### 1 核心初始化

LINUX 核心初始化过程应该不太会再变动，一步步走完一个函式以及函式中的函式与数据结构，建立核心初始化的路径与层次图像。

从 linux/init/main.c 的函式 start\_kernel展开。

### 2. 核心起始函式

核心起始程序包括核心起头程序、I386 核心起始程序、核心起始程序。 其中，

核心起始程序会处理比较重要的核心模块群初始化。

最后执行两个核心行程，核心初始化行程、核心行程建立行程。

核心初始化行程会等核心行程建立行程启动后，再继续执行核心初始化工作。 等核心初始化行程完成后，核心便宣告启动完成。

核心起始函式 start\_kernel 的档案位置是 linux/init/main.c。 核心起始函式的工作是初始化 LINUX 核心功能模块群，包括CPU组态、多任务核心、内存

管理器、分页管理器等等。 这个过程会建立核心运作所需要的所有数据结构。

<http://www.cnblogs.com/cslunatic/archive/2013/05/11/3072811.html>

#### linux/init/main.c

asmlinkage void \_\_init start\_kernel(void){

char \* command\_line;

**extern const struct kernel\_param \_\_start\_\_\_param[], \_\_stop\_\_\_param[];**

##### asmlinkage

asmlinkage是一个宏定义，它的作用主要有两个，一个是让传送给函数的参数全部使用栈式传送，不用寄存器来传送。

因为寄存器的个数有限，使用栈可以传送更多的参数，比如在X86的CPU里只能使用6个寄存器传送，只能传送4个参数，而使用栈就没有这种限制；另外一个用处是声明这个函数是给汇编代码调用的。

不过在ARM体系里，并没有使用栈传送参数的特性，原因何在？由于ARM体系的寄存器个数比较多，多达13个，这样绝大多数的函数参数都可以通过寄存器来传送，达到高效的目标。

ARM

#define asmlinkage CPP\_ASMLINKAGE

#define CPP\_ASMLINKAGE extern "C"

#define CPP\_ASMLINKAGE

X86

#define asmlinkage CPP\_ASMLINKAGE \_\_attribute\_\_((regparm(0)))

##### \_\_init

接着来看另外一个宏定义\_\_init，这个宏定义主要用来标志这个函数编译出来的目标代码放在那一段里。

对于应用程序的编译和连接，不需要作这样的考虑，但是对于内核代码来说，就需要了，因为不同的段代码有着不同的作用，比如初始化段的代码，当系统运行正常以后，这段代码就没有什么用了，聪明的做法就是回收这段代码占用的内存，让内核的代码占最少的内存。

还有另外一个作用，比如同一段的代码都是编译在一起，让相关联的代码尽可能同在一片内存里，这样当CPU加载代码到缓存时，就可以一起命中，提高缓存的命中率，这样就大大提高代码的执行速度。

宏\_\_init定义在文件./include/linux/init.h里，代码如下：

/\* These are for everybody (although not all archs will actually

discard it in modules) \*/

#define \_\_init \_\_section(.init.text) \_\_cold notrace

使用这个宏声明的函数，编译时就会把目标代码放到段.init.text里，这段都是放置初始化的代码。  
最后看到声明一个字符的指针command\_line，这个指针是指向命令行参数的指针，主要用来指向引导程序传送给内核的命令行参数，在后面的函数setup\_arch和函数setup\_command\_line就会对它进行处理。

##### kernel\_param

这里关键是这两个变量的地址是如何确定的。

这两个变量为地址指针，指向内核启动参数处理相关结构体段在内存中的位置（虚拟地址）。

这里是外部变量，定义的位置在arch/../../vmlinux.lds.S，而大多数平台是放到kernel\include\asm-generic\vmlinux.lds.h中，定义如下：

/\* 内建模块的参数处理段. \*/ \

\_\_param : AT(ADDR(\_\_param) - LOAD\_OFFSET) { \

VMLINUX\_SYMBOL(\_\_start\_\_\_param) = .; \

\*(\_\_param) \

VMLINUX\_SYMBOL(\_\_stop\_\_\_param) = .; \

}

对于ARM平台，似乎位于 kernel\include\asm-generic\vmlinux.lds.h

这个段中数据的数据结构：

kernel\_param结构体的定义是：

struct kernel\_param\_ops {

/\* Returns 0, or -errno. arg is in kp->arg. \*/

int (\*set)(const char \*val, const struct kernel\_param \*kp);

/\* Returns length written or -errno. Buffer is 4k (ie. be \*/

int (\*get)(char \*buffer, const struct kernel\_param \*kp);

/\* Optional function to free kp->arg when module unloaded. \*/

void (\*free)(void \*arg);

};

/\* Flag bits for kernel\_param.flags \*/

#define KPARAM\_ISBOOL 2

struct kernel\_param {

const char \*name;

const struct kernel\_param\_ops \*ops;

u16 perm;

u16 flags;

union {

void \*arg;

const struct kparam\_string \*str;

const struct kparam\_array \*arr;

};

};

/\* Special one for strings we want to copy into \*/

struct kparam\_string {

unsigned int maxlen;

char \*string;

};

/\* Special one for arrays \*/

struct kparam\_array

{

unsigned int max;

unsigned int elemsize;

unsigned int \*num;

const struct kernel\_param\_ops \*ops;

void \*elem;

