Table of Contents

读核感悟

读核感悟-Linux 内核启动-内核的生成

这段时间在看《Linux 内核源代码情景分析》，顺便写了一些感悟。

。

读内核源代码是一件很有意思的事。它像一条线，把操作系统，编译原理，C 语言，数据结构与算法，计算机体系结构等等计算机的基础课程串起来。

我看内核源代码是用 lxr+glimpse（不一定要自己架，可以直接访问校内外的 lxr 网站）的。如果在 windows 下也可以用 source insight。以下的当前路径为内核源代码路径，通常为/usr/src/linux。内核版本为 2.6.13，平台为 x86

好，让我们开始 Linux 内核之旅。

我们的出发点是在 CPU 加电的一刹那，系统处于 16 位实地址模式下，终点是内核开始运行 start_kernel()，系统处于 32 位页式寻址的保护模式下。那时内核映象 bzImage 已经解压完毕，运行于内核态。系统中已经有了一个叫 swapper 的 0 号进程，有自己的内核堆栈，情况就相对好理解得多。（尽管与用户态程序相比，还要多操心不少事，包括对硬件的直接操作，内核态各种数据结构的初始化，对页表的操作等等）。

不过，不妨先做些准备动作。

首先，什么是内核?

目前，只知道编译内核后，产生一个叫 bzImage 的压缩内核映象。它不同于任何普通的可执行程序。我们甚至不知道它从哪里开始执行。只知道把它往/boot/下一放，往 boot loader 的配置文件(例如 grub 的 menu.lst)中写上相关信息，机子就顺利启动了。

因此，我对它的生成过程产生了浓厚兴趣。于是，我查看了相关资料，最直接的资料来自于 arch/i386/boot/下的 Makefile。从 Makefile 中可以知道。bzImage 的产生过程是这样的:

不过我不满足于此。于是，我想到了去看 arch/i386/boot/下的 Makefile。从 arch/i386/boot/Makefile 和 arch/i386/boot/compressed/Makefile 中可以看出（具体过程省略，）

1.先生成 vmlinux.这是一个 elf 可执行文件
2.然后 objcopy 成 arch/i386/boot/compressed/vmlinux.bin，去掉了原 elf 文件中的一些无用的 section 等信息。
3.gzip 后压缩为 arch/i386/boot/compressed/vmlinux.bin.gz
4.把压缩文件作为数据段链接成 arch/i386/boot/compressed/piggy.o
5.链接：arch/i386/boot/compressed/vmlinux = head.o+misc.o+piggy.o
其中 head.o 和 misc.o 是用来解压缩的。
6.objcopy 成 arch/i386/boot/vmlinux.bin，去掉了原 elf 文件中的一些无用的 section 等信息。
7.用 arch/i386/boot/tools/build.c 工具拼接 bzImage = bootsect+setup+vmlinux.bin

过程好复杂。

这里要介绍一下 objcopy 命令，它的作用是把一个 object 文件转化为另一种格式的文件。在这里，objcopy 的作用就是去掉原来 elf 文件中的 elf header 和一些无用的 section 信息。为什么要这么做呢? 因为 elf 文件中的 elf header 和一些 section 的作用是告诉 elf loader 如何载入 elf 可执行文件。但是，linux 内核作为一种特殊的 elf 文件，需要特殊折辅助程序去装载它。往往它的装载地址是固定的。这时，为了保证通用性而存在的 elf header 和一些 section 对内核的装载就没有意义了。加上为了使内核尽可能小，所以干脆把这些信息去掉。

我们可以看一下 vmlinux 和 arch/i386/boot/compressed/vmlinux。用 file 命令查看，它们也是 elf 可执行文件。只是没有 main 函数而已

参考:

Documentation/kbuild/makefiles.txt
Documentation/kbuild/modules.txt

读核感悟-Linux 内核启动-从 hello world 说起

内核是从哪里开始执行的呢? 几乎任何一本 Linux 内核源代码分析的书都会给出详细的答案。不过，我试图从一个不同的角度（一个初学者的角度）来叙述，而不是一上来就给出答案。从熟悉的事物入手，慢慢接近陌生的事物，这是比较常见的思路。既然都是二进制代码，那么不妨从最简单的用户态 C 程序，hello world 开始。说不定能找到共同点。恰好我是一个喜欢寻根究底的人。也许，理解了 hello world 程序的启动过程，有助于更好地理解内核的启动。

好，开始寻根究底吧。从普通的 C 语言用户态程序开始写。先写一个简单的 hello world 程序。

```
/*helloworld.c*/
#include <stdio.h>
int main()
{
    printf("hello world\n");
    return 0;
}
```

然后 gcc helloworld.c -o helloworld，一个最简单的 hello world 程序出现了。

它是从哪里开始执行的呢？这还不简单？main 函数么。地球人都知道。

为什么一定要从 main 函数开始呢？于是，我开始琢磨这个 hello world 程序。

file helloworld 可知，它是一个 elf 可执行文件。

反汇编试试。

objdump -d helloworld

反汇编的结果令人吃惊，因为出现了_start()等一堆函数。一定是 gcc 编译时默认链接了一些库函数。

其实，只要运行 gcc -v helloworld.c -o helloworld 就会显示 gcc 详细的编译链接过程。其中包括链接/usr/lib/下的 crti.o crt1.o crtn.o 等等文件。用 objdump 查看，_start()函数就定义在 crt1.o 文件中。

那么 helloworld 的真正执行的入口在哪里呢？我们可以使用 readelf 来查看，看有没有有用信息。

readelf -a helloworld

helloworld 作为一个 elf 文件，有 elf 文件头，section table 和各个 section 等等。有兴趣可以去看看 elf 文件格式的文档。

用 readelf 可知，在 helloworld 的 elf 文件头的信息中，有这么一项信息：

入口点地址：　　0x80482c0

可见，helloworld 程序的入口地址在 0x80482c0 处，而由 objdump 得：

080482c0 <_start>:

可见，_start()是 helloworld 程序首先执行的函数。_start()执行完一些初始化工作后，经过层层调用，最终调用 main().可以设想，如果_start()里最终调用的是 foo()，那么 C 程序的主函数就不再是 main()，而是 foo()了。

再进一步：helloworld 程序具体是如何执行的呢。我们只能猜测是由 bash 负责执行的。然而具体看 bash 代码就太复杂了。我们可以用 strace 跟踪 helloworld 的执行。

strace ./helloworld

出来一大堆函数调用。其中第一个是 execve().这是一个关键的系统调用，它负责载入 helloworld 可执行文件并运行。其中有很关键的一步，就是把用户态的 eip 寄存器（实际上是它在内存中对应的值）设置为 elf 文件中的入口点地址，也就是_start()。具体可见内核中的 sys_execve()函数。

由此可见，程序从哪里开始执行，取决于在刚开始执行的那一刻的 eip 寄存器的值。而这个 eip 是由其它程序设置的，在这里，eip 是由 Linux 内核设置的。具体过程如下：

1.用户在 shell 里运行./helloworld。

2.shell（这里是 bash）调用系统调用 execve()。

3.execve 陷入到内核里执行 sys_execve()，把用户态的 eip 设置为_start()。

4.当系统调用执行完毕，helloworld 进程开始运行时，就从_start()开始执行

5.helloworld 进程最后才执行到 main()。

参考：elf 文件格式
http://www.skyfree.org/linux/references/ELF_Format.pdf

读核感悟-Linux 内核启动-BIOS

“真罗嗦，直接告诉我 Linux 下用 glibc 库编译出来的 C 程序真正的入口地址是_start()不就行了么？”臭鸡蛋扑面而来。

嗯，我说了我只是想用一种特别的方式来叙述问题。我更看重探索的过程中体现的思考方式以及其中的乐趣。

回到我们的主题。Linux 内核为什么不是从 main 函数开始执行？事实上，Linux 内核源代码里有许多 main()函数，但仔细一看。他们都是运行在用户态的。其实，从上一节中可以看到，main()函数只是一个符号而已。很多书上提到 start_kernel()。它类似于 main()。但正如调用 main()的是_start()。在 start_kernel 之前仍然有很多代码。所以内核的真正入口并不是 start_kernel()。

真正决定程序执行入口的是载入程序。对普通的 C 程序 helloworld 来说，Linux 内核（严格的说应该是 bash）负责设置 helloworld 的入口点，并且启动 helloworld 进程的执行。

于是，问题出现了，Linux 内核的载入程序是什么呢？难道是自己载入自己？

这类类似的问题在计算机史上出现过很多次，比如，C 程序可以用 C 编译器来写，那么，C 编译器用什么来写呢？当然，解决的方案有很多种，比如，用汇编语言写。那么汇编器用什么写呢？可以用机器语言写。虽然是件痛苦的事，但是一想到能造福这么多人，（发散一下，C 编译器可以编译出各种新的语言的编译器或者解释器如 perl，然后程序员们再用 perl 来编出各种复杂的系统），简直太伟大了，那位用机器语言写汇编器的程序员一定会浑身干劲。

可以借鉴一下思路，这类问题的常见的解决方式是构造一个简单的系统来解决一个复杂的系统的问题，如此往复，像滚雪球一下，最终把一个极为复杂的问题解决掉。

所以，很自然的想到，用一个简单的内核，（不，它仅仅是一段程序，甚至不能称之为内核，因为它的功能很有限）来启动真正的内核。就像用一个小当量的原子弹来引爆氢弹一样。

本着 KISS(keep it simple and stupid)的原则，你第一时间可能会想到 BIOS。BIOS 通常存在 ROM 上。随着 ROM 容量的扩展（达 1M 甚至更多）PC 上的 BIOS 功能已经很强大了，包括了很多设备的驱动程序。不仅可以进行硬件的检测，还实现了基本的输入输出功能。（basic input / output system，这也是它的本意）。开机时按 del 键出现的 BIOS 设置画面就是 BIOS 的杰作。在 BIOS 上实现一个 OS 也不是不可能。事实上 DOS 就是在 BIOS 的基础上实现的。

然而，BIOS 的一个致命的缺点是到目前为止它基本上只能在 16 位实地址模式下运行，毕竟 ROM 的容量很有限。（当然也不绝对，BIOS 也提供了在保护模式下扫描 PCI 设备的方法，大牛们 yy 一下实现一个保护模式下运行的 BIOS 的可行性。）而 CPU 刚加电时处在 16 位实地址模式下。它的另一个致命的缺点是太小。因为现代操作系统多种多样，BIOS 再强大也无法把所有可能的情况都考虑进去。

不过即使这样，PC 机刚启动时，x86 CPU 仍然会自动从 BIOS 开始启动，这是由硬件决定的，因为加电时，寄存器 CS 里的值为 0xffff，IP 里的值为 0。于是 CPU 从线性地址 0xffff0 处开始取指令。0xffff0 处是什么地方呢？

运行 cat /proc/iomem

前面 5 行

```
00000000-0009fbff : System RAM
0009fc00-0009ffff : reserved
000a0000-000bffff : Video RAM area
000c0000-000cc7ff : Video ROM
000f0000-000fffff : System ROM
```

可以看到 0xffff0 处是 System ROM，也就是 BIOS

注意这里的地址是 16 位实地址模式下的线性地址。

而 BIOS 能做的事，包括开机时对设备的检测，最后去读硬盘上的第一个扇区（即引导扇区 MBR）的 512 个字节，把它们拷贝到地址为 0x7c00 处的内存中。到此为止，BIOS 的引导使命已经完成。虽然以后 Linux 内核的引导还要用到 BIOS 的功能，但引导的使命已经落在其它程序上了。

读核感悟-Linux 内核启动-setup 辅助程序

我们发现，在起点与终点之间，还有几个中转站。最近的一站叫作 MBR。BIOS，带你到 MBR 后，说：“对不起，只能送你到这里了。”

那其它几个中转站是什么呢？我们知道，在 x86 上，保护模式有两种，32 位页式寻址的保护模式和 32 位段式寻址的保护模式。显然，32 位页式寻址的保护模式要求系统中已经构造好页表。从 16 位实地址模式直接到 32 位页式寻址的保护模式是很有难度的，因为你要在 16 位实地址模式下构造页表。所以不妨三级跳，先从 16 位实地址模式跳到 32 位段式寻址的保护模式，再从 32 位段式寻址的保护模式跳到 32 位页式寻址的保护模式。

我们需要这样一个程序，负责从 16 位实地址模式跳到 32 位段式寻址的保护模式，然后设置 eip，启动内核。这个程序的确存在，就是 arch/i386/boot/setup.S。最后汇编成 setup 程序。

事实上，平时所见的压缩内核映象 bzImage，其实由三部分组成。这可以从 arch/i386/boot/tools/build.c 中看出来。build.c 是用户态工具 build 的源代码，后者用来把 bootsect(MBR),setup(辅助程序)和 vmlinux.bin(压缩过的内核)拼接成 bzImage。

setup 有了，它可以启动内核，然而新的问题来了。谁来启动 setup 呢？第一个想到的是 MBR。可惜，MBR 只有 512 个字节，而且有 64 个字节来存放主分区表。这样看来，MBR 的功能就很有限了。所以，最好在 setup 和 MBR 之间再架一座桥梁。引导程序，对，用它来引导 setup 再合适不过了。现有的引导程序如 grub,lilo 不仅功能强大，而且还提供了人机交互的功能。再合适不过了。

所以，我们理清了大概思路，查阅相关资料可知：

1.CPU 加电，从 0xffff0 处，执行 BIOS（可以理解为“硬件”引导 BIOS）

2.BIOS 执行扫描检测设备的操作，然后将 MBR 读到物理地址为 0x7c00 处。然后从 MBR 头部开始执行（可以理解为 BIOS 引导 MBR）

3.MBR 上的代码跳转到引导程序，开始执行引导程序的代码，例如 grub（引以理解为 BIOS 引导 boot loader）。

4.引导程序把内核映象(包括 bootsect，setup，vmlinux)读到内存中，其中 setup 位于 0x90200 处，如果是 zImage，则 vmlinux.bin 位于 0x10000（64K）处。如果是 bzImage，则 vmlinux.bin 位于 0x100000（1M）处。然后执行 setup（可以理解为 boot loader 引导 setup）

5.setup 负责引导 linux 内核 vmlinux.bin

现在，让我们看看 setup 做了些什么。

首先，运行一下 file setup

它报告说这是一个 MS DOS 的可执行程序。看来，file 也有不正常工作的时候。不过有一点是肯定的。setup 不是普通的 elf 可执行程序。事实上，gcc 可以编译出各种格式的程序，具体可以运行 objdump -i，查看 ld（gcc 的链接器）支持的输出格式。其中有一个是 binary 格式。

从 start 开始，setup 做了很多操作，例如，如果是 zImage，则把它从 0x10000 拷贝到 0x1000，调用 bios 功能，查询硬件信息，然后放在内存中供将来的内核使用，然后建立临时的 idt 和 gdt，负责把 16 位实地址模式转化为 32 位段式寻址的保护模式。前面这些我们不关心。我们关心的是后者：

```
_
832         movw    $1, %ax                 # protected mode (PE) bit
833         lmsw    %ax                     # This is it!
```

正是以下指令开启了保护模式的大门。

最后，再临门一脚

对 bzImage 来说：

```
jmpi    0x100000,__BOOT_CS
```

对 zImage 来说：

```
jmpi    0x1000,__BOOT_CS
```

就大功告成了。

不过，且慢，由于当时 CS 寄存器还没有设置为__BOOT_CS，所以，尽管在保护模式下，也仍然不能用通常的方式访问大于 1M 的内存。

不过，x86 处理器提供了特殊的手段来访问大于 1M 的内存，那就是在指令前加前缀 0x66。由于跳转地址与内核大小相关（zImage 和 bzImage 不一样）所以用一个小技巧，即把该指令当作数据处理。在计算机看来，指令和数据是没什么区别的，只要 ip 寄存器指向内存中某地址，计算机就把地址中的数据当作指令来看待。

```
854         .byte 0x66, 0xea                # prefix + jmpi-opcode
855 code32: .long   0x1000                  # will be set to 0x100000
856                                         # for big kernels
857         .word   __BOOT_CS
```

我不仅发出感慨：看 setup.S 的代码真是一种痛苦的体验，并且有以下感悟：

1.为什么不用 C 写呢?

原因很简单，因为 setup 要尽量短小精悍，由于种种原因，它的大小不能超过 63.5K.

启动时内存分配如下：

```
0x00000~0x00400   BIOS 中断向量表
0x00400~0x01000
0x01000~0x10000
0x10000~0x8f000   用来存放 zImage 所以最多 508K
0x8f000~0x90000   引导程序的命令行参数以及 bios 查询的信息
0x90000~0x90200   MBR
0x90200~0xA0000   setup
0xA0000~0x100000  映射到 BIOS 和外设硬件等
```

0x100000~　　存放 bzImage(如果大于 508K)

2.如何引用变量?

在 setup.S 中，定义了不少全局变量。在编译生成 setup 文件时，链接参数

LDFLAGS_setup　:= -Ttext 0x0 -s --oformat binary -e begtext

意为 text 段的地址为 0，输出格式为 binary（而不是默认的 e1f32-i386），入口地址 begtext。

由于基本上处于 16 位实模式下，所以只要设置好 CS 等段寄存器就可以正确地寻址了。

现在我们跳到 vmlinux.bin 的开头(位于 0x1000 或者 0x100000)，也就是 startup_32()，执行相应代码。

3.helloworld，vmlinux，arch/i386/boot/compressed/vmlinux，setup,bootsect 的区别与联系。

这几个可执行文件都是由 gcc 编译生成。只是格式不一样。

其中，helloworld,vmlinux,arch/i386/boot/compressed/vmlinux 都是 e1f32_i386 格式的可执行文件。setup，bootsect 是 binary 格式的可执行文件，它们的区别在于

1).helloworld 是普通的 e1f32_i386 可执行文件。它的入口是_start。运行在用户态空间。变量的地址都是 32 位页式寻址的保护模式的地址，在用户态空间。由 shell 负责装载。

2).vmlinux 是未压缩的内核，它的入口是 startup_32(0x100000，线性地址)，运行在内核态空间，变量的地址是 32 位页式寻址的保护模式的地址，在内核态空间。由内核自解压后启动运行。

3).arch/i386/boot/compressed/vmlinux 是压缩后的内核，它的入口地址是 startup_32(0x100000，线性地址).运行在 32 位段式寻址的保护模式下，变量的地址是 32 位段式寻址的保护模式的地址。由 setup 启动运行。

4).setup 是装载内核的 binary 格式的辅助程序。它的入口地址是：begtext（偏移地址为 0。运行时需要把 cs 段寄存器设置为 0x9020）。运行在 16 位实地址模式下。变量的地址等于相对于代码段起始地址的偏移地址。由 boot loader 启动运行。

5).bootsect 是 MBR 上的引导程序，也为 binary 格式。它的入口地址是_start()，由于装载到 0x7c00 处，运行时需要把 cs 段寄存器设置为 0x7c0。运行在 16 位实地址模式下。变量地址等于相对于代码段起始地址的偏移地址。由 BIOS 启动运行。

问题出现了。startup_32 有两个，分别在这两个文件中:

arch/i386/boot/compressed/head.S

arch/i386/kernel/head.S

那究竟执行的是哪一个呢? 难道编译时不会报错么?

读核感悟-Linux 内核启动-内核解压缩

这得从 vmliux.bin 的产生过程说起。

从内核的生成过程来看内核的链接主要有三步:

第一步是把内核的源代码编译成.o 文件，然后链接，这一步，链接的是 arch/i386/kernel/head.S，生成的是 vmlinux。注意的是这里的所有变量地址都是 32 位页寻址方式的保护模式下的虚拟地址。通常在 3G 以上。

第二步，将 vmlinux objcopy 成 arch/i386/boot/compressed/vmlinux.bin，之后加以压缩，最后作为数据编译成 piggy.o。这时候，在编译器看来，piggy.o 里根本不存在什么 startup_32。

第三步，把 head.o,misc.o 和 piggy.o 链接生成 arch/i386/boot/compressed/vmlinux，这一步，链接的是 arch/i386/boot/compressed/head.S。这时 arch/i386/kernel/head.S 中的 startup_32 被压缩，作为一段普通的数据，而被编译器忽视了。注意这里的地址都是 32 位段寻址方式的保护模式下的线性地址。

自然，在这过程中，不可能会出现 startup_32 重定义的问题。

你可能会说：太 BT 了，平时谁会采用这种方式编译程序?

是啊，然而在内核还没启动的情况下，要高效地实现自解压，还有更好的方式么?

