe *)(tf->tf_esp - (sizeof(struct trapframe) - 8));
            memmove(switchu2k, tf, sizeof(struct trapframe) - 8);
            *((uint32_t *)tf - 1) = (uint32_t)switchu2k;
        }
        break;
    default:
        // in kernel, it must be a mistake
        ...
    }
}
```

x86—64的页表

name	offset
页全局目录(Page Global Directory)	39-47
页上级目录(Page Upper Directory)	30-38
页中间目录(Page Middle Directory)	21-29
页表(Page Table)	12-20
页内偏移(Page Offset)	0-11

上帝视角看进程调度

- https://zhuanlan.zhihu.com/p/363196316
  - 如果在timer中处理的是函数，则与进程调度关系不大
    - 这个timer是某一个task的
  - 但是timer还会处理进程的调度
    - 如果在时钟中断中，发现进程的时间片到期了
    - 那就调度到另一个进程
- 时钟中断的中断处理函数做以下操作

```
_timer_interrupt:
    ...
    # 增加系统滴答数
    incl _jiffies
    ...
    # 调用函数 do_timer
    call _do_timer
    ...
```

```
void do_timer(long cpl) {
    ...
    // 当前线程还有剩余时间片，直接返回
    if ((--current->counter)>0)
        return;
    // 若没有剩余时间片，调度
    schedule();
}
```

```
void schedule(void) {
    int next = get_max_counter_from_runnable();
    refresh_all_thread_counter();
    switch_to(next);
}
```

1. 拿到剩余时间片（counter的值）最大且在 runnable 状态（state = 0）的进程号 next
2. 如果所有 runnable 进程时间片都为 0，则将所有进程（注意不仅仅是 runnable 的进程）的 counter 重新赋值（counter = counter/2 + priority）
   - 然后再次执行步骤 1
3. 最后拿到了一个进程号 next，调用了 switch_to(next) 这个方法，就切换到了这个进程去执行了

进程的优先级

- Nice值的范围是 `-20~+19` ，拥有Nice值越大的进程的实际优先级越小
  - 即Nice值为 `+19` 的进程**优先级最小**

    - 为 `-20` 的进程**优先级最大**
    - 默认的Nice值是0
- 由于Nice值是**静态优先级**，所以**一经设定，就不会再被内核修改**，直到被重新设定
    - Nice值只起干预CPU时间分配的作用，实际中的细节，**由动态优先级决定**