};

smp\_setup\_processor\_id();

#### smp\_setup\_processor\_id

这个函数是针对SMP处理器的，经查阅资料，其作用是**获取当前CPU的的硬件ID。**

如果不是多处理器构架，在其他文件中就不会定义这个函数，此时使用本文件定义的弱引用函数：

void \_\_init \_\_weak smp\_setup\_processor\_id(void)

{

**如果内核只是有单处理器系统，smp\_setup\_processor\_id()函数是是空的，不必要做任保的处理。**

}

##### smp\_setup\_processor\_id 和smp\_processor\_id

smp\_setup\_processor\_id()函数可以不调用setup\_arch()初始化函数就可以使用，而smp\_processor\_id()函数是一定要调用setup\_arch()初始化函数后，才能使用。

smp\_setup\_processor\_id()函数是直接获取对称多处理器的ID，而smp\_processor\_id()函数是获取变量保存的处理器ID，因此一定要调用初始化函数。

由于smp\_setup\_processor\_id()函数不用调用初始化函数，可以放在内核初始化start\_kernel函数的最前面使用，而函数smp\_processor\_id()只能放到setup\_arch()函数调用的后面使用了。

smp\_setup\_processor\_id()函数每次都要中断CPU去获取ID，这样效率比较低。

#### lockdep\_init

lockdep\_init();

由于内核大量使用锁来进行多进程、多处理器的同步操作，那么死锁就会在代码不合理时出现，这时要知道那个锁造成的，真是比较困难的。

遇到这种情况，就需要想办法知道那个锁造成的，因此就需要跟踪锁的使用状态，以便发现出错时，把锁的状态打印出来。

造成死锁的情况有很多，主要有以下几种：

1. 同一个进程递归地加锁同一把锁。

2. 同一把锁在两次中断里加锁。

3. 几把锁形成一个闭环死锁。

##### 依赖锁串行初始化

依赖锁的档案位置是 linux/kernel/lockdep.c。 依赖锁初始化函式会初始化两个串行，依赖锁类别哈希表串行和依赖锁链哈希表串行，完成后设定初始化旗号为真，表示依赖锁已初始化完成。

linux/kernel/lockdep.c

01 void lockdep\_init(void){

02 int i;

03

04 if (lockdep\_initialized) return;

05 for (i = 0; i < CLASSHASH\_SIZE; i++) INIT\_LIST\_HEAD(classhash\_table + i);

06 for (i = 0; i < CHAINHASH\_SIZE; i++) INIT\_LIST\_HEAD(chainhash\_table + i);

07 lockdep\_initialized = 1;

08 }

|  |  |
| --- | --- |
| 行号 | 说明 |
| 04 | 当 Lockdep 已经初始化完成，函式回返。 |
| 05 | 依赖锁类别哈希表串行初始化。这里使用宏处理串行头 INIT\_LIST\_HEAD。 |
| 06 | 锁依赖链哈希表串行初始化，。 |
| 07 | 当完成串行初始化，设定 Lockdep 初始化旗号为一，表示已完成初始化。 |

###### 依赖锁依赖哈希串行表：

linux/kernel/lockdep.c

01 #define MAX\_LOCKDEP\_KEYS\_BITS 13

02 #define CLASSHASH\_BITS (MAX\_LOCKDEP\_KEYS\_BITS - 1)

03 #define CLASSHASH\_SIZE (1UL << CLASSHASH\_BITS)

04 #define \_\_classhashfn(key) hash\_long((unsigned long)key, CLASSHASH\_BITS)

05 #define classhashentry(key) (classhash\_table + \_\_classhashfn((key)))

06 static struct list\_head classhash\_table[CLASSHASH\_SIZE];

行号 说明

01 依赖锁钥匙位数 13 位。

02 类别哈希位数 12 位，。

03 类别哈希体积 4096。

04 类别哈希函式，。

05 类别哈希入口地址，。

05 类别哈希表结构数组，。

###### 依赖锁链哈希串行表：

linux/kernel/lockdep.c

01 #define MAX\_LOCKDEP\_CHAINS\_BITS 15

02 #define CHAINHASH\_BITS (MAX\_LOCKDEP\_CHAINS\_BITS-1)

03 #define CHAINHASH\_SIZE (1UL << CHAINHASH\_BITS)

04 #define \_\_chainhashfn(chain) hash\_long(chain, CHAINHASH\_BITS)

05 #define chainhashentry(chain) (chainhash\_table + \_\_chainhashfn((chain)))

06 static struct list\_head chainhash\_table[CHAINHASH\_SIZE];

行号 说明

01 依赖锁链位数 13 位。

02 链哈希位数 12 位，。

03 链哈希体积 4096。

04 链哈希函式，。

05 链哈希入口地址，。

05 链哈希表结构数组，。

debug\_objects\_early\_init();

在启动早期初始化hash buckets 和链接静态的 pool objects对象到 poll 列表. 在这个调用完成后 object tracker 已经开始完全运作了.

boot\_init\_stack\_canary();

canary值的是用于防止栈溢出攻击的堆栈的保护字 。

参考资料： GCC 中的编译器堆栈保护技术

cgroup\_init\_early();

cgroup: 它的全称为control group.即一组进程的行为控制.