所以前面的问题就迎刃而解。setup 执行完毕，跳转到 vmlinux.bin 中的 startup_32()是 arch/i386/boot/compressed/head.S 中的 startup_32()这是一段自解压程序，过程和内核生成的过程正好相反。这时，CPU 处在 32 位段寻址方式的保护模式下，寻址范围从 1M 扩大到 4G。只是没有页表。

我们对具体的解压过程不感兴趣。

内核解压完毕。位于 0x100000 即 1M 处

最后，执行一条跳转指令，执行 0x100000 处的代码，即 startup_32()，这回是 arch/i386/kernel/head.S 中的 startup_32()代码

```
1jmp $(__BOOT_CS), $__PHYSICAL_START
```

读核感悟-Linux 内核启动-开启页面映射

在 setup 的帮助下，我们顺利地从 16 位实地址模式过渡到 32 位段式寻址的保护模式。又在 arch/i386/boot/compressed/head.S 的帮助下实现了内核的自解压，并且从 arch/i386/kernel/head.S 中的 startup_32 开始。现在在线性地址 0x100000(1M)处开始就是我们的解压后的内核了。而 startup_32()的地址恰好是 0x100000。由于还没有开启页面映射，所以必须引用变量的线性地址（即变量的虚拟地址-PAGE_OFFSET），带来了很多不便。所以下一步的任务，就是建立页表，开启页面映射了。我们不妨从 arch/i386/kernel/head.S 入手。

由于在 Linux 中，每个进程拥有一个页表，那么，第一个页表也应该有一个对应的进程。通常情况下，Linux 下通过 fork()系统调用，复制原有进程，来产生新进程。然而第一个进程该如何产生呢? 既然不能复制，那就只能像女娲造人一样，以全局变量的方式捏造一个出来。它就是 init_thread_union。传说中的 0 号进程，名叫 swapper。只要 swapper 进程运行起来，调用 start_kernel()，剩下的事就好办了。不过，现在离运行 swapper 进程还差得很远。关键的一步，我们还没有为该进程设置页表。

为了保持可移植性，Linux 采用了三级页表。不过 x86 处理器只使用两级页表。所以，我们需要一个页目录和很多个页表（最多达 1024 个页表），页目录和页表的大小均为 4k。swapper 的页目录的创建与该进程的创建思维类似，也是捏造一个页表，叫 swapper_pg_dir.

```
417 ENTRY(swapper_pg_dir)
418         .fill 1024,4,0
```

它的意思是从 swapper_pg_dir 开始，填充 1024 项，每项为 4 字节，值为 0，正好是 4K 一个页面。

页目录有了，接下去看页表。一个问题产生了。该映射几个页表呢? 尽管一个页目录最多能映射 1024 个页表，每个页表映射 4M 虚拟地址，所以总共可以映射 4G 虚拟地址空

间。但是，通常应用程序用不了这么多。最简单的想法是，够用就行。先映射用到的代码和数据。还有一个问题：如何映射呢？运行 cat /proc/$pid/maps 可以看到，用户态进程的地址映射是断断续续的，相当复杂。这是由于不同进程的用户空间相互独立。但是，由于所有进程共享内核态代码和数据，所以映射关系可以大大简化。既然内核态虚拟地址从 3G 开始，而内核代码和数据事实上是从物理地址 0x100000 开始，那么本着 KISS 原则，一切从简，加上 3G 就作为对应的虚拟地址好了。由此可见，对内核态代码和数据来说：虚拟地址=物理地址+PAGE_OFFSET(3G)

内核中有变量 pg0，表示对应的页表。建立页表的过程如下：

```
091 page_pde_offset = (__PAGE_OFFSET >> 20);
092
093         movl $(pg0 - __PAGE_OFFSET), %edi
094         movl $(swapper_pg_dir - __PAGE_OFFSET), %edx
095         movl $0x007, %eax                       /* 0x007 = PRESENT+RW+USER
*/
096 10:
097         leal 0x007(%edi),%ecx                   /* Create PDE entry */
098         movl %ecx,(%edx)                        /* Store identity PDE entry
*/
099         movl %ecx,page_pde_offset(%edx)         /* Store kernel PDE entry */
100         addl $4,%edx
101         movl $1024, %ecx
102 11:
103         stosl
104         addl $0x1000,%eax
105         loop 11b
106         /* End condition: we must map up to and including
INIT_MAP_BEYOND_END */
107         /* bytes beyond the end of our own page tables; the +0x007 is the
attribute bits */
108         leal (INIT_MAP_BEYOND_END+0x007)(%edi),%ebp
109         cmpl %ebp,%eax
110         jb 10b
111         movl %edi,(init_pg_tables_end - __PAGE_OFFSET)
```

用伪代码表示就是：

```
typedef unsigned int PTE;
PTE *pg=pg0;
PTE pte=0x007;
for(i=0;;i++){//把线性地址 i*4MB~(i+1)*4MB-1(用户空间地址)和 3G+i*4MB~3G+
(i+1)*4MB-1(内核空间地址)映射到物理地址 i*4MB~(i+1)*4MB-1
      swapper_pg_dir[i]=pg+0x007;
      swapper_pg_dir[i+page_pde_offset]=pg+0x007;
      for(j=0;j<1024;j++){
            pte+=0x1000;
            pg[i*1024+j]=pte;
```

```
        }
        if(pte>=((char*)pg+i*1024+j)*4+0x007+INIT_MAP_BEYOND_END)
        {
                init_pg_tables_end=pg+i*0x1000+j;
                break;
        }
}
```

大致意思是从 0 开始，把连续的线性地址映射到物理地址。这里的 0x007 是什么意思呢？由于每个页表项有 32 位，但其实只需保存物理地址的高 20 位就够了，所以剩下的低 12 位可以用来表示页的属性。0x007 正好表示 PRESENT+RW+USER（在内存中，可读写，用户页面，这样在用户态和内核态都可读写，从而实现平滑过渡）。

那么结束条件是什么呢？从代码中可知，当映射到当前所操作的页表项往下 INIT_MAP_BEYOND_END（128K）处映射结束。nm vmlinux|grep pg0 得 c0595000。据此可以计算总共映射了多少页（小学计算题:P）

所以映射了 2 个页表，映射地址从 0x0~0x2000-1，大小为 8M。

最后，关键时刻到来了：

```
183 /*
184  * Enable paging
185  */
186         movl $swapper_pg_dir-__PAGE_OFFSET,%eax
187         movl %eax,%cr3          /* set the page table pointer.. */
188         movl %cr0,%eax
189         orl $0x80000000,%eax
190         movl %eax,%cr0          /* ..and set paging (PG) bit */
```

开启页面映射后，可以直接引用内核中的所有变量了。不过离 start_kernel 还有点距离。要启动 swapper 进程，得首先设置内核堆栈。

```
193         /* Set up the stack pointer */
194         lss stack_start,%esp
```

然后设置中断向量表，看到久违的"call"了

```
215         call setup_idt
```

检查 CPU 类型

载入 gdt(原来的 gdt 是临时的)和 ldt

```
302         lgdt cpu_gdt_descr
303         lidt idt_descr
```

最后，调用 start_kernel

```
327         call start_kernel
```

到这一步，我们的目的地终于走到了。在摆脱了晦涩的汇编之后，接下去的代码,虽然与用户态程序相比，还有中断，同步等等的干扰，但相比较而言就好懂很多了。

读核感悟-Linux 内核启动-链接脚本

一般来说，用户是不需要关心 section 的具体位置的。在用户态，内核会解析 elf 可执行文件的各个 section，然后把它映射到虚拟地址空间。然而，在内核启动时，一切得从

零开始。很多在用户态下应用程序不需要操心的东西，例如映射 section 的任务不得不由内核自己来完成。上一篇感悟揭示了内核如何建立页表，并且把自身的一部分映射到虚拟地址。内核还要负责对 BSS 段（所有在代码中未定义的全局变量）的初始化（设置为 0），这就要求内核知道 section 的具体位置（否则如何知道该映射哪一部分呢？）

此外，在开启页面映射的过程中，我最为疑惑的是几个常量（页目录 swapper_pg_dir，页表 pg0 等等）是如何确定的。扩展一下。gcc 链接可执行文件时，是如何确定变量的地址的？按理说应该有某种途径（命令行参数或者文件）告诉链接器 ld 如何定位这些变量。最普通如 he11o wor1d。为什么_start 的地址是 0x80482e0?于是想到，我们需要一个文件来指定各个 section 的虚拟地址。在内核源代码里，还看到这个文件 arch/i386/kerne1/vm1inux.1ds.S。不像是普通的汇编文件。原来这就是 1inker scripts 链接器脚本。

在链接器脚本中，.表示当前 1ocation counter 地址计数器的值。默认为 0。

017 . = __KERNEL_START;

表示地址计数器从__KERNEL_START(0xc00100000)开始。

.text:{...}

表示.text section 包含了哪几个 section

031 . = ALIGN(16);

则表示对齐方式。

具体格式可以调用 info 1d 查看 Linker Scripts 一节。

链接器脚本指定了各个 section 的起始位置和结束位置。它还允许程序员在脚本中对变量进行赋值。这使内核可以通过__initca11_start 和__initca11_end 之类的变量获得段的起始地址和结束地址，从而对某些段进行操作。

根据链接器脚本，以及 nm vm1inux 的结果，内核中各个 section 的虚拟地址就很清楚了。以我的机子为例（粗略）：

地址分配

text section:

从_text:c0100000 A _text

到_etext:c0436573 A _etext

Exception tab1e

从__start___ex_tab1e:c0436580 A __start___ex_tab1e

到__stop___ex_tab1e:c04370b8 A __stop___ex_tab1e

RODATA read only section

.data writab1e section

.data_nosave section

从__nosave_begin:c050f000 A __nosave_begin

到__nosave_end:c050f000 A __nosave_end

.data.page_aligned section

.data.cache1ine_aligned section

.data.read_mostly section

.data.init_task section

init section

从__init_begin c0514000 A __init_begin

到__init_end c0540000 A __init_end

其中.initca11.init section:

从__initcall_start:c053b570 A __initcall_start
到__initcall_end:c053b8c0 A __initcall_end
BSS section
从__bss_start c0540000 A __bss_start
到__bss_end c0594c78 A __bss_stop
其中 swapper 进程的页表
从 c0540000 B swapper_pg_dir
到 c0541000
共一页
empty_zero_page
从 c0541000 B empty_zero_page
到 c0542000
共一页
pg0 页目录 0
从 c0595000 A pg0
到 init_pg_tables_end
.exitcall.exit
section
stab section

几个比较重要的 section:

bss section，存放在代码里未初始化的全局变量，最后初始化为 0。

init sections，所有只在初始化时调用的函数和变量，包括所有在内核启动时调用的函数，以及内核模块初始化时调用的函数。其中最特别的是.initcall.init section。通过__initcall_start 和__initcall_end，内核可以调用里面所有的函数。这些 section 在使用一次后就可以释放，从而节省内存。

读核感悟-伪装现场-系统调用参数

内核支配了整个计算机的硬件资源，好像一位独裁者，高高在上。他有时候必须像法官一样公正，有时候则必须像狐狸一样狡猾。伪装现场就是他的拿手好戏。

系统调用是很特别的函数，因为它里面实现了用户态到内核态的转换。应用程序要创建新进程，不可能在用户态直接调用 sys_fork()。这就需要内核为 sys_fork()伪装一下调用现场。

比如 fork()系统调用，它有一个简洁得不能再简洁的接口。不过它在内核中的对应函数中的声明却是:

```
asmlinkage int sys_fork(struct pt_regs regs)
```

如果有兴趣看早期的 Linux 源代码版本（0.95），它的声明是这样的:

```
int sys_fork(long ebx,long ecx,long edx,

long esi, long edi, long ebp, long eax, long ds,

long es, long fs, long gs, long orig_eax,

long eip,long cs,long eflags,long esp,long ss)
```

这些参数是如何来的呢。为什么用户态参数与内核态参数不一致?

又如clone(),vfork()这几个系统调用在用户态的参数个数和类型都不一样。但在内核态都致。

```
asmlinkage int sys_clone(struct pt_regs regs)
asmlinkage int sys_vfork(struct pt_regs regs)
```

接下去，我们可以看到，内核是多么巧妙地设置堆栈，让内核的函数感觉就好像上层函数在调用它一样。

我们很容易得知，这些参数是在系统调用进入内核时由int 0x80指令和SAVE_ALL宏把一些寄存器压入内核堆栈的。压入的寄存器数量和顺序是一致的，它们恰好与struct pt_regs一一对应。从这个角度讲，所有的系统调用获得的参数是形式是一样的。

```
026 struct pt_regs {

027         long ebx;

028         long ecx;

029         long edx;

030         long esi;

031         long edi;

032         long ebp;

033         long eax;

034         int  xds;

035         int  xes;

036         long orig_eax;

037         long eip;

038         int  xcs;

039         long eflags;

040         long esp;

041         int  xss;

042 };
```

其中xss到eip由int指令压入内核堆栈。orig_eax到ebx为ENTRY(system_call)压入堆

栈。orig_eax 为系统调用号。eax 作为返回值。ebp~ebx 作为系统调用的参数。

linux 的系统调用在内核中对应函数如 sys_fork()的声明前都有一个 asmlinage 的宏。它被定义为:

```
#define asmlinkage CPP_ASMLINKAGE __attribute__((regparm(0)))
```

也就是说，所有的参数都通过堆栈传递。注意，这里的堆栈传递已经在内核态了，与系统调用参数通过寄存器传递并不矛盾（那个是之前在用户态到内核态的切换过程中）。

这样就意味着我们只要巧妙地设置好堆栈，让 sys_fork()产生一种错觉，好像是上层的一个函数直接调用 sys_fork()一样。事实上 sys_fork()也的确可以在内核态直接调用。

我们又观察到:内核中_sysca110，_sysca111，_sysca112，。。。这些宏用来封装系统调用（只是内核自己用，glibc 不用这些宏而已。）传递参数的方式是这样的：参数 1~参数 6 正好放在 ebx ecx edx esi edi ebp 中，这恰好与 pt_regs 中的顺序相对应。为什么采用这样的一种顺序呢?

1.这与编译器的编译规则有关。标准的 C 函数编译时，总是从右往左依次把参数压入堆栈。执行到函数内部时，就根据这样的规则依次从堆栈中取出参数。所以在内核中也不例外。当应用程序执行系统调用，进入到内核态中的 ENTRY(system_ca11)调用

call *sys_call_table(,%eax,4)

时，所有的参数都在堆栈中准备就绪。

具体这些参数怎么理解，就取决于函数的定义了。

如果认为内核堆栈中放的是个结构体 pt_regs，就可以定义为

```
asmlinkage int sys_fork(struct pt_regs regs)
```

如果认为内核堆栈中放的是一个个整数，就可以定义为

```
asmlinkage int sys_fork(long ebx,long ecx,long edx,

long esi, long edi, long ebp, long eax, long ds,

long es, long fs, long gs, long orig_eax,

long eip,long cs,long eflags,long esp,long ss)
```

其它的系统调用也类似。

通过巧妙的构造堆栈，达到调用内核函数的目的。

除了系统调用，还有其它函数如 do_page_fau1t()等等，也是用类似的手段。

读核感悟-伪装现场-fork()系统调用

不仅进入系统调用时要伪装现场，fork 系统调用时返回时也需要伪装现场。因为是“无中生有”。

例如在 fork 创建新进程时，系统要保证新进程与旧进程一样，从相同的代码开始执行。比如:

```
#include<stdio.h>

#include<unistd.h>

int main()
```

```
{
        pid_t pid;
        if((pid=fork())>0)
        {
                printf("parent\n");
        }
        else if(pid==0)
        {
                printf("child\n");
        }
        else
                printf("error\n");
        return 0;
}
```

在 fork()处，也就是执行 call 指令的过程中，产生了一个新进程。

```
 80483f0:       e8 ef fe ff ff          call   80482e4 <fork@plt>
```

执行完 fork()后，新旧两个进程都执行相同的指令：

```
 80483f5:       89 45 fc                mov    %eax,0xfffffffc(%ebp)

 80483f8:       83 7d fc 00             cmpl   $0x0,0xfffffffc(%ebp)
```

给人的感觉是新进程和旧进程一样，也是从 main 函数开始执行，只是执行到 fork()处，返回值不一样而已。内核是如何做到的呢？当内核执行 fork()系统调用时，内核堆栈里保存有很多寄存器的值。其中包括了 eax 等几个通用寄存器。通过 SAVE_ALL 和 RESTORE_ALL 这两个宏，pt_regs 结构体里的变量与对应的寄存器建立了一一对应关系。只要修改 pt_regs 结构体里的变量，就可以达到修改系统调用结束后，进程执行那一刹那的寄存器值。

这样就简单了，因为 fork()时首先把 PCB 复制了一份(sys_fork()->do_fork()->copy_process->dup_task_struct)。所以内核堆栈 pt_regs 结构体里的值也跟着复制了一份。之后，在 copy_thread()中，把新进程的 pt_regs 的 eax，也就是新进程的返回值设置

为0，eip不变。这样当新进程返回到用户态时，就仿佛刚执行完fork()返回0。

这是伪装用户态的现场。还有内核态的现场呢?

copy_thread()中有以下代码:

```
468         childregs = (struct pt_regs *) ((unsigned long) childregs - 8);

469         *childregs = *regs;

470         childregs->eax = 0;

471         childregs->esp = esp;

472

473         p->thread.esp = (unsigned long) childregs;
```

当子进程被调度到时，子进程处于内核态。经过switch_to（）进程切换，eip设置为thread.eip，esp设置为thread.esp。 eip是ret_from_fork，esp是 childregs。伪装成刚执行完系统调用fork准备从内核态返回到用户态。

```
474         p->thread.esp0 = (unsigned long) (childregs+1);
```

thread.esp0在switch_to()->__switch_to()中，会通过load_esp0设置为tss_struct中的esp0。这个字段是在CPU运行状态变化时（如系统调用从用户态进入内核态）保存0级也就是内核态的堆栈指针。这样的话不是有两个地方有内核态堆栈指针了么?不过他们不矛盾。thread.esp在进程切换时使用。thread.esp0在CPU运行状态变化时使用。当子进程返回到用户态后又发生中断或者系统调用。这时使用的是 esp0，内核堆栈是空的。正好指向(unsigned long) (childregs+1)。

如何同步，我也不清楚。

```
475
```

```
476         p->thread.eip = (unsigned long) ret_from_fork;
```

p->thread中保存了进程切换所需的很多信息。包括内核态的esp，eip

伪装的现场，给人的感觉是新进程在内核态正好快执行到 ret_from_fork时由于中断等原因被抢占，所以在fork()完毕后可能进行的进程调度中，新进程可能被调度到，在执行完ret_from_fork中的一些代码，返回到用户态后，开始执行fork()之后的代码。

读核感悟-伪装现场-内核线程:

众所周知，内核中创建一个内核线程是通过kernel_thread实现的。声明如下:

```
int kernel_thread(int (*fn)(void *), void * arg, unsigned long flags);
```

我们知道，用户态创建线程调用clone()，如果要在内核态创建线程，首先想到的是在内核态调用clone()。这是可以的。比如在init内核线程中就直接在内核态调用execve，参数为/sbin/init等等。但是还是要小心翼翼。因为系统调用里会有很多参数要求是用户态的（一般在声明前有__user ），在调用一些内核函数时也会检查参数的界限，严格要求参数在用户态。一旦发现参数是在内核态，就立即返回出错。

所以kernel_thread采用了另外一种办法。

由于不是从用户态进入内核的，它需要制造一种现场，好像它是通过 clone 系统调用进入内核一样。方法是手动生成并设置一个 struct pt_regs,然后调用 do_fork()。但是怎样把线程的函数指针 fn,参数 arg 传进去呢？和 flags 不同，flags 可以作为 do_fork()的参数。但是 fn 正常情况下应该是在 clone()结束后才执行的。此外，线程总不能长生不老吧，所以执行完 fn()还要执行 exit()。

所以，我们希望内核线程在创建后，回到内核态（普通情况下是用户态）后，去调用 fn(arg),最后调用 exit()。而要去“遥控”内核线程在创建以后的事，只能通过设置 pt_regs 来实现了。

看 kernel_thread 的实现:

```
355         regs.ebx = (unsigned long) fn;

356         regs.edx = (unsigned long) arg;
```

这里设置了参数 fn,arg，当内核线程在创建以后，ebx 中放的是 fn,edx 中放的是 arg

```
358         regs.xds = __USER_DS;

359         regs.xes = __USER_DS;

360         regs.orig_eax = -1;

361         regs.eip = (unsigned long) kernel_thread_helper;
```

当内核线程在创建以后，执行的是 kernel_thread_helper 函数

```
362         regs.xcs = __KERNEL_CS;
```

当内核线程在创建以后，cs 寄存器的值表明当前仍然处于内核态。

```
363         regs.eflags = X86_EFLAGS_IF | X86_EFLAGS_SF | X86_EFLAGS_PF | 0x2;

364

365         /* Ok, create the new process.. */

366         return do_fork(flags | CLONE_VM | CLONE_UNTRACED, 0, &regs, 0, NULL,
NULL);
```

看来 kernel_thread_helper 就是我们想要的东西了。

```
336 __asm__(".section .text\n"

337         ".align 4\n"

338         "kernel_thread_helper:\n\t"

339         "movl %edx,%eax\n\t"

340         "pushl %edx\n\t"
```

```
341         "call *%ebx\n\t"

342         "push1 %eax\n\t"

343         "call do_exit\n"

344         ".previous");
```

