Linux内存初始化

针对多种体系结构，linux使用不同的bootloader，因为我看的代码是x86下的，所以我们全文的内核引导程序都指的是grub。其实每个bootloader都很复杂，比如grub如何读，把它加载到哪，怎么执行内核。

第一个问题，grub是怎么读内核映像的？好吧，再说一次，BIOS启动后，grub首先根据系统盘的第一个块，即512字节中的数据（bootsect）来判断识别文件系统的代码（512字节显然不能识别文件系统）在磁盘中的什么块，这些数据是grub安装的时候记录的，然后执行下一个步骤，即识别系统盘的文件系统。

识别系统盘的文件系统主要是通过initrd程序来完成，前面也提到了，initrd就像个小操作系统，但它的作用仅仅是让grub能够获得磁盘驱动程序以识别内核映像所在的文件系统，所以initrd.img压缩包并不大，只有3.5MB左右。initrd程序还是很复杂的，只不过不是我们的研究重点，有兴趣的同志可以尝试解压缩initrd-2.6.18-194.el5.img文件（先要改名成.gz文件，然后用gunzip程序解压缩），并将其mount到一个指定位置。这时你就可以看到，这个内存文件系统提供若干命令和脚本。其中最重要的是提供系统初始化的linuxrc脚本。必须注意的是这里使用的shell是nash而不是bash，nash是专门为linuxrc可执行脚本设计的，因此你也有必要看一看nash的man文档。

总结一下Grub的这一个过程，就是：
1. 调用一个BIOS过程显示“Loading”信息。
2. 调用一个BIOS过程从磁盘装入内核映像的初始部分，即将内核映像的第一个512字节加载到物理地址0x00090000开始的内存中，而将setup程序的代码（参见后面的内存布局）从地址0x00090200开始存入RAM中。
3. 调用一个BIOS过程从磁盘中装载其余的内核映像，并把内核映像放入从低地址0x00010000（适用于使用make zImage编译的小内核映像）或者从高地址0x00100000（适用于使用make bzImage编译的大内核映像，也就是我们现在的情况）开始的RAM中。大内核映像的支持虽然本质上与其他启动模式相同，但是它却把数据放在不同的物理内存地址，以避免ISA黑洞问题。
4. 跳转到arch/x86/boot/header.S的_start处开始执行。

arch/x86/boot/header.s
里面的bootsect_start是无用代码，用于BIOS误将其导入0x07c0内存后打印错误信息，真正代码是以_start开始。此处代码在初始化中并不会执行。

从_start到start_of_setup的代码是些伪指令，即setup的初始化头变量hdr，涉及的是内存布局，并不涉及具体的指令执行。

arch/x86/boot/main.c
总结一下，汇编代码header.S从开始到 # offset 512, entry point功能和以前的bootsect一样。后面的功能和setup.S的一部分类似，主要的工作是设置setup header参数部分；设置堆栈；检查setup中的标签；清除BSS段；调用C入口main。此时cs:eip指向的是main，ss:esp指向的是堆栈栈顶，ds:edi指向哪儿不知道，也不重要。

这段代码实际上还是系统处于是模式下的代码。这个main函数里面的代码大部分对应ULK3中附录A讲setup函数的那一小节，主要是在实模式下做些前期工作。涉及初始化计算机中的硬件设备，并为内核程序的执行建立环境。虽然前面看到BIOS已经初始化了大部分硬件设备，但是Linux并不依赖于BIOS，而是以自己的方式重新初始化设备以增强可移植性和健壮性。

到保护模式的代码了，最先执行的代码就是arch/x86/boot/compressed/head_32.S中的startup_32，对于bzImage，grub把它加载到0x100000的位置。

