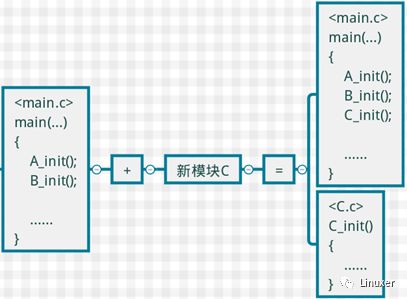
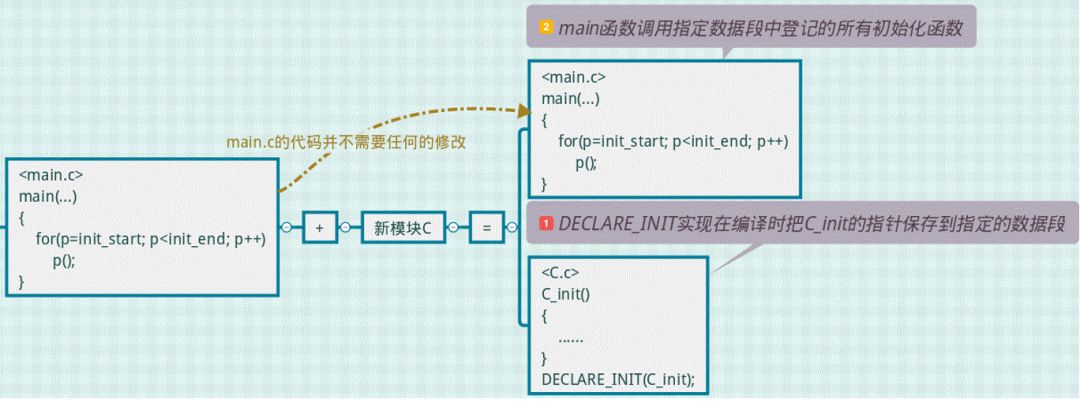
**本文详细讲解了利用\_\_attribute\_\_((section()))构建初始化函数表**

传统的应用编写时，每添加一个模块，都需要在main中添加新模块的初始化



使用\_\_attribute\_\_((section()))构建初始化函数表后，由模块告知main：“我要初始化“，添加新模块再也不需要在main代码中显式调用模块初始化接口。



以此实现main与模块之间的隔离，main不再关心有什么模块，模块的删减也不需要修改main。

那么，如何实现这个功能呢？如何实现DECLARE\_INIT呢？联想到内核驱动，所有内核驱动的初始化函数表在哪里？为什么添加一个内核驱动不需要修改初始化函数表？

下文会从 构建初始化函数表的原理分析、分析内核module\_init实现、演练练习 的3个角度给小伙伴分享。

构建初始化函数表的原理分析

\_\_attribute\_\_((section(”name“)))是gcc编译器支持的一个编译特性（arm编译器也支持此特性），实现在编译时把某个函数/数据放到name的数据段中。因此实现原理就很简单了：

1.       模块通过\_\_attribute\_\_((section("name")))的实现，在编译时把初始化的接口放到name数据段中

2.       main在执行初始化时并不需要知道有什么模块需要初始化，只需要把name数据段中的所有初始化接口执行一遍即可

首先： gcc -c  test.c -o test.o

此时编译过程中处理了\_\_atribute\_\_((section(XXX)))，把标记的变量/函数放到了test.o的XXX的数据段，可用 readelf命令查询。

最后：ld -T <ldscript> test.o –o test.bin

链接时，test.o的XXX数据段（输入段），最终保存在test.bin的XXX数据段（输出段），如此在bin中构建了初始化函数表。

由于自定义了一个数据段，而默认链接脚本缺少自定义的数据段的声明，因此并不能使用默认的链接脚本。

ld链接命令有两个关键的选项：

ld -T <script>：指定链接时的链接脚本

ld --verbose：打印出默认的链接脚本

在我们下文的演练中，我们首先通过”ld --verbose”获取默认链接脚本，然后修改链接脚本，添加自定义的段，最后在链接应用时通过“-T<script>” 指定我们修改后的链接脚本。