首先把 edx 保存到 eax(不明白为什么这么做，因为调用 fn 后返回值就把 eax 覆盖掉了)把 edx(其实就是参数 arg)压入堆栈，然后调用 ebx(也就是 fn)。最后调用 do_exit。kernel_thread_helper 是不返回的。

这里，内核通过巧妙设置 pt_regs，在没有用户进程的情况下，在内核态创建了线程。

读核感悟-伪装现场-信号通信

信号是进程之间通信的一种方式。它包括 3 部分操作:

1.设置信号处理函数。系统调用 signal。内核调用 sys_signal()，设置当前进程对某信号的处理函数。

2.发送信号.系统调用 kill。内核调用 sys_kill()。向目标进程发送信号。

3.接收并处理信号。目标进程调用 do_signal()处理信号。

从用户态的角度看，目标进程在执行用户态的代码时突然“中断”，转而去执行对应的信号处理函数（同样在用户态）。等到信号处理函数执行完后，又从原来被中断的代码开始执行。

如何达到这样的效果呢? 由前面的几种内核的伪装现场的手段，我们可以猜出它这次使用的手段。比如，要让目标进程执行信号处理函数，在内核态中当然不可能直接调用，但是可以通过设置 pt_regs 中的 eip 来达到这种效果。但是，要使目标进程在执行完信号处理函数后，又恢复到被中断的现场继续执行，那得花些技巧。不过，不外乎设置堆栈。这一次还包括了用户态堆栈。由于恢复的任务比较艰巨，系统干脆提供了一个系统调用 sigreturn。

既然内核希望用户在执行完信号处理函数后，调用 sigreturn。接下去的思路就比较简单了。就是先把用户态的 eip 设置为 signal_handler（通过修改 pt_regs 中的 eip 来实现），然后把堆栈中的返回地址改成调用 sigreturn 的一段代码的入口（当然原来的返回地址也还是要保存的）并且把相关参数“压入”用户态堆栈。

这样，在源进程发送信号后不久，目标进程被调度到，然后执行到 do_signal。对信号一一作处理。调用顺序:

do_signal()->handle_signal()->setup_rt_frame()

用来设置用户态堆栈。

我们看看这个函数做了些啥?

```
447         frame = get_sigframe(ka, regs, sizeof(*frame));
```

struct rt_sigframe __user *frame 是内核在用户态的堆栈上新分配的一个数据结构。

```
011 struct rt_sigframe

012 {
```

```
013         char *pretcode;

014         int sig;

015         struct siginfo *pinfo;

016         void *puc;

017         struct siginfo info;

018         struct ucontext uc;

019         struct _fpstate fpstate;

020         char retcode[8];

021 };
```

图示:

高地址

```
--------------    用户进程原堆栈底部

--------------    用户进程原堆栈顶部
frame->retcode          frame 底部

frame->pretcode         返回地址: 从 signal handler 返回后跳转的地址
--------------    frame 顶部: 用户进程新堆栈顶部
```

低地址

里面保存了大量用户态进程的上下文信息。尤其是 pretcode, 现在位于用户进程的新堆栈的顶部。

接下去开始设置 frame

```
458         err |= __put_user(usig, &frame->sig);

459         err |= __put_user(&frame->info, &frame->pinfo);

460         err |= __put_user(&frame->uc, &frame->puc);

461         err |= copy_siginfo_to_user(&frame->info, info);

462         if (err)

463                 goto give_sigsegv;

464
```

```
465         /* Create the ucontext.  */
466         err |= __put_user(0, &frame->uc.uc_flags);
467         err |= __put_user(0, &frame->uc.uc_link);
468         err |= __put_user(current->sas_ss_sp, &frame->uc.uc_stack.ss_sp);
469         err |= __put_user(sas_ss_flags(regs->esp),
470                           &frame->uc.uc_stack.ss_flags);
471         err |= __put_user(current->sas_ss_size, &frame-
>uc.uc_stack.ss_size);
472         err |= setup_sigcontext(&frame->uc.uc_mcontext, &frame->fpstate,
473                                 regs, set->sig[0]);
474         err |= __copy_to_user(&frame->uc.uc_sigmask, set, sizeof(*set));
```

当用户进程根据 pt_regs 中设置好的 eip 执行 signal handler。执行完毕后就会把 frame->pretcode 作为返回地址。（这一点 2.6.13 的内核与 2.4 的不同，后者是把指针指向 retcode,其实仍然是调用 sigreturn 的代码。）

```
478         /* Set up to return from userspace.  */
479         restorer = &__kernel_rt_sigreturn;
480         if (ka->sa.sa_flags & SA_RESTORER)
481                 restorer = ka->sa.sa_restorer;
482         err |= __put_user(restorer, &frame->pretcode);
```

这里的&__kernel_rt_sigreturn 就是内核设置的负责信号处理“善后”工作的代码入口。定义在 arch/i386/kernel/vsyscall-sigreturn.S

```
nm /usr/src/linux/vmlinux|grep _kernel_rt_sigreturn
```

得，它的值是_kernel_rt_sigreturn

&__kernel_rt_sigreturn 的值应该是内核态的。

问题来了。frame->pretcode 是作为进程在用户态执行的代码（事实上，从执行 signal handler 开始，进程一直处于用户态）。它怎么能访问内核态的代码呢？这不是会

段错误么?

这里涉及到 PIII 中用 sysenter 来代替系统调用的 int 0x80 的问题。大概就是内核允许一部分代码给用户态进程访问。

例如:

cat /proc/$pid/maps 可以看到:

```
ffffe000-fffff000 ---p 00000000 00:00 0          [vdso]
```

ldd 一个应用程序也可以看到:

```
linux-gate.so.1 =>  (0xffffe000)
```

在 arch/i386/kernel/中可以看到两个文件:

```
vsyscall-sysenter.so
vsyscall-int80.so
```

也就是说, __kernel_rt_sigreturn 这段代码是链接在两个动态链接文件中。而不是 vmlinux 这个内核文件中。

具体是如何做到的, 就不展开说了。

总之, 在执行完 signal handler 后, 进程将跳转到__kernel_rt_sigreturn。

```
021 __kernel_sigreturn:

022 .LSTART_sigreturn:

023         popl %eax               /* XXX does this mean it needs unwind info?
*/

024         movl $__NR_sigreturn, %eax

025         int $0x80
```

实际上调用的是 sigreturn 系统调用。该系统调用会根据 frame 里的信息, 把堆栈恢复到处理信号之前的状态。所以这段代码是不返回的。然后, 用户进程就像什么事也没发生, 继续照常运行。

这里, 内核通过设置用户态堆栈的手段, 达到了打断用户态进程的运行, 转而调用 signal handler 的目的。手段不可谓不高明。

读核感悟-kbuild 系统-内核模块的编译

Linux 内核是一种单体内核, 但是通过动态加载模块的方式, 使它的开发非常灵活方便。那么, 它是如何编译内核的呢? 我们可以通过分析它的 Makefile 入手。以下是一个简单的 hello 内核模块的 Makefile.

```
ifneq ($(KERNELRELEASE),)
        obj-m:=hello.o
else
KERNELDIR:=/lib/modules/$(shell uname -r)/build
PWD:=$(shell pwd)
default:
        $(MAKE) -C $(KERNELDIR)  M=$(PWD) modules
clean:
```

```
        rm -rf *.o *.mod.c *.mod.o *.ko
endif
```

当我们写完一个 hello 模块，只要使用以上的 makefile。然后 make 一下就行。

假设我们把 hello 模块的源代码放在/home/study/prog/mod/hello/下。

当我们在这个目录运行 make 时，make 是怎么执行的呢?

LDD3 第二章第四节“编译和装载”中只是简略地说到该 Makefile 被执行了两次，但是具体过程是如何的呢?

首先，由于 make 后面没有目标，所以 make 会在 Makefile 中的第一个不是以.开头的目标作为默认的目标执行。于是 default 成为 make 的目标。make 会执行$(MAKE) -C $(KERNELDIR) M=$(PWD) modules

。

shell 是 make 内部的函数,假设当前内核版本是 2.6.13-study,所以$(shell uname -r)的结果是 2.6.13-study

这里，实际运行的是

make -C /lib/modules/2.6.13-study/build M=/home/study/prog/mod/hello/ modules

/lib/modules/2.6.13-study/build 是一个指向内核源代码/usr/src/linux 的符号链接。

可见，make 执行了两次。第一次执行时是读 hello 模块的源代码所在目录/home/study/prog/mod/hello/下的 Makefile。第二次执行时是执行/usr/src/linux/下的 Makefile 时.

但是还是有不少令人困惑的问题:

1.这个 KERNELRELEASE 也很令人困惑，它是什么呢?

在/home/study/prog/mod/hello/Makefile 中是没有定义这个变量的，所以起作用的是 else...endif 这一段。不过，如果把 hello 模块移动到内核源代码中。例如放到/usr/src/linux/driver/中，KERNELRELEASE 就有定义了。

在/usr/src/linux/Makefile 中有

```
162    KERNELRELEASE=$(VERSION).$(PATCHLEVEL).$(SUBLEVEL)$(EXTRAVERSION)$(LOCALVERSION)
```

这时候，hello 模块也不再是单独用 make 编译，而是在内核中用 make modules 进行编译。

用这种方式，该 Makefile 在单独编译和作为内核一部分编译时都能正常工作。

2.这个 obj-m := hello.o 什么时候会执行到呢?

在执行:

```
make -C /lib/modules/2.6.13-study/build M=/home/study/prog/mod/hello/ modules
```

时，make 去/usr/src/linux/Makefile 中寻找目标 modules:

```
862 .PHONY: modules

863 modules: $(vmlinux-dirs) $(if $(KBUILD_BUILTIN),vmlinux)

864         @echo '  Building modules, stage 2.';

865         $(Q)$(MAKE) -rR -f $(srctree)/scripts/Makefile.modpost
```

可以看出，分两个 stage:

1.编译出 hello.o 文件。

2.生成 hello.mod.o hello.ko

在这过程中，会调用

make -f scripts/Makefile.build obj=/home/study/prog/mod/hello

而在 scripts/Makefile.build 会包含很多文件:

```
011 -include .config

012

013 include $(if $(wildcard $(obj)/Kbuild), $(obj)/Kbuild, $(obj)/Makefile)
```

其中就有/home/study/prog/mod/hello/Makefile

这时 KERNELRELEASE 已经存在。

所以执行的是:

obj-m:=hello.o

关于 make modules 的更详细的过程可以在 scripts/Makefile.modpost 文件的注释中找到。如果想查看 make 的整个执行过程，可以运行 make -n。

由此可见，内核的 Kbuild 系统庞大而复杂。

读核感悟-kbuild 系统-编译到内核和编译成模块的区别

代码编译到内核和编译成模块在代码中有什么区别呢?

从模块的代码中看是一样的。入口函数都是 module_init(fun)，但是代码中的条件编译会使宏 module_init()在编译到内核和编译成模块的情况下替换成不同的代码。

include/linux/init.h 中可知

```
#ifndef MODULE

...

#define module_init(x)  __initcall(x);

...

#else /* MODULE */

...

/* Each module must use one module_init(), or one no_module_init */
```

```
#define module_init(initfn)                                    \

        static inline initcall_t __inittest(void)              \

        { return initfn; }                                     \

        int init_module(void) __attribute__((alias(#initfn)));

...

#endif
```

当代码编译成模块时，会定义 MODULE 宏，否则不会。因为在/usr/src/linux/Makefile 中可以看到

```
336 MODFLAGS        = -DMODULE
337 CFLAGS_MODULE   = $(MODFLAGS)

338 AFLAGS_MODULE   = $(MODFLAGS)
```

这两个变量又被 export 成为全局变量。所以可以知道，在编译成模块时，会有 MODULE 这个宏。

由以下代码可以知道

```
#define __initcall(fn) device_initcall(fn)
#define device_initcall(fn)             __define_initcall("6",fn)

085 #define __define_initcall(level,fn) \

086         static initcall_t __initcall_##fn __attribute_used__ \

087         __attribute__((__section__(".initcall" level ".init"))) = fn
```

前者实际上是编译入内核中的.initcall6.init 这个 section
而在
arch/i386/kernel/vmlinux.lds.S 中可以知道:

```
083   __initcall_start = .;

084   .initcall.init : AT(ADDR(.initcall.init) - LOAD_OFFSET) {

085         *(.initcall1.init)
```

```
086         *(.initcall2.init)

087         *(.initcall3.init)

088         *(.initcall4.init)

089         *(.initcall5.init)

090         *(.initcall6.init)

091         *(.initcall7.init)

092     }

093     __initcall_end = .;
```

arch/i386/kernel/vmlinux.lds.S

.initcall6.init 是.initcall1.init 的一部分
执行顺序:
start_kernel->rest_init
系统启动后在 rest_init 中会创建 init 内核线程
init->do_basic_setup->do_initcalls
do_initcalls 中会把.initcall1.init 中的函数依次执行一遍

```
for (call = __initcall_start; call < __initcall_end; call++) {

        ...

        (*call)();

        ...

}
```

于是执行了 module_init(fn)函数

读核感悟-kbuild 系统-make bzImage 的过程

从以上例子中可以看到，内核的编译系统 kbuild 是个很庞大的系统。但是，它所使用的 make 和我们平时用的 make 是一模一样的。kbuild 只是通过预定义一些变量（obj-m,obj-y 等等）和目标(bzImage ,menuconfig 等等)，使内核的编译和扩展变得十分方便。我们不妨 yy 一下 kbuild 的一些功能:

1.考虑到 Linux 能够方便地移植到各个硬件平台，kbuild 也必须很容易添加对某个新的平台的支持，同时上层的 Makefile 不需要做大的改动。

2.Linux 下有众多驱动设备。它们的 Makefile 希望能够尽可能简洁。简洁到只要指定要编译的.o 文件就行。（这方面 kbuild 定义了很多有用的变量如 obj-m obj-y,<module>-

objs 等等，用户只要为这些变量赋值，kbuild 会自动把代码编译到内核或者编译成模块）

3.要有方便的可定制性。很多参数可以让用户指定。这方面 kbuild 也提供了大量的变量如 EXTRA_CFLAGS，用户如果想 include 自己的头文件或者加其它编译参数，只要设置一下 EXTRA_CFLAGS 就可以。

4.有能力递归地调用 Makefile。因为内核是一个庞大的软件。它的源代码的目录层次很深。要提供一种简洁的机制，使上层的 Makefile 能方便地调用下层的 Makefile。在这过程中，面向对象的思想也许值得借鉴。

5.在配置内核时，要提供友好的用户界面。这方面 kbuild 也提供了不少工具，如常用的 make menuconfig 等等。

我们完全可以把 kbuild 想象成一个类库，它为普通的内核开发人员提供了接口（obj-m obj-y EXTRA_CFLAGS 等等），为用户提供了定制工具（make menuconfig）

如果想了解 kbuild 的使用方法，可以参阅源代码自带的文档：

Documentation/kbuild/makefiles.txt

Documentation/kbuild/modules.txt

一般情况下是不需要知道具体的编译顺序的。除了在个别情况下，如 do_initcalls()中就和函数在.initcall.init section 中的顺序有关。不过喜欢寻根究底的我，还是想理一下编译内核时几个常用的命令，如 make bzImage,make menuconfig 等等，进而了解 kbuild 的架构。先看 make bzImage 吧。

它的大概脉络是怎样的呢？可以用以下命令查看。

make -n bzImage

如果嫌内容太多，可以过滤掉多余的信息：

make -n bzImage | grep “make -f”

可以猜到：

先作一些准备工作

```
make -f scripts/Makefile.build obj=scripts/basic
```

然后依次递归地调用源代码中的 Makefile

```
make -f scripts/Makefile.build obj=init

make -f scripts/Makefile.build obj=usr

make -f scripts/Makefile.build obj=arch/i386/kernel

make -f scripts/Makefile.build obj=arch/i386/kernel/acpi

make -f scripts/Makefile.build obj=arch/i386/kernel/cpu

make -f scripts/Makefile.build obj=arch/i386/kernel/cpu/cpufreq

make -f scripts/Makefile.build obj=arch/i386/kernel/cpu/mcheck

make -f scripts/Makefile.build obj=arch/i386/kernel/cpu/mtrr

make -f scripts/Makefile.build obj=arch/i386/kernel/timers
```

。。。
最后压缩内核，生成bzImage
```
make -f scripts/Makefile.build obj=arch/i386/boot arch/i386/boot/bzImage

make -f scripts/Makefile.build obj=arch/i386/boot/compressed
IMAGE_OFFSET=0x100000 arch/i386/boot/compressed/vmlinux
```

好，我们从头开始。找make bzImage的入口：
第一反应，自然是在/usr/src/linux/Makefile中找
```
bzImage:
```
...
可惜没找到。
不过没关系，用lxr搜索一下，可知bzImage定义在arch/i386/Makefile，所以可以猜测，该makefile一定是被include了。果然，在/usr/src/linux/Makefile中有：
```
447 include $(srctree)/arch/$(ARCH)/Makefile
```
又因为在arch/i386/Makefile中定义有
```
141 zImage bzImage: vmlinux
142         $(Q)$(MAKE) $(build)=$(boot) $(KBUILD_IMAGE)
```

其中这个$(build)定义在/usr/src/linux/Makefile中
```
1335 build := -f $(if $(KBUILD_SRC),$(srctree)/)scripts/Makefile.build obj
```

我们在之前查看make -n bzImage信息和之后会经常看到。我们会发现kbuild通常不会直接去调用某个目录下的Makefile，而是让该目录作为scripts/Makefile.build 的参数。scripts/Makefile.build 会对该目录下的Makefile中的内容(主要是obj-m和obj-y等等)进行处理。由此看来 scripts/Makefile.build这个文件很重要。看看它做了什么：
由于scripts/Makefile.build后面没跟目标，所以默认为第一个目标：
```
007 .PHONY: __build

008 __build:

009

010 # Read .config if it exist, otherwise ignore

011 -include .config

012

013 include $(if $(wildcard $(obj)/Kbuild), $(obj)/Kbuild, $(obj)/Makefile)

014
```

```
015 include scripts/Makefile.lib
```

这里可以看到，scripts/Makefile.build 执行时会 include .config 文件。.config 是 make menuconfig 后生成的内核配置文件。
里面有如下语句：

```
CONFIG_ACPI_THERMAL=y

CONFIG_ACPI_ASUS=m

CONFIG_ACPI_IBM=m
```

。。。
以前我一直对它的格式表示奇怪，现在很清楚了，它们是作为 makefile 的一部分，通过读取 CONFIG_XXX 的值就可以知道他们是作为模块还是作为内核的一部分而编译的。
此外，还包含了$(obj)/Makefile。这就是通过在 make 时传递目录名$(obj)间接调用 Makefile 的手法。对于内核普通代码所对应的 Makefile 而言，里面只是对 obj-m obj-y,<module>-objs 等变量进行赋值操作。

接下去是 include scripts/Makefile.lib
。正如它的文件名所示，这类似于一个库文件。它负责对 obj-m obj-y,<module>-objs 等变量进行加工处理。从中提取出 subdir-ym 等变量，这是个很重要的变量，记录了需要递归调用的子目录。以后递归调用 Makefile 全靠它了。这里也充分体现了 GNU make 对字符串进行操作的强大功能。

回到 Makefile.build。这时，重要变量$(builtin-target)，$(subdir-ym)等都已经计算完毕。开始列依赖关系和具体操作了。

```
079 __build: $(if $(KBUILD_BUILTIN),$(builtin-target) $(lib-target) $(extra-
y)) \

080           $(if $(KBUILD_MODULES),$(obj-m)) \

081           $(subdir-ym) $(always)

082          @:
```

$(builtin-target)是指当前目录下的目标文件，即$(obj)/built-in.o
如前文所说，$(subdir-ym)用来递归调用子目录的 Makefile

```
306 # Descending

307 #
--------------------------------------------------------------------------------
```

```
308

309 .PHONY: $(subdir-ym)

310 $(subdir-ym):

311         $(Q)$(MAKE) $(build)=$@
```

通过这种方式，实现了对某个目录及其子目录的编译。

分析完 Makefile.build，回过头来再看 bzImage.从 arch/i386/Makefile 中可以看到，bzImage 是在 vmlinux 基础上加以压缩拼接而成。从 vmlinux 到 bzImage 的过程在《读核感悟-Linux 内核启动-内核的生成》中已经有介绍。现在看看 vmlinux 是如何生成的。
见/usr/src/linux/Makefile

```
728 vmlinux: $(vmlinux-lds) $(vmlinux-init) $(vmlinux-main) $(kallsyms.o) FORCE

729         $(call if_changed_rule,vmlinux__)
```

```
611 vmlinux-init := $(head-y) $(init-y)

612 vmlinux-main := $(core-y) $(libs-y) $(drivers-y) $(net-y)

613 vmlinux-all  := $(vmlinux-init) $(vmlinux-main)

614 vmlinux-lds  := arch/$(ARCH)/kernel/vmlinux.lds
```

vmlinux 所依赖的目标$(vmlinux-lds) 是对 arch/i386/kernel/vmlinux.lds.S 进行预处理的结果：arch/i386/kernel/vmlinux.lds ,其它的依赖关系也都可以在/usr/src/linux/Makefile 中查到。