该函数主要是做数据结构和其中链表的初始化

参考资料： Linux cgroup机制分析之框架分析

#### 控制群早期初始化

控制群(control group)，简称 cgroup，是一个行程群聚(process grouping)系统，用于量测系统效能。 也就是透过将行程群聚的方式，统计行程群集体消耗的 CPU 运算资源。

linux/kernel/cgroup.c

01 int \_\_init cgroup\_init\_early(void){

02 int i;

03 atomic\_set(&init\_css\_set.refcount, 1);

04 INIT\_LIST\_HEAD(&init\_css\_set.cg\_links);

05 INIT\_LIST\_HEAD(&init\_css\_set.tasks);

06 INIT\_HLIST\_NODE(&init\_css\_set.hlist);

07 css\_set\_count = 1;

08 init\_cgroup\_root(&rootnode);

09 root\_count = 1;

10 init\_task.cgroups = &init\_css\_set;

11 init\_css\_set\_link.cg = &init\_css\_set;

12 init\_css\_set\_link.cgrp = dummytop;

13 list\_add(&init\_css\_set\_link.cgrp\_link\_list,&rootnode.top\_cgroup.css\_sets);

14 list\_add(&init\_css\_set\_link.cg\_link\_list,&init\_css\_set.cg\_links);

15 for (i = 0; i < CSS\_SET\_TABLE\_SIZE; i++) INIT\_HLIST\_HEAD(&css\_set\_table[i]);

16 for (i = 0; i < CGROUP\_BUILTIN\_SUBSYS\_COUNT; i++) {

17 struct cgroup\_subsys \*ss = subsys[i];

18

19 BUG\_ON(!ss->name);

20 BUG\_ON(strlen(ss->name) > MAX\_CGROUP\_TYPE\_NAMELEN);

21 BUG\_ON(!ss->create);

22 BUG\_ON(!ss->destroy);

23 if (ss->subsys\_id != i) {

24 printk(KERN\_ERR "cgroup: Subsys %s id == %d\n",ss->name, ss->subsys\_id);

25 BUG();

26 }

27 if (ss->early\_init) cgroup\_init\_subsys(ss);

28 }

29 return 0;

30 }

local\_irq\_disable();

关闭系统总中断（底层调用汇编指令）

在ARM体系里主要就是对CPSR寄存器进行操作。

early\_boot\_irqs\_disabled = true;

设置系统中断的关闭标志（bool全局变量）

中断依然被禁用。做必要的设置后，再使能它

标记内核还在早期初始化代码阶段，并且中断在关闭状态，如果有任何中断打开或请求中断的事情出现，都是会提出警告，以便跟踪代码错误情况。

早期代码初始化结束之后，就会调用函数early\_boot\_irqs\_on来设置这个标志为真。

tick\_init();

初始化内核时钟系统

-->clockevents\_register\_notifier(&tick\_notifier)

往内核通知链中注册内核时钟时间的通知函数

参考资料：《Linux 时钟处理机制》 《Linux 时钟管理》

初始化时钟事件管理器的回调函数，比如当时钟设备添加时处理。

在内核里定义了时钟事件管理器，主要用来管理所有需要周期性地执行任务的设备。

boot\_cpu\_init();

激活当前CPU（在内核全局变量中将当前CPU的状态设为激活状态）

参考资料：《激活第一个CPU》

在多CPU的系统里，内核需要管理多个CPU，那么就需要知道系统有多少个CPU，在内核里使用cpu\_present\_map位图表达有多少个CPU，每一位表示一个CPU的存在。

如果是单个CPU，就是第0位设置为1。

虽然系统里有多个CPU存在，但是每个CPU不一定可以使用，或者没有初始化，在内核使用cpu\_online\_map位图来表示那些CPU可以运行内核代码和接受中断处理。

随着移动系统的节能需求，需要对CPU进行节能处理，比如有多个CPU运行时可以提高性能，但花费太多电能，导致电池不耐用，需要减少运行的CPU个数，或者只需要一个CPU运行。

这样内核又引入了一个cpu\_possible\_map位图，表示最多可以使用多少个CPU。

在本函数里就是依次设置这三个位图的标志，让引导的CPU物理上存在，已经初始化好，最少需要运行的CPU。

page\_address\_init();

高端内存相关，未定义高端内存的话为空函数

在这里引入了高端内存的概念，那么什么叫做高端内存呢？为什么要使用高端内存呢？其实高端内存是相对于低端内存而存在的，那么先要理解一下低端内存了。  
在32位的系统里，最多能访问的总内存是4G，其中3G空间给应用程序，而内核只占用1G的空间。  
因此，内核能映射的内存空间，只有1G大小，但实际上比这个还要小一些，大概是896M，另外128M空间是用来映射高端内存使用的。  
因此0到896M的内存空间，就叫做低端内存，而高于896M的内存，就叫高端内存了。  
如果系统是64位系统，当然就没未必要有高端内存存在了，因为64位有足够多的地址空间给内核使用，访问的内存可以达到10G都没有问题。  
在32位系统里，内核为了访问超过1G的物理内存空间，需要使用高端内存映射表。  
比如当内核需要读取1G的缓存数据时，就需要分配高端内存来使用，这样才可以管理起来。  
使用高端内存之后，32位的系统也可以访问达到64G内存。  
在移动操作系统里，目前还没有这个必要，最多才1G多内存。

printk(KERN\_NOTICE "%s", linux\_banner);