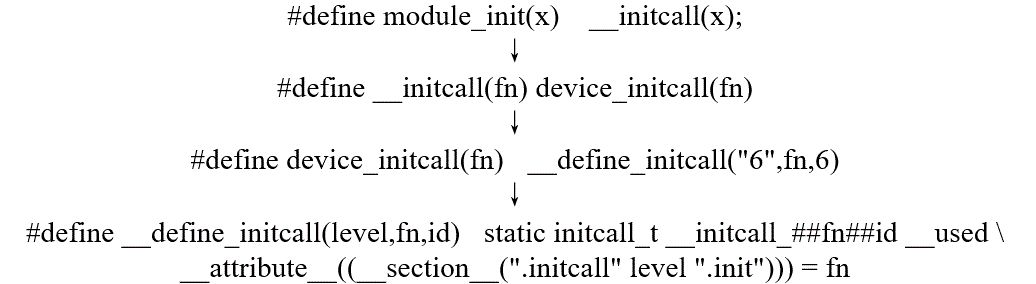
下文，我们首先分析内核module\_init的实现，最后进行应用程序的演练练习。

分析内核module\_init实现

内核驱动的初始化函数表在哪里？为什么添加一个内核驱动不需要修改初始化函数表？为什么所有驱动都需要module\_init？

**1.      module\_init的定义**

module\_init定义在<include/linux/init.h>。代码如下：



代码中使用的“\_section\_”，是一层层的宏，为了简化，把其等效理解为“section”。

分析上述代码，我们发现module\_init由\_\_attribute\_\_((section(“name”)))实现，把初始化函数地址保存到名为".initcall6.init" 的数据段中。

**2.      链接内核使用自定义的链接脚本**

我们看到内核目录最上层的Makefile，存在如下代码：

# Rule to link vmlinux - also used during CONFIG\_KALLSYMS

# May be overridden by arch/$(ARCH)/Makefile

quiet\_cmd\_vmlinux\_\_ ?= LD      $@

cmd\_vmlinux\_\_ ?= $(LD) $(LDFLAGS) $(LDFLAGS\_vmlinux) -o $@ \

      -T $(vmlinux-lds) $(vmlinux-init)                          \

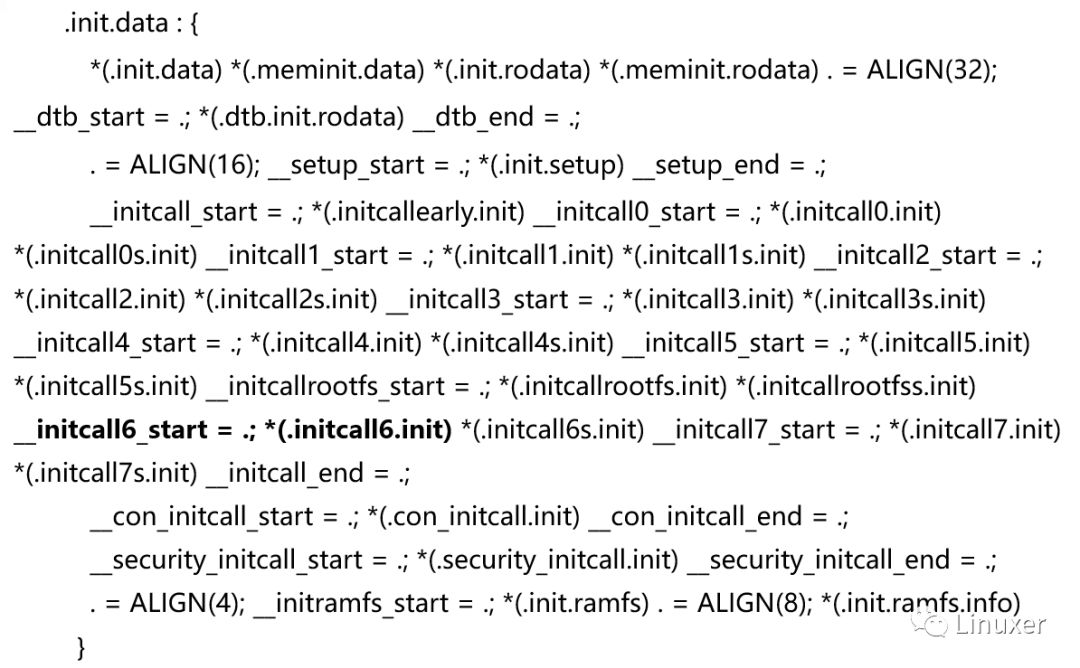
      --start-group $(vmlinux-main) --end-group                  \