所以，当用户在源代码目录下执行 make bzImage。make 会检查 bzImage 的依赖目标，然后不停地递归调用各个 Makefile，最终生成一个 bzImage 文件。

如果我们换个角度，还可以归纳出不少有趣的东西。如果把 make 看成是一种脚本语言，那么 Makefile 就是代码。make 就是解释器。make 里也有函数，也有变量。通过定义目标，可以实现类似于函数的效果。而目标之间的依赖关系则类似于函数内部再调用其它函数。

如果我们考虑变量的作用域，还可以归纳出以下几点：

1.Makefile 内部定义的变量作用域只限于那个 Makefile 中，如 obj-m。

2.要使变量的作用域扩展到整个 make 命令的执行过程（包括递归调用的其它 Makefile），需要使用 export 命令。

调用 Makefile 的方式也有很多种：

1.一种是隐式调用，如运行 make，它会自动在当前目录寻找 Makefile 等。
2.一种是显式调用，如用 make -f 指定。
3.一种是用 include 来调用。

读核感悟-kbuild 系统-make menuconfig

理解了 make bzImage 的过程，理解了整个 kbuild 的结构和运行机制，make menuconfig 的过程就很容易理解了。

先看/usr/src/linux/Makefile。可以找到：

```
452 %config: scripts_basic outputmakefile FORCE

453         $(Q)$(MAKE) $(build)=scripts/kconfig $@
```

%config 是通配符，所有以 config 结尾的目标（menuconfig xconfig gconfig）都采用这个规则。所以 make menuconfig 在进行一些准备工作如 make scripts_basic 等操作后，最终会运行

(Q)(MAKE) $(build)=scripts/kconfig menuconfig

其中$@指要生成的目标文件，这里指伪目标 menuconfig。

接下去调用的是： scripts/kconfig/Makefile

```
013 menuconfig: $(obj)/mconf

014         $(Q)$(MAKE) $(build)=scripts/lxdialog

015         $< arch/$(ARCH)/Kconfig

082 hostprogs-y     := conf mconf qconf gconf kxgettext

083 conf-objs       := conf.o  zconf.tab.o

084 mconf-objs      := mconf.o zconf.tab.o

085 kxgettext-objs  := kxgettext.o zconf.tab.o
```

这里可以看到最终执行的是 scripts/kconfig/mconf arch/i386/Kconfig 其中$<代表“起因”，也就是 scripts/kconfig/mconf

该程序并不属于内核，而是一个用户态程序。Linux 源代码中这一类程序还有很多。如在 scripts/kconfig/目录下就有 mconf,gconf,qconf 等等。它们用来执行内核的配置工作。

又如在 arch/i386/boot/tools/中有个可执行程序叫 build，它用来把 bootsect(引导扇区），setup（辅助程序）和 vmlinux.bin(压缩内核)拼接成 bzImage。

scripts/kconfig/mconf 这个程序采用了 ncurses 类库。这是一个在文本界面下进行画图操作类库。由于要适应不同平台，源代码中的 mconf 不是预编译好的 elf 可执行文件，而是在使用时才去编译生成。这使用户在运行 make menuconfig 时要依赖 ncurses 的开发

包。

arch/i386/Kconfig，准确地说是各个 Kconfig 文件记录了各个内核配置的选项。我们在 make menuconfig 或者 make xconfig 时显示的菜单项和帮助信息，都是从这个文件中读出来的。

到此，我们对平时使用的 make menuconfig 命令的执行流程应该有了一个大概的印象了吧。

读核感悟-文件系统-用 C 来实现面向对象

电脑用户跟文件系统打交道是非常频繁的。不过文件系统在结构上应该属于哪一层呢？它和设备驱动的关系如何？不同的系统有不同的理解。Linux 把文件系统放在设备驱动之上，访问普通文件时，系统先执行文件系统的代码，把文件指针的偏移量映射为块设备的偏移量，然后调用驱动程序的代码。另一方面，Linux 把设备也看作是一种特殊的文件。这有助于简化接口，对程序员来说是很方便的。

我们可以把接口的观点扩展开来。对于应用程序来说，类库是一种接口；对于用户态程序来说，系统调用是一种接口；而对内核的一些模块（如文件系统，驱动程序）来说，内核也提供了很多接口；对于底层的软件开发者来说，处理器的指令集可以看作是接口；对于一辆奔驰汽车来说，各个零部件的规格说明书也可以算是一种接口。。。接口的出现是社会分工的需要，这有力地提高了开发效率。这使一部分人能够专注于编写简洁，高效和功能强大的接口，而另一部分人则专注于具体的需求。当然，在提高开发效率的同时，也会使人局限于某一个小领域。像 Bill Joy（前任 Sun 的首席科学家，当年在 Berkeley 时主持开发了最早版本的 BSD。还主持设计了 sparc 处理器）这类软硬通吃的计算机牛人将会越来越少。这也可以算是社会分工带来的代价吧。

作为一个开放的系统。Linux 也提供了文件系统的接口。这使它能够方便地支持 ext2,fat32,reiserfs 等很多种文件系统。最理想的状况是选择一个 wizard，然后选择和填充若干属性，一路地 Next 下去，最后点 OK，一个基本的文件系统就诞生了。接下去要做的只是实现一下里面的一些函数而已。当然，现在还没有这样的 wizard，不过，基本思路是一致的，即把所有的文件系统中的公共接口抽象出来，让具体的文件系统去实现。这也是面向对象的思想。关于具体的接口，这方面的书已经讲得非常详细了，这里讲一讲如何用 C 语言来实现面向对象。

对

不过问题也出来了：

1.根分区对应的设备是/dev/sda1，也就是说，要挂载根分区，就要读/dev/sda1，可是，/dev/sda1 是保存在根分区上的一个文件，也就是说，要读/dev/sda1，就要挂载根分区，这不是死锁了么？当然，解决办法是有的。关键在于/dev/sda1 这个设备文件本身的信息不多，主要包括设备类型，主设备号与次设备号，这样，我们可以
windows 则相反，是把设备驱动放在文件系统之上。

读核感悟-设计模式-用 C 来实现虚函数表和多态

内核作为一个庞大的系统，要保证扩展性和可维护性，里面用到了大量设计模式。说到设计模式就很容易联想到面向对象，联想到 C++。那么，内核为什么不使用 C++呢。我想是出于效率和平台可移植性的考虑。C++编译器实现起来比 C 复杂多了，想像一下很多编译器都不是 100%地支持 C++标准。所以在一些非主流的硬件上，不一定能找到对应的 C++编译器。

虽然内核使用的是C，但是，内核仍然采用了面向对象的思想。尤其是在文件系统和设备驱动这一块。在这些地方，集中了大量的结构体如file,dentry,inode,cdev,block_device,pci_dev,usb_device,file_operations等等以及大量的函数和宏。我们的目标是理清它们之间的关系，最好能搞清楚为什么要这么设计。如果用面向对象的观点来阅读源代码，很多地方就显得清晰易懂了。好，让我们跟着问题走：

问题一：用C如何实现C++中常见的类，成员函数，虚函数表，多态，继承，封装等等特性呢?

稍加整理，我们可以发现，结构体主要可以分成两类。一类定义了很多数据成员，如struct file.一类定义了很多函数指针，如struct file_operations。并且这两者是一一对应的。其中，file_operations可以看作是接口。file中可以看作是类，而里面的成员变量f_op可以看作是虚函数表指针。只是没有C++的访问权限控制机制罢了。与此类似的还有dentry和dentry_operations，inode和inode_operations。这样，我们用C的结构体和函数指针模拟出了类，成员函数，虚函数表。你可能觉得这样很不直观。是的，很不直观，不过谁叫我们用的是C呢? 还有更不直观的，下面要介绍的模板就是。

这里可以看出，类与接口的区别。类包括了数据和成员函数，也就是“实现”，而接口只是一组操作的集合，类似于抽象类。接口的作用是屏蔽内部细节，方便上层的开发者开发一个通用的模型。对下层的开发者而言，只要实现接口的功能就行。在C++里是不区别类与接口的。而在Java里则在类之外还提供了接口的功能。扯远了，打住。

虚函数表有了，那么，要实现多态，就要实现动态绑定。我们知道，在C++中，编译器会根据类声明中的virtual关键字为每个类初始化一张虚函数表，然后，在每个对象构造的时候，把对象中的一个指针指向对应的虚函数表。现在C编译器宣布它撂担子不管这事了。所以这些辛苦活只能让我们自己来干喽~。以inode为例。假设文件系统是ext2。我们要读的是一个普通的ext2文件。

sys_open()->filp_open()->open_namei()->path_lookup()->link_path_walk()->__link_path_walk()->open_namei()->path_lookup()->do_lookup()->real_lookup()->ext2_lookup()->ext2_read_inode()

最后，在ext2_read_inode()中，它会根据inode类型把i_op设置成相应的inode_operations。这一步相当于C++的构造函数中的动态绑定，即根据inode类型来决定对应的操作。通过这种方式实现了多态。

虽然多态很好地实现了抽象，方便了软件的扩展，但是我得说它给我的代码阅读带来了很大的不便。跟踪函数时经常到了有函数指针（类似于虚函数）时就中断了，很难知道它究竟调用的是哪个函数。

读核感悟-设计模式-用C来实现继承和模板

多态实现了，封装呢? 基本上，C的结构体是不设防的，谁都可以访问。从这一点来看，C很难实现封装。尽管C中有static关键字，可以保证函数和变量的作用仅限于本文件，尽管内核可以通过控制导出符号表（EXPORT_SYMBOL）来控制提供给下层模块的函数和变量，但这些与C++中的封装相去甚远。好在内核的原则是“相信内核不会自己伤害自己”。所以就不苛求啦。

那么继承呢? 这个也基本上很难。不过我们可以通过组合来模拟“继承”。ext2_inode_info是ext2的inode在内存中的结构体(注意，不是ext2_inode,这个是在硬盘上)，有个成员变量是struct inode*，指向inode。使用时，用EXT2_I()宏来实现类型的down_cast。这样很不直观。当然我们也可以理解为ext2_inode_info与inode是两

个完全不同的结构体，ext2_inode_info 包含了 inode。但是这样的理解显然更简单轻巧：ext2_inode_info 继承了 inode，并且加了自己的私有成员变量。

这是一种实现继承，ext2_inode_info 实实在在地继承了 inode 的全部“财产”- 成员变量，还有一种接口继承，也使用得非常广泛。如 ext2_dir_inode_operations(定义了对 ext2 的目录节点的操作)对 inode_operations 的继承。与通常理解的“继承”不同，在 C 中并没有生成新的结构体，而只是定义了一个 inode_operations 类型的变量。这种继承只是把里面的函数指针加以实现。这种继承很辛苦，因为接口除了声明，什么也没做。

有了父类子类，自然也少不了类型之间的转换，以及运行中的类型识别(RTTI)。由于内核中只是用组合来模拟继承关系，即用子类包含父类的方式，所以从子类转换到父类就很方便。如子类是 struct Derive d 里有个成员变量 struct Base *b，从子类转换到父类只要 d->b 就行，但是从父类 down cast，向下类型转换到子类就比较麻烦了。然而这种情况非常常见，如设备驱动接口 file_operations 中

```
int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);
```

如果我们开发一个叫 scull 的字符设备，通常要定义自己的结构体 scull_dev，此外还要继承 cdev（表现为组合）.可是 ioctl 接口里没有 scull_dev!幸好 inode->i_cdev 指向的就是 cdev。那么如何通过 cdev 得到 scull_dev 呢？内核提供了 container_of()宏。

```
struct scullc_dev *dev;
dev = container_of(inode->i_cdev, struct scullc_dev, cdev);
```

这可比 down_cast 之类复杂多了。里面使用了黑客手段，有兴趣可以看看 container_of 的实现。

好像还忘记了什么。。。对了，模板呢？

说内核不需要模板是不可能的。光是链表一项，就有很多地方要用到，进程之间，dentry 之间。。。如果为每种情况写一套链表操作，那是很可怕的事。理论上，我们有两种选择，以循环链表，task_struct 为例：1.把指针 pprev,next 放到 task_struct 中，然后写一套宏，把类型作为参数传进去，实现对循环链表的操作。这个是最自然的思路，最接近 C++的模板。但是，问题来了，如果 task_struct 同时属于好几个链表怎么办（虽然听起来这个想法很怪，但 task_struct 的确有这样的需求）？

2.对第一种方法的改进：实现一个对最简单的结构体 list_head 的链表操作。然后把 list_head 等包含到 task_struct 结构体里。如果要对 task_struct 所在链表进行操作，只要操作对应的 list_head 就可以了。所以解决了 1 的问题。至于怎么通过 list_head 获得 task_struct，可以参考 container_of()宏的做法。

问题是解决了，但是与前面几种模拟办法相比，这种是最不直观的。因为当我在一个结构体里发现了 list_head 后，根本不知道它所在的链表究竟放的是什么结构体。事实上只要某个结构体有 list_head 这个成员，就可以放到链表里。有些类似于 MFC 的 CObjectList。这就有些恐怖了。

不过只要编程者清楚里面放了些什么，就没有问题。

“相信内核不会伤害自己”

不过我很好奇，如果换成 C++，如何实现 list_head 的效果呢？能否实现一种新的 multi_list 模板，它与 list 的区别在于节点可以属于多个链表。

读核感悟-设计模式-文件系统和设备的继承和接口

接着，我们可以看一下文件系统几个关键的结构体之间的关系。显然，一个 inode 对应多个 dentry，一个 dentry 对应多个 file。任何一本介绍文件系统的书对此都有介绍。那

么 inode 和块设备 block_device，字符设备 cdev 的关系如何呢。我们知道，inode 是对很多事物的抽象。在 2.4 内核中。inode 结构体中有一个 union，记录了几十种类型，包括各种文件系统的 inode，各种设备(块设备，字符设备，socket 设备等等)的 inode，我们可以把 block_device 和 cdev 理解为特殊的节点。他们除了普通节点的一些特性外，还有自己的接口。

如 block_device 有个成员 gendisk，有自己的接口 block_device_operations。而对于 ide 硬盘来说，它是一种特殊的 gendisk，它有自己的结构体 ide_driver_t。 它还实现了 block_device_operations 接口，即 idedisk_ops。

整体的继承关系：

block_device,gendisk（类）|block_device_operations（接口）

| is_a（继承）

ide_drive_t（类）|idedisk_ops(实现)

这种分析方法有助于理清内核中众多结构体之间的关系。

与此类似，我们也可以分析一下两种重要的设备 PCI 和 USB 设备：

内核中用来表示 pci 设备的结构体是 pci_dev，此外，还有一个接口 pci_driver，定义了一组 PCI 设备的操作。

内核中用来表示 usb 设备的结构体是 usb_device，此外，还有一个接口 usb_driver，定义了一组 USB 设备的操作。

有趣的是，pci_dev 和 usb_device 有一个共同的成员变量类型为 device，pci_driver 和 usb_driver 有一个共同的成员变量类型为 device_driver。

由此，我们可以猜出他们的关系。

pci_dev 和 usb_device 可以看成对 device 的继承，pci_driver 和 usb_driver 可以看作是从 device_driver 接口的继承。

在设备驱动中，各种层次显得非常分明。内核通过对 usb 和 pci 驱动通用框架的设计，减轻了驱动的开发人员的负担。实际上，内核为驱动开发人员提供的是一个框架(framework)。有些类似于 MFC。开发人员只要实现一些接口就可以了。

此外，我们还可以总结出一些有趣的现象：

比如，接口该如何定义呢?

Linux 把目录和设备看成文件，这使文件操作接口的定义有两种选择：

1.取普通文件，目录文件，设备文件接口的交集，为，把各种“文件”特有的接口放到各种“文件”自定义自己的接口。这样的好处是继承关系比较清楚。不过继承层次比较深。

2．取普通文件，目录文件，设备文件接口的并集，压缩继承层次。各种“文件”实现自己能实现的接口，把不能实现的函数指针置为 NULL

事实上，file_operations 就是这么做的。它里面有通用的操作(read write)。有针对目录文件的操作(getdents)。有针对设备的操作(ioctl)。与之类似的还有 inode_operations，包括了普通文件节点（create），设备文件节点(mknod)，目录文件节点(mkdir)等等。

通过这样的设计，大大简化了整个层次结构。

我们还可以归纳出更多东西：

1．为了保证扩展性，很多结构体提供一个 private 指针，如 file::private_data

2．如果只是为了代码重用，就提供普通库函数和普通结构体。如果为了提供接口以便继承，就提供接口。在 C 中，接口和普通成员函数很容易区分，前者一般定义成函数指

针。

读核感悟-设计模式-文件系统与抽象工厂

抽象工厂的典型应用是在 gui 设计中。为了在运行时方便地替换不同风格的 gui 设计，我们需要在上层把每种不同风格的 gui 的实现细节隐藏起来。我们不再显式地调用 gui 中的某个组件的构造函数，因为这样暴露了组件的实现。相反，我们为每一个风格提供一个“工厂”来生产该风格的“产品”。

抽象工厂可以看作是面向对象设计中常用的多态思想的扩展。普通的多态只是针对一个“产品”，如 shape 抽象类。而抽象工厂是针对一组“产品”。所以显得整齐划一。

在设计文件系统时，我们希望抽象的文件系统 VFS 和具体的文件系统分离。即在 VFS 中不涉及任何具体的文件系统。如果要添加一个新的文件系统，则只要添加几个源文件，然后编译成模块，就万事大吉了。为了达到这一目标，VFS 的代码本身不应该涉及具体的文件系统 。VFS 里需要构造 dentry,inode,super block 对象时，也应该想方设法地避免显式地调用构造函数。但是由于文件系统可以抽象出一些共同的要素和接口，如 dentry,inode,super_block,这时我们想到了抽象工厂的设计模式。

每一类文件系统可以看作是“工厂”，而 dentry,inode，super block 可以看作是“产品”。那么，工厂在哪里呢？我们是在哪里创建”工厂”本身的呢？

我们知道文件访问的大概流程：

1.首先，我们得知道是哪个文件系统。然后，通过 mount 把文件系统安装到某个目录下。这个结构体就是：file_system_type。我们可以看一下 ext2 的 file_system_type

```
1167 static struct file_system_type ext2_fs_type = {

1168         .owner          = THIS_MODULE,

1169         .name           = "ext2",

1170         .get_sb         = ext2_get_sb,

1171         .kill_sb        = kill_block_super,

1172         .fs_flags       = FS_REQUIRES_DEV,

1173 };
```

我们可以在注册文件系统时把这个结构体放到内核的全局链表：file_systems 中。然后通过文件系统的名称找到对应的 file_system_type 结构体。

2.其次，我们得获得 super block 的信息，因为一个文件系统通常只有一个 super block，由于 super block 的重要性，他会有很多个备份，但是从信息量的角度来讲仍然只有一份。文件系统的根目录也在 super block 中。我们可以构造出根目录的 inode 对象和 dentry 对象

好在我们已经有了 file_system_type。不出所料，我们可以发现里面有获得 super block 的函数指针：get_sb。这里有一个有趣的现象：为什么 ext2_fs_type，包括 ext2_get_sb 要设置为 static 呢？因为这样做其它模块（包括 VFS）就没有可能直接引用

ext2_fs_type 了。“逼”着 VFS 引藏细节。这其实是 C 风格的一种封装。由此我们也可以总结出：设计为 static，并不意味着其它模块不会调用到该变量或者函数，只是意味着无法直接调用到而已。那么 VFS 如何调用 ext2_get_sb 呢？在挂载文件系统时，在 do_kern_mount()中有如下代码：

sb = type->get_sb(type, flags, name, data);

由于我们在 mount 文件系统时会指定文件系统的名称。内核就会找到对应的 file_system_type。然后调用 get_sb。

从面向对象的角度来说，super_block 对象不仅包括 super block 的数据成员，还包括对 super_block 的一组操作的实现。所以应该有个接口叫 super_operations。果然，ext2 文件系统有个 super_operations 变量叫 ext2_sops，在 ext2_get_sb 赋给对应的 super block.