解压缩内核使用的是decompress_kernel函数，来自arch/x86/boot/compressed/misc.c

开始执行解压缩后的第一条代码，即第二个startup_32()函数。这个函数主要是为第一个Linux进程（进程0）建立执行环境。该函数主要执行以下操作：
1. 把段寄存器初始化为最终值。
2. 把内核的bss段填充为0。
3. 初始化包含在swapper_pg_dir的临时内核页表，并初始化pg0，以使线性地址一致地映射同一物理地址。
4. 把页全局目录的地址存放在cr3寄存器中，并通过设置cr0寄存器的PG位启用分页。
5. 把从BIOS中获得的系统参数和传递给操作系统的参数boot_params放入第一个页框中。
6. 为进程0建立内核态堆栈。
7. 该函数再一次清零eflags寄存器的所有位。
8. 调用setup_idt用空的中断处理程序填充中断描述符表IDT。
9. 识别处理器的型号。
10. 用编译好的GDT和IDT表的地址来填充gdtr和idtr寄存器。
11. 初始化虚拟机监视器Xen。
12. 向start_kernel()函数进发。

arch/x86/kernel/head_32.S

grub引导
- stage1
  - /stage1/start.s 利用BIOS读入0头0道2扇区
- stage1.5
  - 和文件系统有关
- stage2
  - /stage2/start.s 提供grub启动菜单和交互式的GRUB的shell
  - stage2对启动系统起关键作用，该部分提供了GRUB启动菜单和交互式的GRUB的shell。启动菜单在启动时候通过/boot/grub/grub.conf文件所定义的内容生成。在启动菜单中选择了kernel之后，GRUB会负责解压和装载kernel image并且将initrd也解压并装载到内存中。最后GRUB初始化kernel启动代码。完成之后后续的引导权被移交给kernel。
  - 注意grub的stage2会临时地打开保护模式，把内核映像vmlinuz-2.6.x除setup的其余部分从磁盘整体加载到内存，然后再回到实模式下。所以此时，在内存中的内核映像分成了两个部分，实模式部分和保护模式部分，其分界线就是物理地址0x100000。而实模式部分也分成了两部分，bootsect部分和setup部分。
  - 关于grub常用的几个指令对应的函数：
    grub>root (hd0,0) --root指令为grub指定了一个根分区
    grub>kernel /vmlinuz-2.6.18-194.el5 --kernel指令将操作系统内核载入内存
    grub>module /vmlinuz-2.6.18-194.el5xen ro root=/dev/sda2 --module指令加载指定的模块
    grub>initrd / initrd-2.6.18-194.el5.img --指定initrd文件
    grub>boot --boot 指令调用相应的启动函数启动OS内核

    当执行grub>boot指令后，grub载入内核vmlinuz并解压缩到指定位置（后面会讲到vmlinuz是如何解压缩的），同时载入initrd.img到内存，系统启动的控制权移交给kernel。

bootsect_start

_start
- start_of_setup

main
- copy_boot_params
  - 用来拷贝启动参数到内存的0号页面中
- init_heap
  - 用来检查内核初始化阶段使用的堆
- validate_cpu
  - 确保支持当前运行的CPU
- set_bios_mode
  - 设置BIOS的X86模式
- detect_memory
  - 这里是第一次出现与内存管理相关的代码。
  - detect_memory_e820
  - detect_memory_e801
  - detect_memory_e88
- keyboard_set_repeat
  - 设置键盘属性
- query_mca
- query_ist
  - 填充系统环境配置表
- set_video
  - 填充IST信息
- go_to_protected_mode
  - Linux/arch/X86/boot/video.c 设置视频模式
  - 进入保护模式
  - 禁止可屏蔽和不可屏蔽中断
  - realmode_switch_hook
  - enable_a20
  - reset_coprocessor
  - mask_all_intterupts
  - setup_idt
  - setup_gdt
  - protected_mode_jump

startup_32
- decompress_kernel
  - 注意，在进入保护模式后，Linux进行了第二次段寻址的设置，也就是第二次启动保护模式，这一次设置的原因是在之前的处理过程中，指令地址是从物理地址0x100000开始的，而此时整个vmlinux的编译链接地址是从虚拟地址0xC0000000开始的，所以需要在这里重新设置boot_gdt的位置。

startup_32
- 第二次启动保护模式
- page_pde_offset 第一次启动分页管理
- 初始化中断描述符表
- 加载全局/中断描述符表
- 第三次启动保护模式
- 启动x86虚拟机
- 初始化0号进程