      $(filter-out $(vmlinux-lds) $(vmlinux-init) $(vmlinux-main) vmlinux.o FORCE ,$^)

本文的关注点在于：-T $(vmlinux-lds)，通过“ld -T <script>”使用了定制的链接脚本。定制的链接脚本在哪里呢？在Makefile存在如下代码：

vmlinux-lds  := arch/$(SRCARCH)/kernel/vmlinux.lds

我们以”ARCH=arm“ 为例，查看链接脚本：arch/arm/kernel/vmlinux.lds：



在上述代码中，我们聚焦于两个地方：

\_\_initcall6\_start = .; ： 由\_\_initcall6\_start指向当前地址

 \*(.initcall6.init) ： 所有.o文件的.initcall6.init数据段放到当前位置

如此，“\_\_initcall6\_start”指向“.initcall6.init”数据段的开始地址，在应用代码中就可通过“\_\_initcall6\_start”访问数据段“.initcall6.init”。

是不是如此呢？我们再聚焦到文件<init/main.c>中。

“.initcall.init”数据段的使用

在<init/main.c>中，有如下代码：

static initcall\_t \*initcall\_levels[] \_\_initdata = {

      \_\_initcall0\_start,

      \_\_initcall1\_start,

      \_\_initcall2\_start,

      \_\_initcall3\_start,

      \_\_initcall4\_start,

      \_\_initcall5\_start,

      \_\_initcall6\_start,

      \_\_initcall7\_start,

      \_\_initcall\_end,

};

......

int \_\_init\_or\_module do\_one\_initcall(initcall\_t fn)

{

        ......

    if (initcall\_debug)

        ret = do\_one\_initcall\_debug(fn);

    else

        ret = fn();

        ......

}

......

static void \_\_init do\_initcall\_level(int level)

{

    ......

    for (fn = initcall\_levels[level]; fn < initcall\_levels[level+1]; fn++)

        do\_one\_initcall(\*fn);

}

按0-7的初始化级别，依次调用各个级别的初始化函数表，而驱动module\_init的初始化级别为6。在“for (fn = initcall\_levels[level]; fn <initcall\_levels[level+1]; fn++)”的for循环调用中，实现了遍历当前初始化级别的所有初始化函数。

module\_init的实现总结

通过上述的代码追踪，我们发现module\_init的实现有以下关键步骤：

1. 通过module\_init的宏，在编译时，把初始化函数放到了数据段：.initcall6.init
2. 在链接成内核的时候，链接脚本规定好了.initcall6.init的数据段以及指向数据段地址的变量：\_initcall6\_start
3. 在init/main.c中的for循环，通过\_initcall6\_start的指针，调用了所有注册的驱动模块的初始化接口
4. 最后通过Kconfig/Makefile选择编译的驱动，实现只要编译了驱动代码，则自动把驱动的初始化函数构建到统一的驱动初始化函数表

演练练习

分析了内核使用\_\_attribute\_\_((section(“name”)))构建的驱动初始化函数表，我们接下来练习如何在应用中构建自己的初始化函数表。

下文的练习参考了：https://my.oschina.net/u/180497/blog/177206

**1.      应用代码**

我们的练习代码（section.c）如下：

#include <unistd.h>

#include <stdint.h>

#include <stdio.h>

typedef void (\*init\_call)(void);

/\*

 \* These two variables are defined in link script.