那么根结点如何找？不要急，根据之前的分析，super_operations 应该提供读 inode 的操作，事实上，它是一个叫 read_inode 的函数指针。另一方面，super_block 中有个 dentry 成员变量叫 s_root，就是我们要找的根节点的 dentry!此外，系统还悄悄地把一些函数指针也赋了值。如 inode_operations 。inode ,dentry ,file 对象的构造通常伴随着 inode_operations,dentry_operatons,file_operations 的赋值。这在 C++中是由编译器来负责实现的。在 C 中只能人肉实现了。

通过这种方法，引藏了具体文件系统的细节。

3.接着，有了根目录的 inode，我们可以很方便地访问它的子节点。于是采用 lazy algorithm，只有当子节点被访问到，我们才去硬盘上访问它，并在内存中构造它。这一步是在 sys_open 中实现的。

sys_open()->filp_open()->open_namei()->path_lookup()->link_path_walk()->__link_path_walk()->do_lookup()->real_lookup()

在这里，当系统发现要访问的子节点不在内存中，，它不得不去硬盘上找。不过对应的 lookup 接口已经在父节点的“构造函数”中“人肉”赋值了。这里它只需要调用如下代码：

result = dir->i_op->lookup(dir, dentry, nd);

就实现了在磁盘上查找文件结点的操作。

由此我们可以总结出：

1.文件系统的“工厂”由很多组接口组成，包括 file_system_type, super_operatios,inode_operations,dentry_operations, file_operations.只要文件系统类型确定，这些接口即“工厂”也确定了。在 VFS 中就可以根据 ext2 工厂生产出对应的产品即 ext2 的 super block ,inode ,dentry ,file 对象。只要我们在 mount 时换一下文件系统如 reiserfs，ext2“工厂”就立即摇身一变，成了 reiserfs 的“工厂”。而 VFS 的代码不需要作任何改变。

2.为了在语法上保证松散耦合，除了需要通过 EXPORT_SYMBOL 导出的符号，我们可以把模块中的函数和变量尽量地声明为 static，以实现一定程度的封装。

3.对象的构造总是伴随着成员变量和方法的赋值。尤其是后者，使 VFS 能够不用调用下层文件系统的具体代码。

读核感悟-阅读源代码技巧-查找定义

阅读内核源代码的工具有很多，例如 lxr+swish-e，source insight，emacs+cscope，vim+ctags 等等。这里以 lxr+swish-e 为例。

lxr+swish-e 的安装是比较麻烦的，可以参考下面的教程：
http://forum.ubuntu.org.cn/viewtopic.php?t=80527&sid=66446a9ff4783d61b05cd33a8023fe30

然后用浏览器打开对应的网址，就可以访问了。变量，函数的声明，定义和使用，都能找出来。

不过也有一些特殊情况：

1.只有声明，找不到定义。比如 asmlinkage int system_call(void);

lxr 只会从 c 的源代码中找定义。但是定义不一定在.h 或者.c 中，还可能在.S 汇编代码或者.lds.S 链接脚本中。

1）链接脚本：

比如，在时钟中断中经常出现的 jiffies 就定义在 arch/i386/kernel/vmlinux.lds.S。

```
jiffies = jiffies_64;
```

它的声明为：

```
extern unsigned long volatile __jiffy_data jiffies;
```

而 jiffies_64 则是在 arch/i386/kernel/time.c 里正常定义的：

```
u64 jiffies_64 = INITIAL_JIFFIES;
```

这意味 jiffies 和 jiffies_64 的变量地址是一样的，或者说，他们根本就是同一个变量。为什么要这么写呢？我们注意到，jiffies 被声明为一个 32 位无符号整数，jiffies_64 被定义为一个 64 位无符号整数，也许是为了保证向下兼容吧。因为在 2.4.33 内核里 jiffies 还是 32 位的。但是在 2.6.13 里，内核已经不再使用 jiffies 而使用 64 位的 jiffies_64 了。但是，为了与 2.4 版本的兼容，使那些使用 jiffies 的驱动和其它模块仍然能正常使用 jiffies。内核使用了这样一个小伎俩。

其它如__initcall_start，__initcall_end。也都在（事实上只能在）vmlinux.lds.S 中定义。

2）汇编文件：

还有定义在汇编文件中的变量。很多入口函数都定义在汇编文件中。其中 arch/i386/kernel/entry.S 里面定义了大量的入口函数，包括系统调用入口 system_call，页面异常 page_fault，以及其它在 trap_init()里设置的中断向量表中的入口。

这里不能不提一下普通中断 interrupt 数组。它的声明如下：

```
extern void (*interrupt[NR_IRQS])(void);
```

是一个函数指针

在 2.4.33 的内核中，interrupt 数组是由一组宏来定义的：

```
101 #define IRQ(x,y) \

102         IRQ##x##y##_interrupt

103

104 #define IRQLIST_16(x) \
```

```
105         IRQ(x,0), IRQ(x,1), IRQ(x,2), IRQ(x,3), \
106         IRQ(x,4), IRQ(x,5), IRQ(x,6), IRQ(x,7), \
107         IRQ(x,8), IRQ(x,9), IRQ(x,a), IRQ(x,b), \
108         IRQ(x,c), IRQ(x,d), IRQ(x,e), IRQ(x,f)
110 void (*interrupt[NR_IRQS])(void) = {
111         IRQLIST_16(0x0),
112
113 #ifdef CONFIG_X86_IO_APIC
114                         IRQLIST_16(0x1), IRQLIST_16(0x2), IRQLIST_16(0x3),
115         IRQLIST_16(0x4), IRQLIST_16(0x5), IRQLIST_16(0x6), IRQLIST_16(0x7),
116         IRQLIST_16(0x8), IRQLIST_16(0x9), IRQLIST_16(0xa), IRQLIST_16(0xb),
117         IRQLIST_16(0xc), IRQLIST_16(0xd)
118 #endif
119 };
```

而 IRQ##x##y##_interrupt
又是由 BUILD_IRQ 来定义。

```
175 #define BUILD_IRQ(nr) \
176 asmlinkage void IRQ_NAME(nr); \
177 __asm__( \
178 "\n"__ALIGN_STR"\n" \
179 SYMBOL_NAME_STR(IRQ) #nr "_interrupt:\n\t" \
180         "push1 $"#nr"-256\n\t" \
181         "jmp common_interrupt");
```

这种方式很清晰明了，不过有些罗嗦。

2.6.13 中对 interrupt 的定义方式进行了改进，更简洁，但是也更不容易理解：

事实上它的定义也是放在 entry.S。只是很不容易被发现。

```
396 .data
397 ENTRY(interrupt)

398 .text

399

400 vector=0

401 ENTRY(irq_entries_start)

402 .rept NR_IRQS

403         ALIGN

404 1:      pushl $vector-256

405         jmp common_interrupt

406 .data

407         .long 1b

408 .text

409 vector=vector+1

410 .endr

411

412         ALIGN
```

NR_IRQS

一般被定义为 16。

这段代码利用了汇编的.rept 功能，把代码展开为 16 段，每一段代码只是把 vector 加 1，也就是 irq 号。每一段代码的功能就是把$vector-256

的值压入堆栈（因为 irq 号在 0 到 255 之间，所以要与系统调用号区别开来。），然后跳转到 common_interrupt。

那么 interrupt 数组里的函数指针在哪里定义呢?

我们发现标签 1 处是每段小函数的入口。而

```
406 .data

407         .long 1b
```

则把标签 1 的地址保存到数据段。是哪个数据段呢?

```
396 .data
397 ENTRY(interrupt)
```

我们可以看到。其实就相当于把这 16 个函数指针放在 interrupt 后面。相当于定义了一个 interrupt 数组。

3.函数以宏的方式批量定义。

例如对 IO 端口的操作: outb_p 等等。

定义在 include/asm-i386/io.h 中

```
377 BUILDIO(b,b,char)

378 BUILDIO(w,w,short)

379 BUILDIO(1,,int)
```

每个 BUILDIO 宏将生产很多个函数:

以 BUILDIO(b,b,char)

为例:

能生成以下函数:

```
outb_local
inb_local
outb_local_p
inb_local_p
outb
inb
outb_p
inb_p
outsb
insb
```

如果把 b,w,1 三种情况都按常规方法定义一遍, 无疑将产生大量雷同代码。遇到这种情况, 我的第一反应就是模板。可惜 C 里没有模板。所以只能用宏来代替了。

那么, 遇到这类情况, 通用的解决办法是什么呢?

1.我们可以求助于 lxr 的 free text search。全文搜索包括了.S 汇编文件。不过搜索时量会很多。

2.可以去查查相关资料, 如 Understanding Linux Kernel, Linux 内核源代码情景分析 (虽然只是 2.4 内核的, 但也可以以此为基础, 猜测出 2.6 的定义方法, 例如, 可能在同一个头文件。) 。

读核感悟-阅读源代码技巧-变量命名规则

在阅读源代码的时候，经常会发现在跟踪函数调用时跟踪不下去了，如

result = dir->i_op->lookup(parent_inode, child_dentry, nd);

这类似于C++中的多态。inode_operations中的lookup函数指针具体指向哪个函数，不是在编译时确定的，而是在运行期确定的。具体地说，就是在从磁盘上读入inode时才对对应inode的inode_operations结构体赋值。在ext2_read_inode()中

inode->i_fop = &ext2_file_operations;

也就是在inode对象创建的同时也对inode对应的操作进行了绑定。

但是为了确定函数指针具体指向的是哪一个函数，而去找函数指针的赋值的地方，未免有些麻烦。而通过内核变量的命名规则猜测对应的函数倒不失为一个好办法。在这个例子中，我们使用的文件系统是ext2。于是猜测lookup对应的函数是ext2_lookup。很幸运，我们猜对了。为了保证扩展性，内核中使用了大量的函数指针，采用了大量的“动态绑定”。在这种情况下，总结一下内核变量的命名规则是很有必要的。

我个人认为变量的命名是很重要的，因为这关系到代码的可读性。通常每个公司都有自己的一套变量命名规范。比较有名的有微软的匈牙利命名法，以及Unix应用程序的命名方式。Linux内核采用的是Unix应用程序的命名方式，标志符通常采用“小写加下划线”的方式。

sys_+系统调用名:是系统调用在内核中的入口处理函数如sys_fork(),因为所有的socket通信相关的系统调都通过sys_socketcall，所以他们的命名形式也像普通系统调用一样。如sys_socket,sys_bind等等，

do_+系统调用/中断操作名等等:具体的处理函数，如do_fork(被sys_fork,sys_vfork,sys_clone调用),do_page_fault,do_mmap.

vfs_+VFS相关的系统调用名:第二层的处理函数,如vfs_read,vfs_write,vfs_ioctrl。

文件系统函数方面:

文件系统名称_+操作名:该文件系统对应的操作。如ext2_lookup对应inode_operations中的lookup操作。

文件系统名称_+接口名:该文件系统针对该接口实现的操作：如ext2_file_operations。当然，实际情况比较复杂，如ext2中对inode的操作又分为普通文件的inode(ext2_file_inode_operations)，目录文件的inode(ext2_dir_inode_operations)，特殊文件如设备文件的inode(ext2_special_inode_operations)等等。

结构体名称缩写_+成员变量名:结构体成员变量的命名，如inode的成员变量前面都有i_，dentry的成员变量前面都有d_，super_block的为s，file的为f等等。

一些缩写，有助于理解变量的含义。

nr:number

bh:bottom half

ops:operations如f_ops，i_ops

在新版本的内核代码里，一些缩写开始用完整的单词来代替。这使代码的可读性进一步提高。

读核感悟-内存管理-内核中的页表映射总结

从线性地址到物理地址的转换，实际上是一种映射。所有进程的 3~4G 的线性地址实际上是映射到相同的物理地址的。这一点不多说了。为了方便起见，3~4G 的线性地址与对应的物理地址基本上是呈线性关系的。即线性地址=物理地址+3G。但是如果把这 1G 的线性地址都简单地处理为对应物理地址+3G，就会有新的问题产生。例如，如果物理地址大于 4G，那么内核就没法访问这些地址了。所以，内核必须要从这 1G 的线

性空间中预留一部分，作其它用途，（例如，映射高端物理地址）。经过实践，发现 128M 线性地址够用了。所以，3~4G 的线性地址中 896M 映射到物理地址中的 0~896M，剩下的 128M 挪作它用，例如，内核空间的非线性映射（在 vmalloc 中使用）以及高于 4G 的物理地址的映射。

我曾经产生过的疑问：

1.如果物理内存小于 896M，那不是所有的物理地址都被内核用完，没有内存留给用户空间了么？

事实上，小于 896M 时，同一块物理内存可能同时被映射到用户空间和内核空间，从数学的角度来讲，也就是所谓的“双箭一雕”。我们相信内核是没有 bug 的，所以不用担心因此产生的内核对用户空间数据破坏的可能性。

2.如果物理内存小于 896M，所有物理内存地址都与内核线性地址相差 3G，那 vmalloc 不是没有内存使用了么？

不使用 vmalloc 当然是不可能的，因为模块装载时就会用到 vmalloc。原理与问题 1 一样，即使在内核空间，也可能多个线性地址映射到同一个物理地址，也就是“双箭一雕”。所以 vmalloc 仍然能获得相应的物理内存。

对以上两个问题的小结。系统在启动时会把前 896M 物理地址映射到 3G 以上的线性空间。但是映射不等于使用。事实上内核只使用了很小一部分（一般才几 M）。只要有足够的未使用的物理内存，用户进程和 vmalloc 都不会有问题。

3.如果物理内存大于 896M。那对内核而言不是无法在同一时刻访问所有的物理内存了么？

是的，虽然内核可以通过临时映射和永久映射来访问所有的物理内存，但是用到的线性地址很有限，即一次只能访问一小部分物理内存。当无法访问到更高地址物理内存时，只能通过修改内核页表来达到目的。但是，这对内核就足够了。因为上面说过，内核所需要的内存很少。内核只是偶尔会访问一下高端内内存。而对于用户进程而言，只要页表开启了 PAE，就可以访问 64G 以内的物理内存（虽然只有 4G 的线性空间

，但是也足够了。）

4.内核页表是如何共享的呢?

我们知道所有的进程共享内核空间，所以共享内核页表是很自然的事。理论上内核只有一个页表，对应的内核全局页目录 swapper_pg_dir，所有进程的页目录的最高 256 项与 swapper_pg_dir 相同。可惜的是，每个进程有自己的页目录，共 1024 项。其中最高的 256 项指向的是内核空间。尽管这些页目录项可以指向同样的内核页表，但是一旦内核页目录改变了，所有进程的页目录都需要同步。这种情况是存在的。比如

内核调用 vmalloc 时。

幸亏有了页错误，我们可以从容地处理页目录的同步。如果内核调用了 vmalloc，内核只修改内核全局页目录。当其它进程访问 vmalloc 产生的线性空间时，会产生页错误。页错误处理程序可以判断当前的页错误是由于 vmalloc 产生的，于是修改对应的页目录，使它们与内核全局页目录保持一致。

读核感悟-健壮的代码-exception table-内核中的刑事档案

如果系统调用的参数是错误的怎么办? 如果参数仅仅是简单的整型，只要直接判断参数范围就可以了，但是如果是指针呢? 情况就复杂了。

如果指针为 null，是很容易判断的。如果指针不为 null，但是却指向非法的地址，该怎么办呢? 如果在内核里段错误，内核就太冤了：明明是用户态程序的问题，却让我来背黑锅。。。

一种办法是在系统调用开始时就判断指针在不在用户空间里。进一步说，对指针指向的 buffer 的每一个字节都要作判断。这样可以提前检查出错误的参数，返回一个错误码。

这样的缺点是太复杂，效率太低。因为出错的情况是少数。

针对这种情况，内核采用了 exception table。把所有可能出错的指令地址都放到 exception table 中，与之对应的是一个错误处理函数。

例如，系统调用从用户态拷贝参数到内核态，要通过 copy_from_user()。这个函数是用汇编写的。除了效率因素，还有一个原因是可以精确地判定出错指令的地址（即其中的一个 mov 指令）。fix code 则是一小段代码，它返回 0。于是，当 copy_from_user()出错时，在 page fault 处理函数里检查当前的 eip 是不是在 exception table 中（即那条 mov 指令），如果是的话，就跳到对应的 fix code（即返回为 0）。

这个方法兼顾了效率（不需要做额外的检查）和可靠性（保证用户程序的错误不会导致内核崩溃）。我们可以把 exception table 看作刑事档案，它告诉我们“某某某代码有前科，可能会有麻烦，请特别关照”。

这从另一个侧面告诉我们，内核中的代码对健壮性要求特别高。

读核感悟-定时器-巧妙的定时器算法

内核中经常要用到各种定时器。比如 nanosleep()系统调用，让当前进程睡眠一段时间，再把它唤醒。即在 expires 时刻（以时钟滴答为单位），自动调用 wake_up_process。

最直接的思路是定义一个定时器，里面有 function（函数指针），data（函数参数），expires（调用时刻）。然后排成一个链表。每次时钟中断发生时，扫描整个链表，发现有触发的定时器，就调用 function(data)。寻找触发的定时器的时间复杂度 O(N)。N 是定时器数量。明显，这个算法效率太低。

改进：事实上我们只对最快触发的定时器感兴趣，所以在插入元素时就对元素进行排序。这样，每次时钟中断，只要检查链表中第一个定时器就行。寻找触发的定时器的时间复杂度 O(1)。但是每次插入和删除操作的时间复杂度又变成 O(N)了。这个算法还是不好。

继续改进：把数据结构改成优先队列（最小堆），这样插入删除操作的时间复杂度为 O(logN)。寻找触发的定时器的时间复杂度 O(1)。

有没有可能继续改进呢？墙上的挂钟给人以灵感。当秒针飞快走动时，分针走的很慢，而时针几乎不动。这提醒我们要区别对待定时器。要把定时器划分成不同区间。对最早可能触发的定时器频繁扫描（就像秒针），而对很晚才触发的定时器隔一段时间再扫描（就像分针）。

基本思路：2^8-1 个时钟滴答内触发的定时器放到第一组，2^8~2^14-1 个时钟滴答内触发的定时器放到第二组，依次类推，最后分成五组。第一组每个时钟中断都要检查，第二组则每 2^8=256 个时钟中断检查一次，依次类推。每组中有 256 个队列。那么检查每一组是否意味着检查这 256 个队列？不是的。事实上，根据触发时间，我们可以把定时器放到对应的队列中（如在第一组中，把触发时间为 expires 的定时器放到第 expires%256 个队列中）。那么我们在检查每一组时，只要扫描其中一个队列就可以了。对于第一组，被检查到的定时器需要执行定时器函数。对于其它组，则意味着“升级”（跳到前一组，由于临近触发而受到更频繁的检查）。

可以看出，这个算法中，寻找触发的定时器的时间复杂度接近 O(m)(m 是第一组每个队列中定时器的个数，其它组的操作几乎可以忽略)。而这个 m 在绝大部分情况下接近 1。插入和删除操作的时间复杂度也为 O(1)。这是迄今为止最好的算法。

更详细的算法解释见 ULK 第六章 timing mesuring。

从中也可以看出，把一个大队列分割成几个小队列，可以极大的降低算法的时间复杂度，提高效率。2.6 内核中，调度算法也采用了类似的思想。

读核感悟-内存管理-page fault 处理流程

page fault 是 Linux 内存管理中比较关键的部分。理解了 page fault 的处理流程，

有助于对 Linux 内核的内存管理机制的全面理解。因为要考虑到各种异常情况，并且为了使内核健壮高效，所以 page fault 的处理流程是比较复杂的。我把这个繁琐的处理流程放在最后。在 page fault 处理函数中使用了很多 lazy algorithm。它的核心思想是，由于磁盘 IO 非常耗时，所以把这些操作尽可能的延迟，从而省略不必要的操作。

以下是几种会导致 page fault 的情景：

1.用户态按需调页：

为了提高效率，Linux 实现了按需调页。应用程序在装载时，并不立即把所有内容读到内存里，而仅仅是设置一下 mm_struct，直到产生 page fault 时，才真正地分配物理内存。如果没有分配对应的页表，首先分配页表。这种情况下的缺页可能是匿名页(调用 do_no_page)，可能是映射到文件中的页(调用 do_file_page)，也可能是交换分区的页(调用 do_swap_page)。此外，还可以判断是不是 COW（写时复制）。

2.主内核页目录的同步：

内核页表信息保存在主内核页全局目录中，虚存段信息放在 vm_struct 中。进程页表的内核部分要保持与主内核页全局目录的同步。当内核调用 vmalloc 等函数，对内核态虚拟地址进行非线性映射时，修改主内核页全局目录，但是不修改进程页表的内核部分。这会引起 page fault。page fault 处理函数会执行 vmalloc_fault 里的代码，对进程的页表进行同步。

3.对 exception table 中的异常操作的处理

内核函数通过系统调用等方式访问用户态的 buffer，可能会在内核态导致 page fault。这一类 page fault 是可以被 fixup 的，所有这些代码的地址都放在 exception table 中。并且这些代码有异常处理函数，被称为 fixup code。page fault 处理函数查找对应的 fixup code，并且把返回时的 rip 设置为 fixup code。当 page fault 处理完毕，内核会调用 fixup code，对异常进行处理。典型的例子是 copy_from_user。