打印内核版本信息，也就是平时我们在内核启动时在串口中看到的：

Linux version 2.6.37+ (tekkaman@tekkaman-desktop) (gcc version 4.3.3 (Sourcery G++ Lite 2009q1-203) ) #40 Tue Mar 20 17:49:58 CST 2012

linux\_banner是在文件kernel/init/version.c里定义，这个字符串是由编译脚本自动生成。

#### setup\_arch

setup\_arch(&command\_line);

内核构架相关初始化函数,可以说是非常重要的一个初始化步骤。其中包含了处理器相关参数的初始化、内核启动参数（tagged list）的获取和前期处理、内存子系统的早期的初始化（bootmem分配器）。

对于ARM构架来说，这个函数位于：arch/arm/kernel/setup.c

以后会详细分析这个函数。

**对内核架构进行初始化。**

**再次获取CPU类型和系统架构，分析引导程序传入的命令行参数，进行页面内存初始化，处理器初始化，中断早期初始化等等。**

mm\_init\_owner(&init\_mm, &init\_task);

mm\_init\_cpumask(&init\_mm);

初始化代表内核本身内存使用的管理结构体init\_mm。

ps：每一个任务都有一个mm\_struct结构以管理内存空间，init\_mm是内核的mm\_struct，其中：

设置成员变量\* mmap指向自己，意味着内核只有一个内存管理结构;

设置\* pgd=swapper\_pg\_dir，swapper\_pg\_dir是内核的页目录(在arm体系结构有16k， 所以init\_mm定义了整个kernel的内存空间)。

这些内容涉及到内存管理子系统，以后再仔细分析。

setup\_command\_line(command\_line);

对cmdline进行备份和保存：

/\* 为处理的command line备份 (例如eg. 用于 /proc) \*/

char \*saved\_command\_line;

/\* 用于参数处理的command line \*/

static char \*static\_command\_line;

setup\_nr\_cpu\_ids();

setup\_per\_cpu\_areas();

smp\_prepare\_boot\_cpu();

针对SMP处理器的内存初始化函数，如果不是SMP系统则都为空函数。

他们的目的是给每个CPU分配内存，并拷贝.data.percpu段的数据。为系统中的每个CPU的per\_cpu变量申请空间并为boot CPU设置一些数据。

在SMP系统中，在引导过程中使用的CPU称为boot CPU

build\_all\_zonelists(NULL);

page\_alloc\_init();

设置内存管理相关的node（节点，每个CPU一个内存节点）和其中的zone（内存域，包含于节点中，如）数据结构,以完成内存管理子系统的初始化，并设置bootmem分配器。

page\_alloc\_init函数暂时不知其目的

printk(KERN\_NOTICE "Kernel command line: %s\n", boot\_command\_line);

parse\_early\_param();

parse\_args("Booting kernel", static\_command\_line, \_\_start\_\_\_param,

\_\_stop\_\_\_param - \_\_start\_\_\_param,

&unknown\_bootoption);

打印从内核启动参数中获取的cmdline字符串。

解析cmdline中的启动参数。

setup\_log\_buf(0);

使用大量bootmem分配，且必须先于kmem\_cache\_init()

1. 以上注释的含义在于bootmem是内核启动时使用的临时内存分配器。之后由slab接替。
2. kmem\_cache\_init()初始化了内核高速缓存分配器（slab分配器），这个函数标标志着bootmem的终结，同时内核的内存管理系统正式启用了。
3. 所以在kmem\_cache\_init()之后，bootmem的API不再可用，所以bootmem分配必须先于kmem\_cache\_init()。

使用bootmem分配一个记录启动信息的缓存区

pidhash\_init();

使用bootmem分配并初始化PID散列表，由PID分配器管理空闲和已指派的PID

进程ID的HASH表初始化，这样可以提供通PID进行高效访问进程结构的信息。

LINUX里共有四种类型的PID，因此就有四种HASH表相对应。

vfs\_caches\_init\_early();

前期VFS缓存初始化（虚拟文件系统的缓存初始化。）

sort\_main\_extable();

对内核异常表（ exception table ）按照异常向量号大小进行排序。

对内核内部的异常表进行堆排序，以便加速访问。

trap\_init();

对内核陷阱异常进行初始化，在ARM构架中为空函数。

对异常进行初始化，在ARM系统里是空函数，没有任何的初始化。

mm\_init();

初始化内核内存分配器，其包含6个子函数，作用如下：

1、page\_cgroup\_init\_flatmem(); 获取page\_cgroup 所需内存

2、mem\_init(); 关闭并释放bootmem分配器，打印内存信息。在内核启动时看到的类似如下信息，就是其子函数mem\_init输出的：

Memory: 86MB 39MB = 125MB total

Memory: 120768k/120768k available, 99392k reserved, 0K highmem

Virtual kernel memory layout:

vector : 0xffff0000 - 0xffff1000 ( 4 kB)

fixmap : 0xfff00000 - 0xfffe0000 ( 896 kB)

