当计算机启动以后，xen会被引导器（一般使用grub）加载。

对于xen来说,最开始运行的是一段汇编代码x86\_64.S,在这段代码的最后,调用了\_\_start\_xen()

void \_\_init \_\_start\_xen(unsigned long mbi\_p) 启动xen hypervisor

在\_\_start\_xen()有几个关注的点：

1. init\_idle\_domain() domain初始化

init\_idle\_domain() -> scheduler\_init() -> SCHED\_OP(init) -> ops.init() -> csched\_init()

1. reset\_stack\_and\_jump(init\_done) -> startup\_cpu\_idle\_loop()重置rsp后,跳转到startup\_cpu\_idle\_loop(),进入无限循环运行态,xen从启动状态进入运行状态
2. before startup\_cpu\_idle\_loop(), 在进行timer functions的初始化时调用do\_presmp\_initcalls();

这个函数的主要作用是：通过遍历\_\_initcall\_start至\_\_presmp\_initcall\_end之间所有的函数，对smp CPU进行初始化

extern initcall\_t \_\_initcall\_start, \_\_presmp\_initcall\_end, \_\_initcall\_end;

void \_\_init do\_presmp\_initcalls(void)

{

initcall\_t \*call;

for ( call = &\_\_initcall\_start; call < &\_\_presmp\_initcall\_end; call++ )

(\*call)();

}

上述实现跟普通的用户态程序不太一样，即我们可以看到首先声明了\_\_initcall\_start, \_\_presmp\_initcall\_end, \_\_initcall\_end这3个变量是外部定义的函数指针，但是当在源代码目录中搜索这3个变量时，却找不到其定义的位置。

为搞清楚这个问题，我们需要了解Linux的内核相关机制：

1. do\_presmp\_initcalls（）函数之前有修饰词\_\_init。

关于\_\_init、\_\_initdata、\_\_exit、\_\_exitdata及类似的宏（其定义往往在Linux Kernel源代码树中的位置：include/init.h），\_\_init 宏最常用的地方是驱动模块初始化函数的定义处，其目的是将驱动模块的初始化函数放入名叫.init.text的输入段。对于\_\_initdata来说，用于数据定义，目的是将数据放入名叫.init.data的输入段。其它几个宏也类似。当初始化结束后就可以释放这部分内存，这样就减少了内存的占用空间；

1. 关于initcall的一些宏定义说明

在该文件中，下面这条宏定议更为重要，它是一条可扩展的宏：

#define \_\_define\_initcall(level,fn,id) \

static initcall\_t \_\_initcall\_##fn##id \_\_attribute\_used\_\_ \

\_\_attribute\_\_ ((\_\_section\_\_(".initcall" level ".init"))) = fn

这条宏带有3个参数：level,fn, id，分析该宏可以看出：

　1) 其用来定义类型为initcall\_t的static函数指针，函数指针的名称由参数fn和id决定：\_\_initcall\_##fn##id，这就是函数指针的名称，它其实是一个变量名称。从该名称的定义方法我们其学到了宏定义的一种高级用法，即利用宏的参数产生名称，这要借助于"##"这一符号组合的作用。

　2) 这一函数指针变量放入什么输入段呢，请看\_\_attribute\_\_ ((\_\_section\_\_ (".initcall" levle ".init")))，输入段的名称由level决定，如果level="1"，则输入段是.initcall1.init，如果level="3s"，则输入段是.initcall3s.init。这一函数指针变量就是放在用这种方法决定的输入段中的。

　3) 这一定义的函数指针变量的初始值是什么叫，其实就是宏参数fn，实际使用中，fn其实就是真实定义好的函数。

该宏定义并不直接使用，请看接下来的这些宏定义：

#define pure\_initcall(fn) \_\_define\_initcall("0",fn,0)

#define core\_initcall(fn) \_\_define\_initcall("1",fn,1)

#define core\_initcall\_sync(fn) \_\_define\_initcall("1s",fn,1s)

#define postcore\_initcall(fn) \_\_define\_initcall("2",fn,2)

#define postcore\_initcall\_sync(fn) \_\_define\_initcall("2s",fn,2s)

#define arch\_initcall(fn) \_\_define\_initcall("3",fn,3)

#define arch\_initcall\_sync(fn) \_\_define\_initcall("3s",fn,3s)

#define subsys\_initcall(fn) \_\_define\_initcall("4",fn,4)