4.堆栈自动扩展

并不是所有的指针越界都会导致 SEGV 段错误。当指针越界的量很小，并且正好在当前堆栈的下方时，内核会认为这是正常的堆栈扩展，为堆栈分配更多物理内存。

5.对用户态指针越界的检查

如果指针越界，并且不是堆栈扩展，那么内核认为是应用程序的段错误，向应用程序强制发送 SEGV 信号。

6.oops

如果 page fault 不是应用程序引起，并且不是内核中正常的缺页，那么内核认为是内核自己的错误。page fault 会调用__die()打印这时的内核状态，包括寄存器，堆栈等等。

Page fault 的处理流程如下：

1.对参数有效性的检查：

a)如果出错地址在内核态，并且不是 vmalloc 引起的，那么 oops，内核 bug

b)如果内核在执行内核线程或者进行不容打断的操作（中断处理程序，延迟函数，禁止抢占的代码），oops

c)如果出错地址在用户态，并且可以在 exception table 中找到，那么执行 exception 的处理函数，正常返回，否则，oops

2.如果在进程的地址空间 vma 找不到对应的 vma，

a)判断是不是堆栈扩展，如果是，扩展堆栈。

b)如果错误在内核态发生，在 exception table 寻找异常处理函数：fixup

c)如果在用户态，向当前进程发送一个 SIGV 的信号。

3.如果在进程的地址空间内

a)如果是写访问

i.如果没有写权限，非法访问

ii.如果 vma 有写权限，pte 没有写权限，判断是不是 COW，是的话调用 do_wp_page。

b)如果是读访问，没有读权限，非法访问

c)如果不是权限问题，是普通的缺页，调用 handle_mm_fault 来解决

i.handle_mm_fault，如有需要，分配 pud,pmd,pte

ii.如果页不在内存中

1.如果还没有分配物理内存，调用 do_no_page

2.如果映射到文件中，还没有读入内存，调用 do_file_page

3.如果该页的内容在交换分区上，调用 do_swap_page

4.如果在内核态缺页，并且是由于 vmalloc 引起

a)根据 master kernel page table 同步

读核感悟-文件读写-select 实现原理

1 功能介绍

select 系统调用的功能是对多个文件描述符进行监视，当有文件描述符的文件读写操作完成，发生异常或者超时，该调用会返回这些文件描述符。

```
int select(int nfds, fd_set *readfds, fd_set *writefds,
           fd_set *exceptfds, struct timeval *timeout);
```

2 关键的结构体(内核版本 2.6.9)：

这里先介绍一些关键的结构体：

```
typedef struct {
        unsigned long *in, *out, *ex;
        unsigned long *res_in, *res_out, *res_ex;
} fd_set_bits;
```

这个结构体负责保存 select 在用户态的参数。在 select()中，每一个文件描述符用一个位表示，其中 1 表示这个文件是被监视的。in，out，ex 指向的 bit 数组表示对应的读，写，异常文件的描述符。res_in，res_out，res_ex 指向的 bit 数组表示对应的读，写，异常文件的描述符的检测结果。

```
struct poll_wqueues {
        poll_table pt;
        struct poll_table_page * table;
        int error;
};
```

这是最主要的结构体，它保存了 select 过程中的重要信息。它包括了两个最重要的结构体 poll_table 和 struct poll_table_page。接下去看看这两个结构体。

```
typedef void (*poll_queue_proc)(struct file *, wait_queue_head_t *, struct
poll_table_struct *);
struct poll_table_struct {
        poll_queue_proc qproc;
} poll_table;
```

在执行 select 操作时，会用到回调函数，poll_table 就是用来保存回调函数的。这个回调函数非常重要，因为当文件执行 poll 操作时，一般都会调用这个回调函数。所以，这个回调函数非常重要，通常负责把进程放入等待队列等关键操作。下面可以看到，在 select 中，这个回调函数是__pollwait()，在 epoll 中，这个回调函数是 ep_ptable_queue_proc。

```
struct poll_table_page {
        struct poll_table_page * next;
        struct poll_table_entry * entry;
        struct poll_table_entry entries[0];
};
```

这个表记录了在 select 过程中生成的所有等待队列的结点。由于 select 要监视多个文件描述符，并且要把当前进程放入这些描述符的等待队列中，因此要分配等待队列的节点。这些节点可能如此之多，以至于不可能像通常做的那样，在堆栈中分配它们。所以，select 以动态分配的方式把它保存在 poll_table_page 中。保存的方式是单向链表，每个节点以页为单位，分配多个 poll_table_entry 项。

现在看一下 poll_table_entry:

poll table 的表项

```
struct poll_table_entry {
        struct file * filp;
        wait_queue_t wait;
        wait_queue_head_t * wait_address;
};
```

其中 filp 是 select 要监视的 struct file 结构体，wait_address 是文件操作的等待队列的队首，wait 是等待队列的节点。

3 select 的实现

接下去介绍 select 的实现

在进行一系列参数检查后，sys_select 调用 do_select()。该函数会遍历所有需要监视的文件描述符，然后调用 f_op->poll()，这个操作会做两件事:

1.查看文件操作的状态，如果这些文件操作完成或者有异常发生（下面统称为“用户感兴趣的事件”），在对应的 fdset 中标记这些文件描述符。

2.如果 retval 为 0（在这一轮遍历中，迄今为止，文件没有发生感兴趣的事，这一点有些不明白，为什么不是通知所有监视的并且没有发生感兴趣事件的文件描述符，这样返回得更快）并且没有超时，那么，通知这些文件，让他们在文件操作完成时唤醒本进程。

如果发现文件具体通知的方式是：通过__pollwait()把自己挂到各个等待队列中。这样，当有文件操作完成时，select 所在进程会被唤醒。这里涉及到一个回调函数 __pollwait()。它是在什么时候被注册的呢？在进入 for 循环之前，有这样一行代码:

```
poll_initwait(&table);
```

它的作用就是把 poll_table 中的回调函数设置为__pollwait。

对文件描述符的遍历的循环会继续，直到发生以下事件:

如果有文件操作完成或者发生异常，或者超时，或者收到信号，select 会返回相应的值，否则，do_select 会调用 schedule_timeout()进入休眠，直到超时或者被再次唤醒（这表明有用户感兴趣的事件产生），然后重新执行 for 循环，但是这一次一定能跳出循环体。

通知的过程如下（以管道的 poll 函数为例，在 pipe 中，f_op->poll 对应的函数是 pipe_poll: ）:

当 do_select 遍历所有需要监视的文件描述符时，假设有一个文件描述符对应的是一个管道，那么，它执行的 f_op->poll 实际上是 pipe_poll。pipe_poll->poll_wait->__pollwait。最终__pollwait 会把当前进程挂到对应文件的 inode 中的文件描述符中。当执行 pipe_write 对管道进行写操作时，操作完成后会唤醒等待队列中所有的进程。

4 性能分析

从中可以看出，select 需要遍历所有的文件描述符，就遍历操作而言，复杂度是 O(N)，N 是最大文件描述符加 1。此外，select 参数包括了所有的文件描述符的信息，所以 select 在遍历文件描述符时，需要检查文件描述符是不是自己感兴趣的。

读核感悟-文件读写-poll 的实现原理

1 功能介绍：

poll 与 select 实现了相同的功能，只是参数类型不同。它的原型是：

```
int poll(struct pollfd *fds, nfds_t nfds, int timeout);
```

可以看到，poll 的参数中，直接列出了要监视的文件描述符的信息，而不像 select 一样要列出从 0 开始到 nfds-1 的所有文件描述符。这样的好处是，poll 不需要查询很多无关的文件描述符的信息，在一定场合下效率会有所提高。

2 关键的结构体：

poll 用到的很多结构体与 select 是一样的，如 struct poll_wqueues。这是因为 poll 的实现机制与 select 没有本质区别。

poll 也用到了一些不同的结构体，这是因为 poll 的参数类型与 select 不同，用来保存参数的结构体也不同：

相关的数据结构：

```
struct poll_list {
        struct poll_list *next;
        int len;
        struct pollfd entries[0];
};
```

这个结构体用来保存被监视的文件描述符的参数。其中 struct pollfd entries[0] 表示这是一个不定长数组。

```
struct pollfd {
        int fd;
        short events;
        short revents;
};
```

这个结构体记录被监视的文件描述符和它的状态。

3 poll 的实现

poll 的实现与 select 也十分类似。一个区别是使用的数据结构。poll 中采用了 poll_list 结构体来记录要监视的文件描述符信息。poll_list 中，每个 pollfd 代表一个要监视的文件描述符的信息。这些 pollfd 以数组的形式链接到 poll_list 链表中，每个数组

的元素个数不能超过 POLLFD_PER_PAGE。在把参数拷贝到内核态之后，sys_poll 会调用 do_poll()。

在 do_poll()中，函数遍历 poll_list 链表，然后调用 do_pollfd()对每个 poll_list 节点中的 pollfd 数组进行遍历。在 do_pollfd()中，检查数组中的每个 fd，检查的过程与 select 类似，调用 fd 对应的 poll 函数指针：

```
mask = file->f_op->poll(file, *pwait);
```

1.如果有必要，把当前进程挂到文件操作对应的等待列队中，同时也放到 poll table 中。

2.检查文件操作状态，保存到 mask 变量中。

在遍历了所有文件描述符后，调用

```
timeout = schedule_timeout(timeout);
```

让当前进程进入休眠状态，直到超时或者有文件操作完成，唤醒当前进程才返回。

那么，在 f_op->poll 中做了些什么呢？在 sys_poll 中有这样一行代码：

```
poll_initwait(&table);
```

可以发现，在这里，它注册了和 select()相同的回调函数__pollwait(),内部的实现机制也是一样的。在这里就不重复说了。

4 性能分析：

poll 的参数只包括了用户感兴趣的文件信息，所以 poll 在遍历文件描述符时不用像 select 一样检查文件描述符是否是自己感兴趣的。从这个意义上说，poll 比 select 稍微要高效一些。前提是：要监视的文件描述符不连续，非常离散。

poll 与 select 共同的问题是，他们都是遍历所有的文件描述符。当要监视的文件描述符很多，并且每次只返回很少的文件描述符时，select/poll 每次都要反复地从用户态拷贝文件信息，每次都要重新遍历文件描述符，而且每次都要把当前进程挂到对应事件的等待队列和 poll_table 的等待队列中。这里事实上做了很多重复劳动。

读核感悟-文件读写-epoll 的实现原理

1 功能介绍

epoll 与 select/poll 不同的一点是，它是由一组系统调用组成。

```
int epoll_create(int size);
int epoll_ctl(int epfd, int op, int fd, struct epoll_event *event);
int epoll_wait(int epfd, struct epoll_event *events,
                int maxevents, int timeout);
```

epoll 相关系统调用是在 Linux 2.5.44 开始引入的。该系统调用针对传统的 select/poll 系统调用的不足，设计上作了很大的改动。select/poll 的缺点在于：

1.每次调用时要重复地从用户态读入参数。

2.每次调用时要重复地扫描文件描述符。

3.每次在调用开始时，要把当前进程放入各个文件描述符的等待队列。在调用结束后，又把进程从各个等待队列中删除。

在实际应用中，select/poll 监视的文件描述符可能会非常多，如果每次只是返回一

小部分，那么，这种情况下 select/poll 显得不够高效。epoll 的设计思路，是把 select/poll 单个的操作拆分为 1 个 epoll_create+多个 epoll_ctrl+一个 wait。此外，内核针对 epoll 操作添加了一个文件系统”eventpollfs”，每一个或者多个要监视的文件描述符都有一个对应的 eventpollfs 文件系统的 inode 节点，主要信息保存在 eventpoll 结构体中。而被监视的文件的重要信息则保存在 epitem 结构体中。所以他们是一对多的关系。

由于在执行 epoll_create 和 epoll_ctrl 时，已经把用户态的信息保存到内核态了，所以之后即使反复地调用 epoll_wait，也不会重复地拷贝参数，扫描文件描述符，反复地把当前进程放入/放出等待队列。这样就避免了以上的三个缺点。

接下去看看它们的实现:

2 关键结构体：

```
/* Wrapper struct used by poll queueing */
struct ep_pqueue {
        poll_table pt;
        struct epitem *epi;
};
```

这个结构体类似于 select/poll 中的 struct poll_wqueues。由于 epoll 需要在内核态保存大量信息，所以光光一个回调函数指针已经不能满足要求，所以在这里引入了一个新的结构体 struct epitem。

```
/*
 * Each file descriptor added to the eventpoll interface will
 * have an entry of this type linked to the hash.
 */
struct epitem {
        /* RB-Tree node used to link this structure to the eventpoll rb-tree */
        struct rb_node rbn;
```

红黑树，用来保存 eventpoll

```
        /* List header used to link this structure to the eventpoll ready list
*/
        struct list_head rdllink;
```

双向链表，用来保存已经完成的 eventpoll

```
        /* The file descriptor information this item refers to */
        struct epoll_filefd ffd;
```

这个结构体对应的被监听的文件描述符信息

```
        /* Number of active wait queue attached to poll operations */
        int nwait;
```

poll 操作中事件的个数

```
        /* List containing poll wait queues */
        struct list_head pwqlist;
```

双向链表，保存着被监视文件的等待队列，功能类似于 select/poll 中的 poll_table

```
        /* The "container" of this item */
        struct eventpoll *ep;
```

指向 eventpoll，多个 epitem 对应一个 eventpoll

```
        /* The structure that describe the interested events and the source fd
```

```
*/
        struct epoll_event event;
```
记录发生的事件和对应的 fd
```
        /*
         * Used to keep track of the usage count of the structure. This avoids

         * that the structure will desappear from underneath our processing.
         */
        atomic_t usecnt;
```
引用计数
```
        /* List header used to link this item to the "struct file" items list */
        struct list_head fllink;
```
双向链表，用来链接被监视的文件描述符对应的 struct file。因为 file 里有 f_ep_link，用来保存所有监视这个文件的 epoll 节点
```
        /* List header used to link the item to the transfer list */
        struct list_head txlink;
```
双向链表，用来保存传输队列
```
        /*
         * This is used during the collection/transfer of events to userspace
         * to pin items empty events set.
         */
        unsigned int revents;
```
文件描述符的状态，在收集和传输时用来锁住空的事件集合
```
};
```
该结构体用来保存与 epoll 节点关联的多个文件描述符，保存的方式是使用红黑树实现的 hash 表。至于为什么要保存，下文有详细解释。它与被监听的文件描述符一一对应。
```
struct eventpoll {
        /* Protect the this structure access */
        rwlock_t lock;
```
读写锁
```
        /*
         * This semaphore is used to ensure that files are not removed
         * while epoll is using them. This is read-held during the event
         * collection loop and it is write-held during the file cleanup
         * path, the epoll file exit code and the ctl operations.
         */
        struct rw_semaphore sem;
```
读写信号量
```
        /* Wait queue used by sys_epoll_wait() */
        wait_queue_head_t wq;
        /* Wait queue used by file->poll() */
        wait_queue_head_t poll_wait;
        /* List of ready file descriptors */
        struct list_head rdllist;
```

已经完成的操作事件的队列。

```
        /* RB-Tree root used to store monitored fd structs */
        struct rb_root rbr;
保存 epoll 监视的文件描述符
};
```

这个结构体保存了 epoll 文件描述符的扩展信息，它被保存在 file 结构体的 private_data 中。它与 epoll 文件节点一一对应。通常一个 epoll 文件节点对应多个被监视的文件描述符。所以一个 eventpoll 结构体会对应多个 epitem 结构体。

那么，epoll 中的等待事件放在哪里呢？见下面

```
/* Wait structure used by the poll hooks */
struct eppoll_entry {
        /* List header used to link this structure to the "struct epitem" */
        struct list_head llink;
        /* The "base" pointer is set to the container "struct epitem" */
        void *base;
        /*
         * Wait queue item that will be linked to the target file wait
         * queue head.
         */
        wait_queue_t wait;
        /* The wait queue head that linked the "wait" wait queue item */
        wait_queue_head_t *whead;
};
```

与 select/poll 的 struct poll_table_entry 相比，epoll 的表示等待队列节点的结构体只是稍有不同，与 struct poll_table_entry 比较一下。

```
struct poll_table_entry {
        struct file * filp;
        wait_queue_t wait;
        wait_queue_head_t * wait_address;
};
```

由于 epitem 对应一个被监视的文件，所以通过 base 可以方便地得到被监视的文件信息。又因为一个文件可能有多个事件发生，所以用 llink 链接这些事件。

3 epoll_create 的实现

epoll_create()的功能是创建一个 eventpollfs 文件系统的 inode 节点。具体由 ep_getfd()完成。ep_getfd()先调用 ep_eventpoll_inode()创建一个 inode 节点，然后调用 d_alloc()为 inode 分配一个 dentry。最后把 file,dentry,inode 三者关联起来。

在执行了 ep_getfd()之后，它又调用了 ep_file_init(),分配了 eventpoll 结构体，并把 eventpoll 的指针赋给 file 结构体，这样 eventpoll 就与 file 结构体关联起来了。

需要注意的是 epoll_create()的参数 size 实际上只是起参考作用，只要它不小于等于 0，就并不限制这个 epoll inode 关联的文件描述符数量。

4 epoll_ctl 的实现

epoll_ctl 的功能是实现一系列操作，如把文件与 eventpollfs 文件系统的 inode 节

点关联起来。这里要介绍一下 eventpoll 结构体，它保存在 file->f_private 中，记录了 eventpollfs 文件系统的 inode 节点的重要信息，其中成员 rbr 保存了该 epoll 文件节点监视的所有文件描述符。组织的方式是一棵红黑树，这种结构体在查找节点时非常高效。

首先它调用 ep_find()从 eventpoll 中的红黑树获得 epitem 结构体。然后根据 op 参数的不同而选择不同的操作。如果 op 为 EPOLL_CTL_ADD，那么正常情况下 epitem 是不可能在 eventpoll 的红黑树中找到的，所以调用 ep_insert 创建一个 epitem 结构体并插入到对应的红黑树中。

ep_insert()首先分配一个 epitem 对象，对它初始化后，把它放入对应的红黑树。此外，这个函数还要作一个操作，就是把当前进程放入对应文件操作的等待队列。这一步是由下面的代码完成的。

```
init_poll_funcptr(&epq.pt, ep_ptable_queue_proc);

。。。
revents = tfile->f_op->poll(tfile, &epq.pt);
```

函数先调用 init_poll_funcptr 注册了一个回调函数 ep_ptable_queue_proc，这个函数会在调用 f_op->poll 时被执行。该函数分配一个 epoll 等待队列结点 eppoll_entry：一方面把它挂到文件操作的等待队列中，另一方面把它挂到 epitem 的队列中。此外，它还注册了一个等待队列的回调函数 ep_poll_callback。当文件操作完成，唤醒当前进程之前，会调用 ep_poll_callback()，把 eventpoll 放到 epitem 的完成队列中，并唤醒等待进程。

如果在执行 f_op->poll 以后，发现被监视的文件操作已经完成了，那么把它放在完成队列中了，并立即把等待操作的那些进程唤醒。

5 epoll_wait 的实现

epoll_wait 的工作是等待文件操作完成并返回。

它的主体是 ep_poll()，该函数在 for 循环中检查 epitem 中有没有已经完成的事件，有的话就把结果返回。没有的话调用 schedule_timeout()进入休眠，直到进程被再度唤醒或者超时。

6 性能分析

epoll 机制是针对 select/poll 的缺陷设计的。通过新引入的 eventpollfs 文件系统，epoll 把参数拷贝到内核态，在每次轮询时不会重复拷贝。通过把操作拆分为 epoll_create,epoll_ctl,epoll_wait，避免了重复地遍历要监视的文件描述符。此外，由于调用 epoll 的进程被唤醒后，只要直接从 epitem 的完成队列中找出完成的事件，找出完成事件的复杂度由 O(N)降到了 O(1)。

但是 epoll 的性能提高是有前提的，那就是监视的文件描述符非常多，而且每次完成操作的文件非常少。所以，epoll 能否显著提高效率，取决于实际的应用场景。这方面需要进一步测试。

读核感悟-同步问题-同步问题概述

1 同步问题的产生背景

在多线程应用程序中，线程同步是一个非常重要的环节，没有做好线程同步，应用程序不仅难以保证结果的正确性，也难以保证应用程序的稳定性。

另一方面，在应用程序调试中，线程同步的调试难度也比较大。线程同步引起的 bug 通常具有随机性。因为线程的调度具有随机性。很多情况下，只有当产生一个特定的执行序列时，bug 才会产生。这使多线程的 bug 很难复现和追踪。

现有的调试工具中，针对多线程的工具很少。很多情况下只能使用打印日志的方式来调试多线程。这使我们在写多线程程序时必须分外小心。

pthread 线程库提供了丰富的同步手段。如互斥锁(pthread_mutex_t)，自旋锁(pthread_spin_t)，读/写自旋锁(pthread_rwlock_t)等等。