DMA : 0xffc00000 - 0xffe00000 ( 2 MB)

vmalloc : 0xde800000 - 0xf8000000 ( 408 MB)

lowmem : 0xc0000000 - 0xde400000 ( 484 MB)

pkmap : 0xbfe00000 - 0xc0000000 ( 2 MB)

modules : 0xbf000000 - 0xbfe00000 ( 14 MB)

.init : 0xc0008000 - 0xc003b000 ( 204 kB)

.text : 0xc003b000 - 0xc04c0000 (4628 kB)

.data : 0xc04c0000 - 0xc0501a80 ( 263 kB)

3、kmem\_cache\_init(); 初始化slab分配器

4、percpu\_init\_late(); PerCPU变量系统后期初始化

5、pgtable\_cache\_init();页表缓存初始化，对于ARM，“介是一个空函数”

6、vmalloc\_init();初始化虚拟内存分配器

sched\_init();

在开启任何中断(比如定时器中断)前设置调度器。完整的拓扑设置发生在smp\_init()中

-但与此同时，我们仍然有一个正常运作的调度器。

初始化调度器数据结构，并创建运行队列

对进程调度器进行初始化，比如分配调度器占用的内存，初始化任务队列，设置当前任务的空线程，当前任务的调度策略为CFS调度器。

**禁用强占 - 早期启动时的调度是极为脆弱的，**

**直到cpu\_idle()的首次运行。**

preempt\_disable();

关闭优先级调度。由于每个进程任务都有优先级，目前系统还没有完全初始化，还不能打开优先级调度。

if (!irqs\_disabled()) {

printk(KERN\_WARNING "start\_kernel(): bug: interrupts were "

"enabled \*very\* early, fixing it\n");

local\_irq\_disable();

}

判断是否过早打开中断，如果是这样，就会提示，并把中断关闭。

在启动的初期关闭抢占和中断。

idr\_init\_cache();

为IDR机制分配缓存，主要是为struct idr\_layer结构体分配空间

参考资料：[《浅析linux内核中的idr机制》](http://blog.sina.com.cn/s/blog_476d8cf30100nhfc.html)

**创建IDR机制的内存缓存对象。所谓的IDR就是整数标识管理机制（integerIDmanagement）。**

**引入的主要原因是管理整数的ID与对象的指针的关系，由于这个ID可以达到32位，也就是说，如果使用线性数组来管理，那么分配的内存太大了；如果使用线性表来管理，又效率太低了，所以就引用IDR管理机制来实现这个需求。**

perf\_event\_init();

CPU性能监视机制初始化

此机制包括CPU同一时间执行指令数，cache miss数，分支预测失败次数等性能参数

rcu\_init();

内核RCU（Read-Copy Update：读取-复制-更新）机制初始化

初始化直接读拷贝更新的锁机制。RCU主要提供在读取数据机会比较多，但更新比较的少的场合，这样减少读取数据锁的性能低下的问题。

radix\_tree\_init();

内核radix树算法初始化(初始化radix树，radix树基于二进制键值的查找树。)

early\_irq\_init();

init\_IRQ();

硬件中断系统初始化：

early\_irq\_init();前期外部中断描述符初始化，主要初始化数据结构。

init\_IRQ;对应构架特定的中断初始化函数，在ARM构架中：

void \_\_init init\_IRQ(void)

{

   machine\_desc->init\_irq();

}

也就是运行设备描述结构体中的init\_irq函数，此函数一般在板级初始化文件（arch/\*/mach-\*/board-\*.c）中定义。

prio\_tree\_init();

初始化内核基于radix数的优先级搜索树（PST），主要是对其结构体进行初始化。

初始化优先搜索树，主要用在内存反向搜索方面。

init\_timers();

主要初始化引导CPU的时钟相关的数据结构，注册时钟的回调函数，当时钟到达时可以回调时钟处理函数，最后初始化时钟软件中断处理。

hrtimers\_init();

初始化高精度的定时器，并设置回调函数。

softirq\_init();

初始化软件中断，**软件中断与硬件中断区别就是中断发生时，软件中断是使用线程来监视中断信号，而硬件中断是使用CPU硬件来监视中断。**

timekeeping\_init();

初始化系统时钟计时，并且初始化内核里与时钟计时相关的变量。

time\_init();

以上几个函数主要是初始化内核的软中断及时钟机制：

前面几个函数主要是注册一些内核通知函数到cpu和hotcpu通知链，并开启部分软中断（tasklet等）。

最后的time\_init是构架相关的，旨在开启一个硬件定时器，开始产生系统时钟。对于ARM构架：

void \_\_init time\_init(void)

{

system\_timer = machine\_desc->timer;

system\_timer->init();

#ifdef CONFIG\_HAVE\_SCHED\_CLOCK

sched\_clock\_postinit();

#endif

}

其实就是调用板级初始化文件（arch/arm/mach-\*/board-\*.c）中定义“设备描述结构体”中的timer成员的初始化函数。

profile\_init();

初始化内核profile子系统，她是内核的性能调试工具。

call\_function\_init();

初始化所有CPU的call\_single\_queue（具体作用还没搞明白），并注册CPU热插拔通知函数到CPU通知链中。

if (!irqs\_disabled())

printk(KERN\_CRIT "start\_kernel(): bug: interrupts were " "enabled early\n");

early\_boot\_irqs\_disabled = false;

设置内核还在早期初始化阶段的标志，以便用来调试时输出信息。

local\_irq\_enable();

检测硬件中断是否开启，如果开启了就打印出警告。

设置启动早期IRQ使能标志，允许IRQ使能。

最后开启总中断（ARM构架是这样，其他构架可能也是这个意思）。

**是打开本CPU的中断，也即允许本CPU处理中断事件，在这里打开引CPU的中断处理。如果有多核心，别的CPU还没有打开中断处理。**

gfp\_allowed\_mask = \_\_GFP\_BITS\_MASK;

中断已经开启，因此所有GFP分配是安全的.