 \*/

extern init\_call \_init\_start;

extern init\_call \_init\_end;

#define \_init \_\_attribute\_\_((unused, section(".myinit")))

#define DECLARE\_INIT(func) init\_call \_fn\_##func \_init = func

static void A\_init(void)

{

    write(1, "A\_init\n", sizeof("A\_init\n"));

}

DECLARE\_INIT(A\_init);

static void B\_init(void)

{

    printf("B\_init\n");

}

DECLARE\_INIT(B\_init);

static void C\_init(void)

{

    printf("C\_init\n");

}

DECLARE\_INIT(C\_init);

/\*

 \* DECLARE\_INIT like below:

 \*  static init\_call \_fn\_A\_init \_\_attribute\_\_((unused, section(".myinit"))) = A\_init;

 \*  static init\_call \_fn\_C\_init \_\_attribute\_\_((unused, section(".myinit"))) = C\_init;

 \*  static init\_call \_fn\_B\_init \_\_attribute\_\_((unused, section(".myinit"))) = B\_init;

 \*/

void do\_initcalls(void)

{

    init\_call \*init\_ptr = &\_init\_start;

    for (; init\_ptr < &\_init\_end; init\_ptr++) {

        printf("init address: %p\n", init\_ptr);

        (\*init\_ptr)();

    }

}

int main(void)

{

    do\_initcalls();

    return 0;

}

在代码中，我们做了3件事：

1. 使用\_\_attribute\_\_((section()))定义了宏：DECLARE\_INIT，此宏把函数放置到初始化函数表
2. 使用DELCARE\_INIT的宏，声明了3个模块初始化函数：A\_init/B\_init/C\_init
3. 在main中通过调用do\_initcalls函数，依次调用编译时构建的初始化函数。其中，“\_init\_start”和“\_init\_end”的变量在链接脚本中定义。

**2.      链接脚本**

通过命令”ld --verbose”获取默认链接脚本：

GNU ld (GNU Binutils for Ubuntu) 2.24

  支持的仿真：

   elf\_x86\_64

   ......

使用内部链接脚本：

==================================================

XXXXXXXX （缺省链接脚本）

==================================================

我们截取分割线”=====“之间的链接脚本保存为：ldscript.lds

在.bss的数据段前添加了自定义的数据段：

\_init\_start = .;

.myinit : { \*(.myinit) }

\_init\_end = .;

”\_init\_start“和”\_init\_end“是我们用于识别数据段开始和结束的在链接脚本中定义的变量，而.myinit则是数据段的名称，其中：

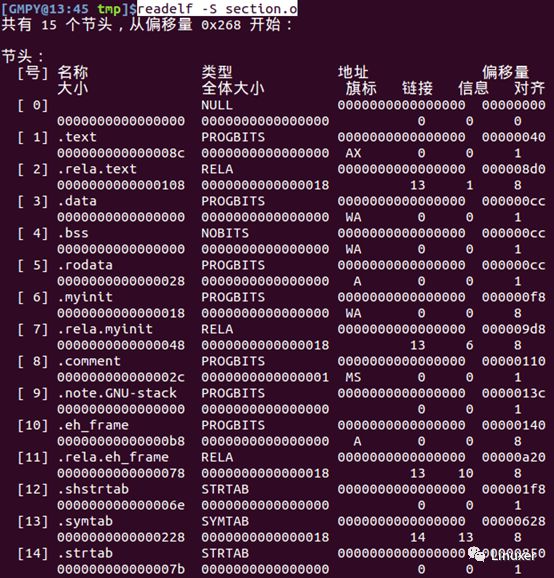
.myinit : { \*(.myinit) }：表示.o中的.myinit数据段（输入段）保存到bin中的.myinit数据段（输出段）中

前期准备充足，下面进行编译、链接、执行的演示

**3.      编译**

执行：gcc -c section.c -o section.o 编译应用源码。

执行：readelf -S section.o 查看段信息，截图如下：

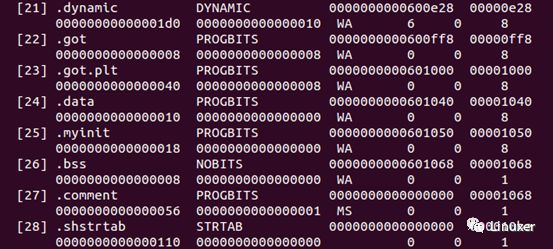


可以看到，段[6]是我们自定义的数据段

**4.      链接**

执行：gcc -T ldscript.lds section.o -o section 链接成可执行的bin文件

执行：readelf -S section 查看bin文件的段分布情况，部分截图如下：



在我链接成的可执行bin中，在[25]段中存在我们自定义的段

**5.      执行**

执行结果：