事实上，内核态同样有同步机制，而且比用户态更加丰富。了解内核态的同步机制，对了解和实现用户态的同步机制很有意义。

2 内核态与用户态的区别

现代操作系统都建筑在保护模式基础上。与实模式不同，保护模式下程序运行在不同的级别下。普通的应用程序运行在用户态，操作系统内核则运行在内核态。

与用户态不同，内核态的程序可以使用很多特权指令（如用 cli 关中断，用 lgdt 设置全局描述符表，等等），可以响应外部设备的中断，也可以对各个进程进行调度。操作系统内核运行在内核态下，负责了对所有进程的调度，对文件系统的读写，对设备中断的响应和处理，对物理内存的分配等等。由于操作系统要统一分配计算机资源，它把关键的信息如 PCB(进程控制块)，设备驱动，文件系统等等代码映射到相同的虚拟地址空间。在 i386 上，是 3G-4G。

内核态可以执行普通的系统调用函数，中断处理程序等等，这些被统称为内核控制路径（kernel control path）。内核控制路径可以互相嵌套，如系统调用中可以嵌套中断处理函数，中断处理函数内部还可以嵌套执行。这样一来，内核态的代码的执行流程就显得非常复杂。如果是在多 CPU 的环境下，同一时刻会有多个 CPU 访问同一块数据。如果把系统调用函数和中断处理程序比作线程，想象一下多个线程同时执行的复杂情况。所以内核需要自己实现同步。

为此，内核拥有丰富的同步原语，如信号量（类似于用户态的互斥锁），自旋锁（类似于用户态的自旋锁）等等。内核还拥有用户态没有的杀手锏－－关中断。在单处理器上，只要关了中断，就没有其它内核代码来干扰当前代码的执行了。

在用户态，进程使用的指令是有限的，也不能响应外部设备。这大大简化了用户态程序的设计。对每个进程来说，它拥有自己独立的用户空间，并且大部分情况下不用考虑其它进程的干涉。但是线程的出现给用户态带来了新的同步问题。pthread 库提供了丰富的同步语，如 pthread_mutex_t（用户态互斥锁）、pthread_spinlock_t（用户态自旋锁）、pthread_rwlock_t(用户态读/写自旋锁)。

了解一下内核态同步原语的实现，有助于了解和设计用户态的同步原语

读核感悟-同步问题-内核态自旋锁的实现

1 自旋锁的总述

在多处理器上，同一时间可能有多个 CPU 访问同一个数据。在多核处理器上，情况是类似的，因为对操作系统而言，一个核，甚至一个超线程（在一个核里同时执行多个代码流的技术，就好像有多个核）等价于一个“逻辑 CPU”。这时候，关中断的能力就显得有限了，因为关中断只对本地的 CPU 起作用。x86-64 处理器提供了锁总线的机制。由于多个 CPU 通过总线访问内存，Intel 提供了在指令前加 lock 前缀的方法来强制性地锁住总线，使同一时刻只有一个 CPU 能访问内存。

但是，锁总线只对一条指令起作用，对于一个代码段，光锁总线也不够了。所以内核提供了自旋锁。

自旋锁(spin lock)，顾名思义是“自己不停地旋转（循环）的锁”。它的基本思想是通过不停地检测锁的状态，来获得锁。它有以下特点：

1.由于在这段时间 CPU 一直处于忙等状态（非抢占式内核中），所以除了响应中断，无法做其它事情。所以自旋锁针对的代码段一定要能够在短时间内执行完，不能休眠（即不能让出 CPU）。

2.由于获得自旋锁的过程中不会引起进程切换，所以它的开销很小。自旋锁要比信号量快得多。

3.由于获得自旋锁的过程中不会引起进程切换，所以它不仅可以用在进程上下文（如系统调用，内核线程），也可以用在中断上下文（如中断处理函数，bottom half 等等）。

4.自旋锁可以与其它同步手段相结合，如禁止中断（对应 spin_lock_irq），禁止下半部(对应 spin_lock_bh)

5.自旋锁有非阻塞版本，如 spin_trylock()。

自旋锁有多个版本的实现

多 CPU 版本：定义了 CONFIG_SMP

它又根据 CONFIG_PREEMPT(抢占)宏定义为抢占和非抢占式。

单 CPU 版本：没有定义 CONFIG_SMP

它根据 CONFIG_PREEMPT(抢占)宏定义为抢占和非抢占式。

此外，它根据 CONFIG_DEBUG_SPINLOCK 定义了调试和非调试版本。

我们只对 SMP 非抢占版本感兴趣。

在抢占式和非抢占式内核中，spin lock 的实现之所以不同，是因为在抢占式内核中，spin lock 可以做一些优化，即在无法立即获得锁的情况下，spin lock 会允许其它进程抢占当前进程，而不只是死循环来争夺锁。这使进程的实时性有了一定的提高。但是会牺牲性能。在非抢占式内核中，当前 CPU 只是不停循环来尝试获取锁。

2 非抢占式的自旋锁

下面看一看自旋锁(spin lock)的非抢占式版本，也是服务器上经常使用的版本：

spinlock_t 结构体中包含了一个 lock 的成员，当锁释放时，它的值为 1，当有代码获得锁时，它的值小于等于 0。

获得锁的函数 spin_lock()它的关键部分的代码：

```
#define spin_lock_string \
        "\n1:\t" \
        "lock ; dec1 %0\n\t" \
```

```
        "js 2f\n" \
        LOCK_SECTION_START("") \
        "2:\t" \
        "rep;nop\n\t" \
        "cmpl $0,%0\n\t" \
        "jle 2b\n\t" \
        "jmp 1b\n" \
        LOCK_SECTION_END
```

它的伪代码如下：

```
for(;;) {
      lock--;
      if (lock == 0) {
            获得锁，执行下面的代码。
            Break;
      }
      else {
            for(;;) {
                  if (lock == 1) {
                        锁已经被释放，跳转到 trylock 处尝试获得锁。
                        Break;
                  } else 执行空操作
            }
      }
}
```

这段代码的流程是这样的：

1.首先把锁计数 lock 减一，如果不小于 0，表示获得锁，跳出外部 for 循环，执行下面的代码。

2.否则，进入内部 for 循环。在内部的 for 循环中，不停地检测 lock 计数是否为 1。为 1 表示锁已经释放，然后跑出内部 for 循环，在外部循环中再次尝试获得锁。

3.在这个过程中，关键的操作是对 lock 计数进行减 1 操作。这个操作必须锁总线，以保证减操作的正确性。

3 锁的释放

锁的释放相对比较简单，只要把 lock 计数赋值成 1 就行了。

4 与用户态的自旋锁的比较

用户态也有自旋锁。pthread 库提供了 pthread_spinlock_t。它的实现机制是与内核态的自旋锁类似的。当其它 CPU 无法获得锁时，也会旋转空等。

不同的是，由于在用户态无法关中断，如果自旋锁保护的代码段执行时间过长，可能会引发进程调度，导致自旋锁迟迟不能释放。其它 CPU 如果长时间得不到锁，就会因为时间片的耗尽而让出 CPU。所以，自旋锁保护的代码段要简短，这一点是与内核态一致的。

一种改进的策略是：在尝试获得锁的次数超过一定次数（如 100），就主动让出 CPU，这样的好处是，既保证了在其它 CPU 能够迅速释放锁的情况下能够不通过进程切换就

能获得锁，又保证了其它 CPU 不能够迅速释放锁的情况下，主动让出 CPU，不会导致 CPU 资源的浪费。

与互斥锁相比，自旋锁适合保护执行时间比较快的代码段，因为在大部分情况下，等待很短时间，不需要进行进程上下文切换就可以获得锁，会比互斥锁高效一些。

5 总结

自旋锁的思路很简单，在实现中，只需要保证对 lock 计数的写操作是原子的就可以，这过程中需要锁住总线。

在内核中，开发人员可以通过关中断等方式，保证自旋锁对应的代码段不会引起调度。用户态应用程序中，由于无法通过关中断来禁止进程调度，所以照搬内核态的那种自旋锁的实现方式，在一定条件下可能会引起性能的下降。

读核感悟-内存管理-free 命令详解

free 命令可以用来显示内存使用状况。

运行 free 的结果显示如下：

```
             total       used       free     shared    buffers     cached
Mem:       2067312    2015268      52044          0     102812     647252
-/+ buffers/cache:    1265204     802108
Swap:       979924          0     979924
```

但是关于这里的字段的具体含义，man 手册语焉不详。尤其是 buffer 和 cache 究竟是指什么，”-/+ buffers/cache”的意义，基本没有提及。不过我们可以通过源代码来获得答案。

通过 man 手册可以知道，free 命令的数据都来自于/proc/meminfo。在 proc_misc_init()中，meminfo 文件节点被初始化。读取信息是 meminfo_read_proc。在 meminfo_read_proc()中，si_meminfo（i）获得内存信息。虽然有很多字段，但是我们重点关注的是/proc/meminfo 中的 MemBuffers 和 Cached 字段，后面可以看到，free 中的 buffers 和 cached 就是对应这两个字段。

先看一下 MemBuffers 对应的是什么，在 meminfo_read_proc()这个函数中可以看到：

```
val->bufferram = nr_blockdev_pages();
```

在 linux 系统中，为了加快文件的读写，内核中提供了 page cache 作为缓存。为了加快对快设备的读写，内核中还提供了 buffer cache 作为缓存。在 2.4 内核中，这两者是分开的。这样就造成了双缓冲，因为文件读写最后还是转化为对块设备的读写。在 2.6 中，buffer cache 合并到 page cache 中，对应的页面叫作 buffer page。当进行文件读写时，如果文件在磁盘上的存储块是连续的，那么文件在 page cache 中对应的页是普通的 page，如果文件在磁盘上的存储块是不连续的，那么文件在 page cache 中对应的页是 buffer page。buffer page 与普通的 page 相比，每个页多了几个 buffer_head 结构体（个数视块的大小而定）。此外，如果对单独的块（如超级块）直接进行读写，对应的 page cache 中的页也是 buffer page。这两种页面虽然形式略有不同，但是最终他们的数据都会被封装成 bio 结构体，提交到通用块设备驱动层，统一进行 IO 调度。

此外，page cache 中还包含了 swap cache。它里面包括了曾经被交换出内存，现在又被换入内存，但是在交换设备（如交换分区）中仍然有对应的内容。之所以设置 swap

cache 是因为如果这些页面要被再度换出内存，不必执行真正的换出操作，因为交换设备中已经有了。这样做节省了不必要的 I/O。

由上可知：/proc/meminfo 中的 MemBuffers 对应块设备的缓存，即绕过文件系统直接对块设备读写时系统的缓存。

此外，在显示“Cached”的时候，使用的是这个表达式：

get_page_cache_size()-total_swapcache_pages-i.bufferram

其中 get_page_cache_size()返回的是总的 page cache 的大小，total_swapcache_pages 对应的是 swap cache 的大小，i.bufferram 对应的是块设备的缓存。所以，“Cached”表示 page cache 中除去块设备的缓存和 swap cache 后余下的部分。这部分主要保存在磁盘上的文件的页面，这些文件对应的物理存储块可能连续(普通的 page)也可能不连续(buffer page)。

可以看到，/proc/meminfo 中的 MemBuffers 和 Cached 对应的内容都是内核中的缓存，对加快文件读写很有意义。

其它的/proc/meminfo 参数详见：

http://unixfoo.blogspot.com/2008/02/know-about-procmeminfo.html

现在回过头来看一下 free。它的源代码也很容易得到。

在 debian/ubuntu 下，运行

dpkg -S free

可以很容易找到 free 对应的软件包 procps。运行

apt-get source procps

可以方便地下载到 procps 的代码。

在 proc/sysinfo.c 中，可以知道 free 中的 buffers 和 cached 就对应/proc/meminfo 中的 MemBuffers 和 Cached

在 free.c 中，可以看到，第一行中的 used 列表示系统已经使用的内存，free 表示当前空余的内存。

-/+ buffers/cache 的意思是：

第二行 used 列对应的值 = 系统已经使用的内存 -(buffers+cached)

第二行 free 列对应的值 = 系统空余的内存 + (buffers+cached)

个人理解如下：

buffers 和 cached 本质上是系统为加快文件读写所使用的缓存。没有这些 buffers 和 cached，系统也能运行，只是 IO 性能会变得非常低下。当多个应用程序访问相同的文件时，很可能这些文件已经被放在 page cache 里了，所以对这些应用程序来说 buffers 和 cached 对应的内存也是可用的，因为他们要访问的文件的数据放在 buffers 和 cached 里，而这些内存并不需要他们显式地分配。所以 free 命令输出的第一行是从系统的角度来看内存使用情况，第二行是从应用程序的角度来看内存使用情况。

读核感悟-文件读写-2.6.9 内核中的 AIO

1 AIO 概述

AIO（Asynchronous I/O，异步 I/O）是比较普遍的一种 IO 处理方式，这种方式允许在 I/O 传输完成之前，执行其它的操作。与它相对应的 IO 处理方式是同步 IO，要求在当前的 I/O 传输完成之后，再执行其它操作 。它可以看到，在相同的时间，用异步 IO 的方式能

够提交更多的 I/O 请求。

AIO 的实现形式有很多种，包括

1.轮询，通过反复检查 IO 状态来判断 IO 操作是否完成。

2.循环调用 select/poll，这两个系统调用可以用来监视一组文件描述符的 IO 操作。

3.信号，当 IO 操作完成后，应用程序会收到一个信号。应用程序只要注册相应的信号处理函数，就可以在 IO 操作完成时执行相应的操作。

Posix 标准定义了一组异步 I/O 的接口（aio_read/aio_write 等等），标准并没有规定这些接口的实现方式。在 Linux 下，这组接口是由 glibc 库函数实现的。它的实现机制如下：

异步 I/O 操作在用户态维护一个队列，叫做 runlist，每个线程从 runlist 读取请求进行处理。同时还有个 freelist 作为请求结构的缓冲池。当上层应用程序的读写操作调用 aio 读写函数时，从 freelist 中获取一个请求结构填充数据与操作类型等，然后挂入 request 链表中；随即向上层返回，表示完成此次 IO 操作。

用户态 AIO 线程从 runlist 取请求进行处理，处理完毕后，从 runlist 和 request 队列中删除请求放入 freelist，并通知应用程序操作已完成，用户可以读取数据或者继续写入数据。

POSIX 标准库提供了很多异步 IO 的函数。

库函数	功能	原型
aio_read	请求异步读操作	int aio_read(struct aiocb *aiocbp);
aio_error	检查异步请求的状态	int aio_error(const struct aiocb *aiocbp);
aio_return	获得完成的异步请求的返回状态	ssize_t aio_return(struct aiocb *aiocbp);
aio_write	请求异步写操作	int aio_write(struct aiocb *aiocbp);
aio_suspend	挂起调用进程，直到一个或多个异步请求已经完成（或失败）	int aio_suspend(const struct aiocb * const cblist[], int n, const struct timespec *timeout);
aio_cancel	取消异步 I/O 请求	int aio_cancel(int fd, struct aiocb *aiocbp);
lio_listio	发起一系列 I/O 操作	int lio_listio(int mode, struct aiocb *restrict const list[restrict], int nent, struct sigevent *restrict sig);

这是一种通过用户态实现 AIO 的方法，但是效率与在内核态实现的 AIO 有很大的区别。

因为这种实现不仅需要额外的用户态线程创建，而且要执行多次系统调用。

在2.6内核中，Linux逐步在内核态实现了异步I/O。但是，到2.6.11为止，内核态异步I/O仍然只支持O_DIRECT标记。在2.6.9中，内核态实现了以下几个异步IO的系统调用：

系统调用	功能	原型
io_setup	为当前进程初始化一个异步IO上下文	int io_setup(unsigned nr_events, aio_context_t *ctxp);
io_submit	提交一个或者多个异步IO操作	int io_submit(aio_context_t ctx_id, long nr, struct iocb **iocbpp);
io_getevents	获得未完成的异步IO操作的状态	int io_getevents(aio_context_t ctx_id, long min_nr, long nr, struct io_event *events, struct timespec *timeout);
io_cancel	取消一个未完成的异步IO操作	int io_cancel(aio_context_t ctx_id, struct iocb *iocb, struct io_event *result);
io_destroy	从当前进程删除一个异步IO上下文	int io_destroy(aio_context_t ctx);

2 内核态AIO的使用

用户态的AIO的使用很方便，只要在源代码里包含aio.h头文件，在链接时加上-lrt就可以了。

内核态的AIO的使用，则首先要保证内核支持AIO，其次，要安装libaio库。在当前的2.6.9内核中，内核支持AIO，但是系统中并没有安装libaio。所以要下载libaio的源代码，然后在源代码目录下执行 make prefix=/usr install，libaio就会安装到系统的/usr目录下。如果只是出于实验目的，可以把prefix参数改成普通用户自定义的目录，这样就不会影响其它应用程序的运行。在源代码里包含libaio.h，在链接时加上-laio就行。

读核感悟-文件读写-内核态AIO相关结构体

1 内核态AIO操作相关信息

内核态AIO的操作相关信息，是保存在对应异步I/O上下文中的，具体下文中会讲到。AIO的控制信息包括了执行一次AIO操作的所有必要信息。

结构体定义如下：

```
struct kiocb {
        struct list_head        ki_run_list;//这个字段会把kiocb挂载到对应kioctx结构体中的run_list队列中
        long                    ki_flags;
        int                     ki_users;//引用计数
```

```
        unsigned                ki_key;         /* id of this request */
        struct file             *ki_filp;//指向 struct file 结构体的指针
        struct kioctx           *ki_ctx;        /* may be NULL for sync ops */
//对应的 kioctx 结构体
        int                     (*ki_cancel)(struct kiocb *, struct io_event
*);//取消 AIO 的操作
        ssize_t                 (*ki_retry)(struct kiocb *);//要执行的 retry 操作
        void                    (*ki_dtor)(struct kiocb *);//析构函数
        struct list_head        ki_list;        /* the aio core uses this
                                                 * for cancellation */
        union {
                void __user             *user;//指向用户态的 iocb 结构体
                struct task_struct      *tsk;//指向 PCB
        } ki_obj;
        __u64                   ki_user_data;   /* user's data for completion */
        loff_t                  ki_pos;//文件指针的偏移量
        /* State that we remember to be able to restart/retry  */
        unsigned short          ki_opcode;//文件操作
        size_t                  ki_nbytes;      /* copy of iocb->aio_nbytes *///
文件操作字节数
        char                    __user *ki_buf; /* remaining iocb->aio_buf *///
异步 IO 的用户态缓冲区
        size_t                  ki_left;        /* remaining bytes *///剩余操作
的字节数
        wait_queue_t            ki_wait;//等待队列
        long                    ki_retried;     /* just for testing */retry 的次
数
        long                    ki_kicked;      /* just for testing */
        long                    ki_queued;      /* just for testing */
        void                    *private;
};
```

这个结构体是在内核中内部使用的，当执行 io_submit 时，该系统调用会把异步 IO 的请求挂载到异步 IO 上下文结构体 kioctx 的 active_reqs 队列中。

此外，在用户态，应用程序也需要把 AIO 操作相关信息传递给系统调用。

```
struct iocb {
        /* these are internal to the kernel/libc. */
        __u64   aio_data;       /* data to be returned in event's data */用来返
回异步 IO 事件信息的空间。
        __u32   PADDED(aio_key, aio_reserved1);
                                /* the kernel sets aio_key to the req # */
        /* common fields */
        __u16   aio_lio_opcode; /* see IOCB_CMD_ above */
        __s16   aio_reqprio;//请求的优先级
```

```
        __u32   aio_fildes;// 文件描述符
        __u64   aio_buf;//    用户态缓冲区
        __u64   aio_nbytes;// 文件操作的字节数
        __s64   aio_offset;// 文件操作的偏移量
        /* extra parameters */
        __u64   aio_reserved2;  /* TODO: use this for a (struct sigevent *) */
        __u64   aio_reserved3;
}; /* 64 bytes */
```

用户态的 iocb 结构体在内核态最后会转化为 kiocb 结构体，挂到相应的 kioctx 结构体中。

2 AIO 上下文：

这个结构用来保存异步 I/O 上下文，一个进程可以有多个异步 I/O 上下文，他们都被保存在 mm->ioctx_list 链表中。一个异步 I/O 上下文可以保存多个异步 I/O 操作。每个异步 I/O 上下文都有用来标志的唯一的 id 号。此外，为了方便地与用户态应用程序交互，每个异步 I/O 上下文还拥有一个 AIO ring，用来保存对应异步 I/O 操作的结果。下面会具体介绍。

对应的结构体如下：

```
struct kioctx {
        atomic_t                users;//引用计数
        int                     dead;
        struct mm_struct        *mm;//进程虚拟内存管理信息
        /* This needs improving */
        unsigned long           user_id;//用来标记 aio 上下文的 id，本质上是对应的 AIO ring 的起始地址
        struct kioctx           *next;//指向下一个 kioctx
        wait_queue_head_t       wait;//等待队列
        spinlock_t              ctx_lock;//用来保护 kioctx 的锁
        int                     reqs_active;//活跃的请求数
        struct list_head        active_reqs;    /* used for cancellation */
        struct list_head        run_list;       /* used for kicked reqs */
        unsigned                max_reqs;//最大请求数
        struct aio_ring_info    ring_info;//AIO ring
        struct work_struct      wq;//work queue 中的“任务”
};
```