开启所有GFP分配允许标志

GFP(get free page)

kmem\_cache\_init\_late();

slab分配器的后期初始化。如果使用的是slob或slub，则为空函数。

console\_init();

初始化控制台，这样可以早点看到启动信息，避免出错时无法查找原因。

**用来初始化控制台，从这个函数之后就可以输出内容到控制台了。**

**在这个函数初化之前，都没有办法输出内容，就是输出，也是写到输出缓冲区里，缓存起来，等到这个函数调用之后，就立即输出内容。**

if (panic\_later) panic(panic\_later, panic\_param);

检查内核恐慌标志。如果出了问题，就打印信息。

lockdep\_info();

打印lockdep调试模块的信息。

/\*

\* 当irq使能的时候必须运行这个函数，因为它也要自检

\* [hard/soft]-irqs 开/关 锁反转的bug：

\*

\*/

locking\_selftest();

#ifdef CONFIG\_BLK\_DEV\_INITRD

if (initrd\_start && !initrd\_below\_start\_ok &&

page\_to\_pfn(virt\_to\_page((void \*)initrd\_start)) < min\_low\_pfn) {

printk(KERN\_CRIT "initrd overwritten (0x%08lx < 0x%08lx) - "

"disabling it.\n",

page\_to\_pfn(virt\_to\_page((void \*)initrd\_start)),

min\_low\_pfn);

initrd\_start = 0;

}

#endif

locking\_selftest();

#ifdef CONFIG\_BLK\_DEV\_INITRD

if (initrd\_start && !initrd\_below\_start\_ok &&

page\_to\_pfn(virt\_to\_page((void \*)initrd\_start)) < min\_low\_pfn) {

printk(KERN\_CRIT "initrd overwritten (0x%08lx < 0x%08lx) - "

"disabling it.\n",

page\_to\_pfn(virt\_to\_page((void \*)initrd\_start)),

min\_low\_pfn);

initrd\_start = 0;

}

#endif

检查initrd的位置是否符合要求。

min\_low\_pfn是系统可用的最小的pfn（页帧号）。

也就是判断传递进来initrd\_start对应的物理地址是否正常。如果有误就打印错误信息，并清零initrd\_start。

page\_cgroup\_init();

mem\_cgroup是cgroup体系中提供的用于memory隔离的功能，此处对此功能进行初始化。

参考资料：《linux内核mem\_cgroup浅析》

enable\_debug\_pagealloc();

使能页分配的调试标志。

**设置内存分配是否需要输出调试信息，如果调用这个函数，当分配内存时，不会输出一些相关的信息。**

debug\_objects\_mem\_init();

debug objects机制的内存分配初始化

kmemleak\_init();

内核内存泄漏检测机制初始化；

**创建调试对象内存缓存，所以紧跟内存缓存初始化后面**

setup\_per\_cpu\_pageset();

设置每个CPU的页组，并初始化。此前只有启动页组。

**创建每个CPU的高速缓存集合数组。因为每个CPU都不定时需要使用一些页面内存和释放页面内存，为了提高效率，就预先创建一些内存页面作为每个CPU的页面集合。**

numa\_policy\_init();

非一致性内存访问（NUMA）初始化

if (late\_time\_init) late\_time\_init();

如果构架存在此函数，就调用后期时间初始化。

**主要运行时钟相关后期的初始化功能。**

sched\_clock\_init();

对每个CPU，初始化调度时钟

calibrate\_delay();

计算BogoMIPS值，他是衡量一个CPU性能的标志。

**主要计算CPU需要校准的时间，这里说的时间是CPU执行时间。如果是引导CPU，这个函数计算出来的校准时间是不需要使用的，主要使用在非引导CPU上，因为非引导CPU执行的频率不一样，导致时间计算不准确。**

pidmap\_init();

PID分配映射初始化。

**进程位图初始化，一般情况下使用一页来表示所有进程占用情况。**

anon\_vma\_init();

匿名虚拟内存域（ anonymous VMA）初始化

#ifdef CONFIG\_X86

if (efi\_enabled) efi\_enter\_virtual\_mode();

#endif

thread\_info\_cache\_init();

获取thread\_info缓存空间，大部分构架为空函数（包括ARM）

线程信息的缓存初始化。

cred\_init();

任务信用系统初始化。详见：Documentation/credentials.txt

fork\_init(totalram\_pages);

进程创建机制初始化。为内核"task\_struct"分配空间，计算最大任务数。

**根据当前物理内存计算出来可以创建进程（线程）的数量，并进行进程环境初始化。**

proc\_caches\_init();