#define subsys\_initcall\_sync(fn) \_\_define\_initcall("4s",fn,4s)

#define fs\_initcall(fn) \_\_define\_initcall("5",fn,5)

#define fs\_initcall\_sync(fn) \_\_define\_initcall("5s",fn,5s)

#define rootfs\_initcall(fn) \_\_define\_initcall("rootfs",fn,rootfs)

#define device\_initcall(fn) \_\_define\_initcall("6",fn,6)

#define device\_initcall\_sync(fn) \_\_define\_initcall("6s",fn,6s)

#define late\_initcall(fn) \_\_define\_initcall("7",fn,7)

#define late\_initcall\_sync(fn) \_\_define\_initcall("7s",fn,7s)

这些宏定义出来是为了方便的使用\_\_define\_initcall宏定义的，上面每条宏第一次使用时都会产生一个新的输入段。

接下来还有一条

#define \_\_initcall(fn) device\_initcall(fn)

这一条其实只是定义了另一个别名，即平常使用的\_\_initcall其实就是这儿的device\_initcall，用它定义的函数指定位于段.initcall6.init中

1. xen内核源代码编译、链接后，形成内核可执行文件映像。该镜像中包含了进程执行的代码和数据，同时也包含了操作系统用来将映像正确装入内存并执行的信息。这些信息包含了如下文本段、数据段、init数据段、bass段等。这些数据都是由一个称为“链接器脚本”的文件链接并装入的，这个文件的功能时间这些输入信息的各段装入到指定的地址处。

我们从xen源代码中可以找到xen内核对应的lds文件是：xen/arch/x86/xen.lds.S，从中可以看到initcall部分的说明如下：

.initcall.init : {

\_\_initcall\_start = .;

\*(.initcallpresmp.init)

\_\_presmp\_initcall\_end = .;

\*(.initcall1.init)

\_\_initcall\_end = .;

} :text

可以看到smp的initcall函数放在.initcallpresmp.init段中，但是这个段并没有在xen.lds.S文件中定义。

我们发现，由presmp\_initcall()函数生成.initcallpresmp.init段，并完成smp initcall函数的注册：

#define presmp\_initcall(fn) \

static initcall\_t \_\_initcall\_##fn \_\_init\_call("presmp") = fn

进一步看\_\_init\_call的定义：

#define \_\_init\_call(lvl) \_\_used\_section(".initcall" lvl ".init")

进一步看\_\_used\_section的定义：

#define \_\_used\_section(s) \_\_used \_\_attribute\_\_((\_\_section\_\_(s)))

其中，gcc的\_\_attribute\_\_的编绎属性，\_\_attribute\_\_主要用于改变所声明或定义的函数或数据的特性，它有很多子项，用于改变作用对象的特性。比如对函数，noline将禁止进行内联扩展、noreturn表示没有返回值、pure表明函数除返回值外，不会通过其它（如全局变量、指针）对函数外部产生任何影响。但这里我们比较感兴趣的是对代码段起作用子项section。

\_\_attribute\_\_的section子项的使用格式为：

\_\_attribute\_\_((section("section\_name")))

其作用是将作用的函数或数据放入指定名为"section\_name"输入段

这里还要注意一下两个概念：输入段和输出段

输入段和输出段是相对于要生成最终的elf或binary时的Link过程说的，Link过程的输入大都是由源代码编绎生成的目标文件.o，那么这些.o 文件中包含的段相对link过程来说就是输入段，而Link的输出一般是可执行文件elf或库等，这些输出文件中也包含有段，这些输出文件中的段就叫做输出段。输入段和输出段本来没有什么必然的联系，是互相独立，只是在Link过程中，Link程序会根据一定的规则（这些规则其实来源于Link Script），将不同的输入段重新组合到不同的输出段中，即使是段的名字，输入段和输出段可以完全不同。

其用法举例如下：

int var \_\_attribute\_\_((section(".xdata"))) = 0;

这样定义的变量var将被放入名为.xdata的输入段（注意：\_\_attribute\_\_这种用法中的括号很严格，这里的几个括号一个也不能少）

static int \_\_attribute\_\_((section(".xinit"))) functionA(void)

{

.....

}

这个例子将使函数functionA被放入名叫.xinit的输入段。

需要着重注意的是，\_\_attribute\_\_的section属性只指定对象的输入段，它并不能影响所指定对象最终会放在可执行文件的什么段。