3 AIO ring

为了方便地把 I/O 操作的结果返回给用户态进程，内核态的异步 I/O 设计了 AIO ring。这是一段内核态与用户态都可以访问的内存，用来保存一组 I/O 操作的结果。每个 I/O 操作的结果用 io_event 结构体表示。

之所以叫作 ring 是因为它是一个循环数组。它在虚拟地址空间中的起始地址是 mmap_base，大小是 mmap_size。ring_pages 指向一个 struct page 指针数组。nr_pages 是页面的数量。Nr,tail 是来保存循环数组的大小和数组末尾。它由一个 aio ring 缓冲区的头部(用 aio_ring 结构体表示)和一个 io_event 数组组成。

对应的内核态结构体如下：

```
struct aio_ring_info {
        unsigned long           mmap_base;//aio ring 的起始地址
        unsigned long           mmap_size;//aio ring 的大小
        struct page             **ring_pages;
        spinlock_t              ring_lock;//用来保护的锁
        long                    nr_pages;//aio ring 使用的物理页面个数
        unsigned                nr, tail;//用来记录 aio ring 循环数组起始和结束信息的下标
        struct page             *internal_pages[AIO_RING_PAGES];//指向 aio ring 对应物理页面的 struct page 结构体指针
};
```

此外，在用户态的 aio ring，还有一个头部信息。

结构体定义如下：

```
struct aio_ring {
        unsigned        id;     /* kernel internal index number */ aio 上下文的 id
        unsigned        nr;     /* number of io_events */ io events 的个数
        unsigned        head; //aio ring 循环数组的起始下标
        unsigned        tail;//aio ring 循环数组的尾部下标
        unsigned        magic;
        unsigned        compat_features;
        unsigned        incompat_features;
        unsigned        header_length;  /* size of aio_ring */ aio_ring 结构体的大小
        struct io_event         io_events[0];//接下去是 io_event 数组。
}; /* 128 bytes + ring size */
```

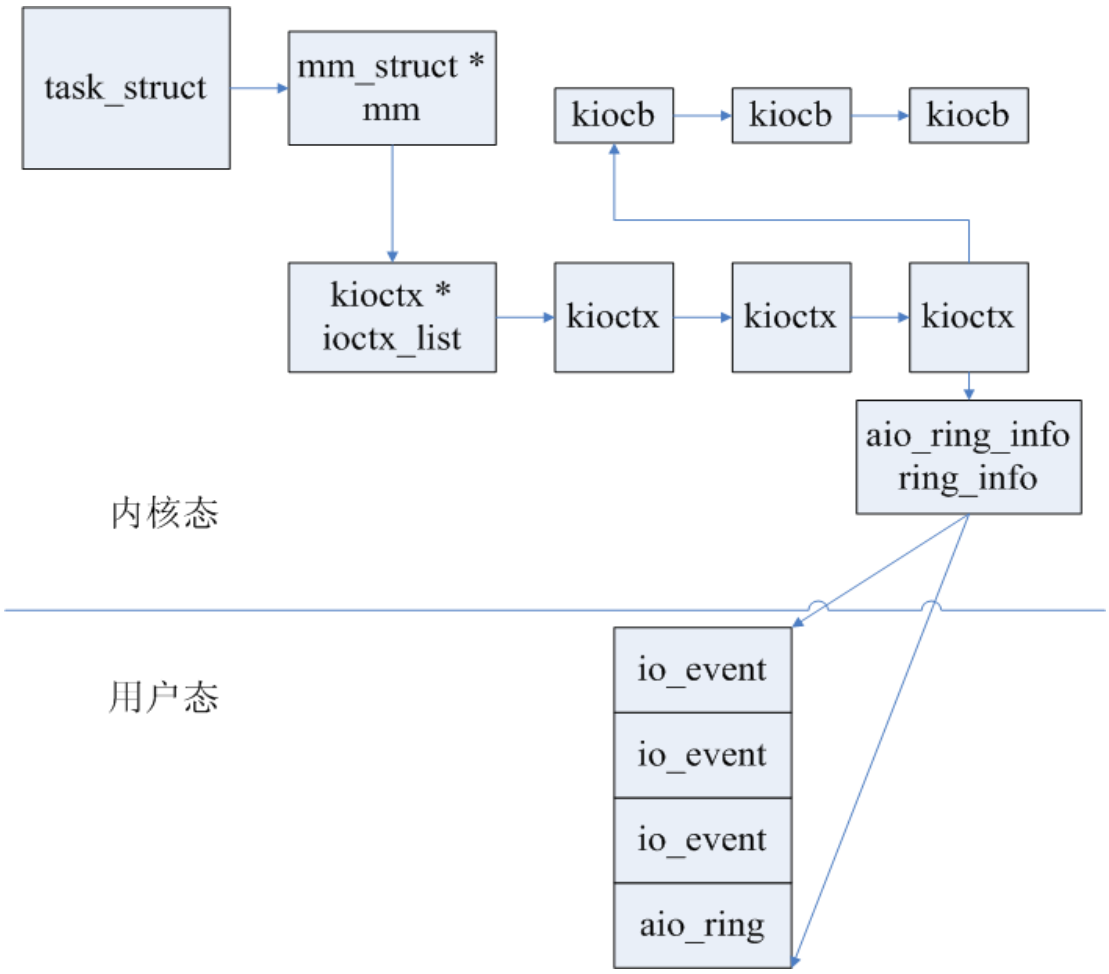
这个是 AIO ring 缓冲区的头部。与 aio_ring_info 结构体不同的是，它是保存在用户态的 AIO ring 中，用来说明 AIO ring 的基本信息，而 aio_ring_info 结构体是用来记录 AIO ring 在用户空间的位置，以及对应的物理页面的信息。

4 异步 I/O 事件的返回信息

异步 I/O 事件的结果保存在 io_event 结构体中：

```
/* read() from /dev/aio returns these structures. */
struct io_event {
        __u64           data;           /* the data field from the iocb */
        __u64           obj;            /* what iocb this event came from *///对应的用户态 iocb 结构体指针
        __s64           res;            /* result code for this event *///操作的结果
        __s64           res2;           /* secondary result */
};
```

这些结构体的关系如下图：



读核感悟-文件读写-内核态 AIO 创建和提交操作

1 AIO 上下文的创建-io_setup()

io_setup 的功能是创建一个异步 I/O 的上下文，它的原型如下：

int io_setup(unsigned nr_events, aio_context_t *ctxp);

其中，nr_events 是 AIO 上下文中同一时刻能容纳的 I/O 事件的数量(已经操作完成的不算)。Ctxp 是指向 AIO 上下文 id 的指针。AIO 上下文 id 的本质是指向 AIO 上下文的指针。

io_setup 在内核态实际调用的是 sys_io_setup。它主要分两步来创建异步 I/O 上下文：

1.调用 ioctx_alloc()创建一个新的 kioctx 结构体，并对它进行初始化。其中，对 kioctx 中的 work_struct(工作队列中的“任务”)进行初始化，把对应的执行函数设置为 aio_kick_handler。

2.调用 aio_setup_ring()对 kioctx 中的 AIO ring 信息进行初始化。

AIO ring 的开头几个字节是描述信息，接下去是一个 io_event 数组，数组元素的个数为 nr_events。计算所需的页面数量，分配这些页面，然后调用 do_mmap()把这些页面映射到进程的用户态地址空间。这样的好处是应用程序不通过系统调用就能方便地访问到 AIO ring 的信息。因为 do_mmap()并不真正为 AIO ring 分配物理页面，所以接下去还要调用 get_user_pages()为它们分配物理页面。AIO ring 的物理页面数组信息存放在 aio_ring_info 的 internal_pages 变量中。

初始化 kioctx 完毕后，把它挂到进程的内存管理结构体 mm 的 ioctx_list 中。

2 AIO 请求的提交：io_submit 实现机制

io_submit 的功能是提交异步 IO 的请求

int io_submit(aio_context_t ctx_id, long nr, struct iocb **iocbpp);

其中 ctx_id 是 AIO 上下文 id，nr 是请求个数，iocbpp 是请求数组，里面包含着 iocb 信息。

该系统调用的内核函数是 sys_io_submit()，该函数首先把用户态的异步 IO 请求 iocb 结构体拷贝到内核态，然后调用 io_submit_one()把这些 io 请求一一提交到内核。

对 kiocb 结构体的初始化：

io_submit_one 会为每个异步 IO 请求分配内核态的结构体 kiocb，然后把用户态的 iocb 结构体的内容转化为内核态的 kiocb 结构体。

对 AIO 操作函数的初始化：

调用 aio_setup_iocb 对 kiocb 中的 ki_retry 作进一步的初始化操作：读操作实际上执行的是 aio_pread 函数，写操作实际上执行的是 aio_pwrite 操作，FDSYNC 操作实际上执行的是 aio_fdsync 操作，FSYNC 操作实际上执行的是 aio_fsync 操作。

对 kiocb 初始化完成后，如果 kioctx 中的 run_list 为空，则直接调用 aio_run_iocb()来执行真正的异步 IO 操作，否则，放入 run_list 队列，执行 __aio_run_iocbs()，执行队列中的 AIO 操作。

读核感悟-文件操作-AIO 操作的执行

1.在提交时执行 AIO

异步 I/O 操作通过 io_submit()提交后，首先会在提交时尝试执行 aio_run_iocb()，若不能立即执行完毕，返回，则放到 AIO 上下文 kioctx 的 run_list 中执行 __aio_run_iocbs()。但是，大部分的 AIO 操作是在工作队列中完成的。

2.在工作队列中执行 AIO

这里，需要简单介绍一下工作队列。工作队列(work queue)是内核中的一种机制，内核开发者可以通过 create_workqueue()创建工作队列。开发者可以选择创建单个内核线程，也可以选择创建多个线程，每个 CPU 上绑定一个线程。开发者可以把任务封装到 work_struct 结构体中（包括函数，参数和定时器等等），然后调用 queue_work()把任务放到工作队列中。当工作队列的内核线程被内核调度到时，里面的任务会得到执行。

运行 ps aux 可以看到 aio/0、aio/1、aio/2 等等，这些就是内核启动时创建的工作队列在每个 CPU 核上对应的内核线程。它是如何生成的呢？在系统启动时，内核会调用 aio_setup()，该函数会创建一个名叫”aio”的工作队列。

该工作队列会在每个 CPU 核上创建一个内核线程。用户通过 ps aux 命令可以看到，aio/0、aio/1、aio/2、aio/3 等等。当内核在执行系统调用 io_submit 时不能立即把 AIO 操作处理完毕，就会把这些任务放到工作队列中，内核会在适当的时机执行它们。工作队列的对应的处理函数是 aio_kick_handler()。它会调用__aio_run_iocbs()完成 IO 操作。

3.负责 AIO 执行的核心函数 aio_run_iocb

__aio_run_iocbs()负责把每个 kiocb 结构体从 kioctx 中取出来，然后调用 aio_run_iocb()完成 AIO 操作。由此可见，最核心的函数是 aio_run_iocb()，对异步 I/O 的操作最后都会落实到这个函数。

aio_run_iocb()的功能是调用每个 AIO 操作的 retry 方法，这个方法必须要尽可能的非阻塞，防止其它等待处理的 AIO 操作等待时间过长。

aio_run_iocb()在调用 retry 方法时，会判断返回值。

当返回值是 EIOCBRETRY 时，这意味着 AIO 操作只是部分完成，需要把 kiocb 的状态标记为 Kicked，并且把该操作放入 ki_run_list 队列中。这样在工作队列中，aio 内核线程会尝试着去再次执行它。

当返回值是 EIOCBQUEUED 时，这意味着 AIO 操作是以 direct Io 的方式进行的。请求已经提交，但是还没有完成。

当返回值是其它值时，表示 AIO 操作结束，可能是正常结束，也可能是发生 I/O 错误。这时调用 aio_complete 完成 I/O 操作。

那么 AIO 操作的 retry 方法是什么呢？以读操作为例，retry 方法实际上指向的是 aio_pread 函数。它走的流程与普通文件的读写并没有什么不同。

4 AIO 操作的完成

从前面的分析可以看到，当 AIO 操作结束时，会调用 aio_complete 来进行善后操作。这个函数的功能是把 AIO 操作的返回值写到 AIO ring 中。前面提到，这是一块内核态与用户态都可以访问的缓冲区。它其实是一个循环数组，里面存放了 io_event。当应用程序调用 io_getevents 系统调用时，就会从 AIO ring 中读取 AIO 操作的状态和返回值。

读核感悟-文件读写-内核态是否支持非 direct I/O 方式的 AIO

不止一篇文档指出：2.6.9 的内核只支持 direct I/O 方式的 AIO。那么，这是否意味着无法用 AIO 的系统调用来执行非 direct I/O 的文件操作呢？（即通过 page cache 的操作？）

如果文件是以 direct I/O 的方式打开的（即 open 时有 O_DIRECT 标志）代码执行流程如下：

aio_pread->generic_file_aio_read->__generic_file_aio_read->generic_file_direct_IO

可以看到，走的流程与同步的普通文件读写是类似的。区别在于，同步的文件读写中，kiocb 的 ki_key 域的值是 KIOCB_SYNC_KEY。而 AIO 的文件读写中，kiocb 的 ki_key 的值是 0（刚提交）或者 KIOCB_C_CANCELLED(操作取消)或者 KIOCB_C_COMPLETE(操作完成)。这样，内核在底层函数中就可以判断操作是由 AIO 还是其它方式发起的，从而执行不同的操作。当内核发现发起的操作是采用 AIO 的方式并且是 direct I/O 的方式，那么，它会以异步的方式提交请求，即在请求提交之后立即返回。返回值为-EIOCBQUEUED。表示请求已经提交。而在同步的 direct I/O 的方式下，进程提交请求后要等待请求完成，然后再返回完成读写的字节数。

从这里可以看到，在发起 IO 操作后，是立即返回，还是等待 I/O 操作完成后再返回，

是同步和异步两种 I/O 方式的重要区别。

如果文件是以非 direct I/O 的方式打开的，事实证明，这种方式同样能够执行 AIO 的系统调用。代码的执行流程如下：

aio_pread->generic_file_aio_read->__generic_file_aio_read->do_generic_file_read

可以看到，与普通的文件读写的流程没有什么区别。内核并不会因为发起操作的方式是异步 I/O 而立即返回，而是与同步的 I/O 操作一样，等待 I/O 操作完成后才返回。从这个意义上说，内核对非 direct I/O 方式发起的 AIO 的实现是虚假的。

所以说，在 2.6.9 中异步 io 只支持 direct I/O。如果以非 direct I/O 的方式使用 AIO，也能够运行。但是，由于 retry 对应的方法是走普通的 I/O 操作的流程，即读写都经过 page cache，那么 retry 方法就会阻塞。如果一个文件重来没有被访问过，那么，当进程访问它时，必须等待 I/O 操作完成。这与 AIO 的本意是相悖的(AIO 的本意是进程发起 IO 操作后能立即返回，执行后续操作。)。所以，2.6.9 中也可以用非 direct I/O 的方式使用 AIO，但是并不是真正意义上的异步 I/O。

读核感悟-文件读写-内核态 AIO 的相关测试

对异步 IO 的性能测试需要考虑多个方面，包括

1.是 diretct I/O 还是非 direct I/O

2.是同步 I/O 还是异步 I/O

3.是用户态的异步 I/O 还是内核态的异步 I/O

4.是连续提交多个 AIO 请求，最后等待操作完成，还是每提交一个 I/O 请求就立即等待。

根据这些情况，编写了相应的测试程序，用不同的方式完成多种读写请求。共完成 5000 次读请求和 5000 次写请求，每次 IO 请求的数据大小为 8K。采用随机的方式进行访问。共有 10 个测试程序：包括 5 种方式（每种方式采用 diretct I/O 和非 diretct I/O 两种方式）

1-2 普通的文件读写（使用系统调用 read write 而不用 glibc 的 fread 和 fwrite）

3-4 内核态的 AIO，采用每提交一次，就等待 AIO 的返回值的方式（仅作参考）。

5-6 用户态的 AIO，采用每提交一次，就等待 AIO 的返回值的方式（仅作参考）。

7-8 内核态的 AIO，采用一次性提交大量 IO，然后等待 AIO 的返回值的方式。

9-10 用户态的 AIO，采用一次性提交大量 IO，然后等待 AIO 的返回值的方式。

1 kio_testdio 2 kio_testnodio

3 kaio_testdio 4 kaio_testnodio

5 uaio_testdio 6 uaio_testnodio

7 kaio_async_testdio 8 kaio_async_testnodio

9 uaio_async_testdio 10 uaio_async_testnodio

注意到 AIO 的测试情况采用了两种方式：

1 每提交一次，就等待 AIO 的返回值，这种方式其实本质上与同步 I/O 没有太大的差别。

2 一次性提交大量 IO，然后等待 AIO 的返回值，这种方式才体现了异步 I/O 的优势：能够在相同时间内提交更多的 I/O 请求，并且当前进程不用返回。测试结果如下：

测试程序	是否异步	是否 direct I/O	AIO 是否基于内核态	AIO 提交请求后是否等待	总时间	用户态时间	内核时间	I/O 提交时间	CPU 利用率

kio_testdio	否	是			0.73	0.00	0.15		19%
kio_testnodio	否	否			0.19	0.00	0.15		98%
kaio_testdio	是（但每次提交后都等待）	是	是	是	1.00	0.00	0.25		21%
kaio_testnodio	是（但每次提交后都等待）	否	是	是	0.20	0.00	0.17		99%
uaio_testdio	是（但每次提交后都等待）	是	否	是	1.14	0.04	0.40		42%
uaio_testnodio	是（但每次提交后都等待）	否	否	是	0.33	0.05	0.37		131%
kaio_async_testdio	是	是	是	是	0.59	0.00	0.26	0.457	36%
kaio_async_testnodio	是	否	是	是	0.34	0.00	0.22	0.323	99%
uaio_async_testdio	是	是	否	否	3.22	2.48	0.43	0.049	62%
uaio_async_testnodio	是	否	否	否	2.82	2.46	0.31	0.048	101%

从测试的结果来看(我们不考虑 3-6 的情况，因为不是常用的 AIO 的使用方式)
1.在 direct I/O 的情况下 内核态 AIO 与用户态 AIO 的比较：
速度而言：内核态 AIO 快于同步 I/O，同步 I/O 快于用户态 AIO。
内核态的 AIO 在 direct I/O 的情况下，采用一次性提交大量 AIO 操作的方式，效率要远远高于用户态的 AIO，效率提高 5 倍(3.22/0.59=6)。而且这是在内核态 AIO 的 CPU 利用率比用户态低的情况下取得的（36%<62%）。因为用户态的 AIO 需要在用户态做大量的工作从而在用户态模拟异步 I/O。它在用户态做了大量的工作，消耗了大量的用户态 CPU 时间（2.48s），而内核态 AIO 在用户态要做的事情很少(user time 几乎为 0s)，几乎所有的操

作都是在内核态完成的。
2.在 direct I/O 的情况下，内核态 AIO 与同步 I/O 的比较
采用一次性提交大量 AIO 操作的方式，效率略高于同步方式的 direct I/O，（0.59s vs 0.73s）但是注意到 CPU 利用率高于同步方式的 direct I/O（19% vs 36%）。如果判断内核态时间的话，要慢于同步方式的 direct I/O（0.15 vs 0.26）。出现这种情况并不奇怪，因为异步 IO 的优势在于能够提高 I/O 吞吐量。
内核态的 AIO 在 direct I/O 的情况下，花了较少的时间就提交了大量的 I/O 请求，而同步 I/O 则要花较多时间才能提交 I/O 请求(0.457s vs 0.77s)。
3.在非 direct I/O 的情况下 内核态 AIO 与用户态 AIO 的比较：
速度而言：同步 I/O 快于内核态 AIO 快于用户态 AIO。内核态 AIO 与用户态 AIO(0.34svs2.82s)时间差别比较大，相差 7 倍。
4.在非 direct I/O 的情况下，内核态 AIO 与同步 I/O 的比较
内核态 AIO 要慢于同步 I/O（0.19 和 0.34）。 即使只比较 I/O 提交的时间的话也要慢于同步的 I/O(0.323s vs 0.19s)。之所以出现这种情况，是因为 2.6.9 内核不支持非 direct I/O 方式的异步 I/O，这种情况下它的异步 I/O 走的流程与同步 I/O 是类似的，在读 page cache 时会阻塞。由于在提交 AIO 请求的时候会阻塞，所以导致提交请求与同步地执行 IO 没有本质区别，异步的优势体现不出来。

读核感悟-健壮的代码-让问题代码尽早暴露