**初始化进程创建机制所需的其他数据结构，为其申请空间。**

**进程缓存初始化。**

buffer\_init();

缓存系统初始化，创建缓存头空间，并检查其大小限时。

初始化文件系统的缓冲区，并计算最大可以使用的文件缓存。

key\_init();

**内核密匙管理系统初始化。**

**初始化安全键管理列表和结构。**

security\_init();

内核安全框架初始化

**初始化安全管理框架，以便提供访问文件／登录等权限。**

dbg\_late\_init();

内核调试系统后期初始化

vfs\_caches\_init(totalram\_pages);

虚拟文件系统（VFS）缓存初始化

**虚拟文件系统进行缓存初始化，提高虚拟文件系统的访问速度。**

signals\_init();

信号管理系统初始化; 初始化信号队列缓存。

page\_writeback\_init();

根文件系统的填充可能需要也回写机制

页回写机制初始化

#ifdef CONFIG\_PROC\_FS

proc\_root\_init();

#endif

proc文件系统初始化

cgroup\_init();

control group的正式初始化

cpuset\_init();

CPUSET初始化。

参考资料：《多核心計算環境—NUMA與CPUSET簡介》

taskstats\_init\_early();

任务状态早期初始化函数：为结构体获取高速缓存，并初始化互斥机制。

**初始化任务状态相关的缓存、队列和信号量。任务状态主要向用户提供任务的状态信息。**

delayacct\_init();

**任务延迟机制初始化**

**初始化每个任务延时计数。当一个任务等CPU运行，或者等IO同步时，都需要计算等待时间。**

check\_bugs();

**检查CPU BUG的函数，通过软件规避BUG**

**用来检查CPU配置、FPU等是否非法使用不具备的功能**

acpi\_early\_init();

ACPI早期初始化函数。

ACPI - Advanced Configuration and Power Interface高级配置及电源接口

sfi\_init\_late();

SFI 初始程序晚期设置函数，

SFI - SIMPLE FIRMWARE INTERFACE。

ftrace\_init();

功能跟踪调试机制初始化，ftrace 是 function trace 的简称。

rest\_init();

}

### rest\_init();

所剩下的非-\_\_init的初始化, 内核现在已经启动了

虽然从名字上来说是剩余的初始化。但是这个函数中的初始化包含了很多的内容，后面我回单独写一篇来分析。

内核启动参数的获取和处理

setup\_arch(&command\_line);函数

内存管理的初始化（从bootmem到slab）

rest\_init();函数

其他的部分都是对内核各个组件的数据结构申请内存，并初始化。

《ARM Linux系统启动-start\_kernel函数》

一个很棒的台湾博士生 陳育書 的关于Linux启动分析的博文

《深入Linux内核构架》附录D 系统启动

前面粗略分析start\_kernel函数，此函数中基本上是对内存管理和各子系统的数据结构初始化。在内核初始化函数start\_kernel执行到最后，就是调用rest\_init函数，这个函数的主要使命就是创建并启动内核线程init。这个函数虽然意思为剩下的初始化，但是这个“剩下”的可是内容颇多，下面详细分析如下：

/\*

\* 我们必须确定在一个非\_\_init函数或

\* 其他根线程（root thread）和初始化线程（init thread）间的竞态。

\* （这种竞态可能导致start\_kernel在根线程运作到cpu\_idle前被free\_initmem“收割”。）

\*

\*

\* gcc-3.4 偶尔会将这个函数作为内联函数, 所以使用了noinline.

\*/

static \_\_initdata DECLARE\_COMPLETION(kthreadd\_done);

定义一个complete变量来告诉init线程：kthreads线程已经创建完成。

从前似乎不是用complete锁，而是用大内核锁。

static noinline void \_\_init\_refok rest\_init(void)

{

int pid;

#### rcu\_scheduler\_starting();

内核RCU锁机制调度启动,因为下面就要用到

/\*

\* 我们必须先创建init内核线程，这样它就可以获得pid为1。

\* 尽管如此init线程将会挂起来等待创建kthreads线程。

\* 如果我们在创建kthreadd线程前调度它，就将会出现OOPS。

\*/

#### kernel\_init

kernel\_thread(kernel\_init, NULL, CLONE\_FS | CLONE\_SIGHAND);

创建kernel\_init内核线程，内核的1号进程！！！！！

#### numa\_default\_policy();

设定NUMA系统的内存访问策略为默认

#### kthreadd

pid = kernel\_thread(kthreadd, NULL, CLONE\_FS | CLONE\_FILES);

创建kthreadd内核线程，它的作用是管理和调度其它内核线程。

它循环运行一个叫做kthreadd的函数，该函数的作用是运行kthread\_create\_list全局链表中维护的内核线程。

调用kthread\_create创建一个kthread，它会被加入到kthread\_create\_list 链表中；

被执行过的kthread会从kthread\_create\_list链表中删除；

且kthreadd会不断调用scheduler函数让出CPU。此线程不可关闭。

上面两个线程就是我们平时在Linux系统中用ps命令看到：

tekkaman@Super-MAGI:~/development/analyze/linux-3.0$ ps -A

PID TTY TIME CMD

1 ? 00:00:00 init

2 ? 00:00:00 kthreadd

......