根据e820的数据来获得32位可用物理内存地址的最大值并右移PAGE_SHIFT，也就是12位，最后由函数e820_end_pfn返回这个20位的值，保存在内部变量max_pfn中，作为总的页面数量

start_kernel
- init/main.c
- boot_cpu_init
  - 激活第一个CPU
- page_address_init
  - 初始化地址散列表 mm/highmem.c
- setup_arch
  - setup_memory_map
    - 拷贝可用内存区信息
    - x86_init.resources.memory_setup
      - arch/x86/kernel/x86_init.c
      - default_machine_specific_memory_setup
  - e820_end_of_ram_pfn
    - e820_end_pfn
  - find_low_pfn_range
    - highmem_pfn_init
    - 调用find_low_pfn_range()，它根据max_pfn是否大于MAXMEM_PFN来判断是否启用了高端内存区。这个MAXMEM_PFN值等于多少，大家自己去查，不是很复杂，只是这个值跟max_pfn一样，也是除去低12位的20位的一个数值。如果启用了，就调用highmem_pfn_init()函数将全局变量max_low_pfn设置为MAXMEM_PFN；并设置全局变量highmem_pages为max_pfn - MAXMEM_PFN，作为高端页面的数量：
  - init_memory_mapping
    - 建立内核永久页表
    - find_early_table_space
      - 确定了PUD和PMD以及PTE、固定内存映射等所有的选项所使用的内存空间的大小
      - find_e820_area
      - find_early_area
    - kernel_physical_mapping_init
      - 把整个物理内存地址都映射到从内核空间的开始地址，即从0xc0000000的整个内核空间中，直到物理内存映射完毕为止。
    - initmem_init
      - arch/x86/mm/numa_32.c
      - arch/x86/mm/init_32.c
    - setup_bootmem_allocator
      - x86下面没有init_bootmem，而是这个setup函数
  - paging_init
    - 页面初始化
    - pagetable_init
      - permanent_kmaps_init
        - page_table_range_init
    - kmap_init
    - sparse_init
    - zone_sizes_init
      - free_area_init_nodes
        - free_area_init_node
          - free_area_init_core
            - init_currently_empty_zone
              - zone_init_free_lists
  - mm_init_owner
    - 设置每CPU环境
  - build_all_zonelists
    - 初始化内存管理区列表 mm/page_alloc.c
  - page_alloc_init
    - 利用early_res分配内存
  - vfs_caches_init_early
    - vfs_caches_init_early调用两个函数dcache_init_early和inode_init_early。内核第一次触碰文件系统的主要目的就是初始化VFS的两个重要数据结构dcache和inode的缓存。
  - sort_main_extable
    - 把编译期间，kbuild设置的异常表，也就是__start___ex_table和__stop___ex_table之中的所有元素进行排序。
  - trap_init
    - 初始化中断向量表
  - mm_init
    - （1）内存区的开始部分包含的是对前896MB RAM 进行映射的线性地址。直接映射的物理内存末尾所对应的线性地址保存在high_memory全局变量中。当物理内存小于896MB，则线性地址0xc0000000以后的896MB与其一一对应；当物理内存大于896MB而小于4GB时，只直接映射前896MB的地址到0xc0000000以后的线性空间，然后把线性空间的其他部分与896MB和4GB物理空间映射起来，称为动态重映射，这是本博的重点；当物理内存大于4GB，则需要考虑PAE的情况，其他的东东没什么区别，我们不做过多的回忆了。
    （2）内核的页表由内核页全局目录变量swapper_pg_dir维护；pagetable_init()建立内核页表项。
    （3）内存区的结尾部分包含的是固定映射的线性地址，主要用于存放一些常量线性地址，具体查看“高端内存映射 ”博文。
    （4）从PKMAP_BASE 开始，我们查找用于高端内存页框的永久内核映射的线性地址，具体查看“高端内存映射 ”博文。
    （5）其余的线性地址可以用于非连续内存区。在物理内存映射的末尾与第一个内存区之间插入一个大小为8MB（宏VMALLOC_OFFSET）的安全区，目的是为了“捕获”对内存的越界访问。出于同样的理由，插入其他4KB 大小的安全区来隔离非连续的内存区。
    - mem_init
      - 伙伴管理算法初始化
      - free_all_bootmem
        - free_all_memory_core_early
          - get_free_all_memory_range
    - kmem_cache_init
      - 初始化内核slab分配体系 mm/slab.c
    - pgtable_cache_init
      - 空函数
    - vmalloc_init
      - vmalloc的初始化
  - sched_init
  - kmem_cache_sizes_init