rcu\_read\_lock();

kthreadd\_task = find\_task\_by\_pid\_ns(pid, &init\_pid\_ns);

rcu\_read\_unlock();

complete(&kthreadd\_done);

获取kthreadd的线程信息，获取完成说明kthreadd已经创建成功。并通过一个complete变量（kthreadd\_done）来通知kernel\_init线程。

/\*

\* 为让系统运作起来，

\* boot idle线程必须至少执行一次schedule():

\*/

#### init\_idle\_bootup\_task(current);

设置当前进程为idle（闲置）进程类。

#### preempt\_enable\_no\_resched();

使能抢占，但不重新调度

#### schedule();

执行调度，切换进程。

#### preempt\_disable();

进程调度完成，回到这里，禁用抢占。

#### cpu\_idle();

此时内核本体进入了idle状态，用循环消耗空闲的CPU时间片，该函数从不返回。在有其他进程需要工作的时候，该函数就会被抢占！这个函数因构架不同而异。

在以上的函数中，内核创建了两个内核线程，一个是内核线程的管理者，另一个是内核初始化线程init，后者是我们分析内核启动需要关注的，这个线程继续做系统的初始化（其中就包含了设备驱动系统）：

下面这个函数就是内核init线程运行的函数，它将完成设备驱动程序的初始化，并调用init\_post函数启动用户空间的init进程。

### init线程

static int \_\_init kernel\_init(void \* unused)

{

/\*

\* 等待kthreadd的启动完成.

\*/

wait\_for\_completion(&kthreadd\_done);

/\*

\* init可以在任何节点（node）分配到内存页

\*/

set\_mems\_allowed(node\_states[N\_HIGH\_MEMORY]);

/\*

\* init可以在任何CPU上运行.

\*/

set\_cpus\_allowed\_ptr(current, cpu\_all\_mask);

增加当前进程的CPU亲和力，使所有的CPU（如果是SMP）都可以运行本线程。

线程可以被迁移到被设置掩码的CPU上运行，但如果在位掩码中删除该CPU位，此线程就不会在那个CPU上运行。

cad\_pid = task\_pid(current);

cad\_pid为接收Ctrl-alt-del操作的INT信号的进程ID，此处很明显是设为了init的PID

smp\_prepare\_cpus(setup\_max\_cpus);

do\_pre\_smp\_initcalls();

lockup\_detector\_init();

smp\_init();

sched\_init\_smp();

以上代码是在SMP系统做准备，激活所有CPU，并开始SMP系统的调度。

#### do\_basic\_setup();

到此，与构架相关的部分已经初始化完成了，do\_basic\_setup函数主要是初始化设备驱动，完成其他驱动程序（直接编译进内核的模块）的初始化。内核中大部分的启动数据输出（都是各设备的驱动模块输出）都是这里产生的。

此函数比较重要，以后会详细分析！

/\* 打开根文件系统中的 /dev/console , 此处不可失败 \*/

if (sys\_open((const char \_\_user \*) "/dev/console", O\_RDWR, 0) < 0)

printk(KERN\_WARNING "Warning: unable to open an initial console.\n");

这是kernel\_init（以后的init进程）打开的第一个文件，它也就成为了标准输入。

这里需要打开 /dev/console，如果没有这个节点，系统就出错。这个错误信息也是经常碰到的。可能的原因是：

1、制作文件系统的时候忘记创建/dev/console节点

2、文件系统挂载问题，挂载上的文件系统不是什么都没有，就是挂错了节点。

(void) sys\_dup(0);

(void) sys\_dup(0);

复制两次标准输入（0）的文件描述符（它是上面打开的/dev/console，也就是系统控制台）：

一个作为标准输出（1）

一个作为标准出错（2）

现在标准输入、标准输出、标准出错都是/dev/console了。

这个console在内核启动参数中可以配置为某个串口（ttySn、ttyOn等等），也可以是虚拟控制台（tty0）。所以我们就在串口或者显示器上看到了之后的系统登录提示。

/\*

\* 检查是否有早期用户空间的init程序。如果有，让其执行

\*

\*/

if (!ramdisk\_execute\_command)

ramdisk\_execute\_command = "/init";

if (sys\_access((const char \_\_user \*) ramdisk\_execute\_command, 0) != 0) {

ramdisk\_execute\_command = NULL;

prepare\_namespace();

}

/\*

\* Ok, 我们已经完成了启动初始化, and

\* 且我们本质上已经在运行。释放初始化用的内存（initmem）段

\* 并开始用户空间的程序..

\*/

init\_post();

return 0;

}

在内核init线程的最后执行了init\_post函数，在这个函数中真正启动了用户空间进程init，详解如下：

### init\_post

/\* 这是一个非\_\_init函数。强制让它为非内联函数，以防 gcc

\* 让它内联到init()中并成为init.text段的一部分。

\*/

从此函数名可知，这个函数是运行在用户空间的init程序之前

static noinline int init\_post(void)

{

/\* 在释放内存前，必须完成所有的异步 \_\_init 代码 \*/

async\_synchronize\_full();

free\_initmem();

释放所有init.\* 段中的内存。

mark\_rodata\_ro();

通过修改页表，保证只读数据段为只读属性。大部分构架为空函数。

system\_state = SYSTEM\_RUNNING;

设置系统状态为运行状态

 numa\_default\_policy();

设定NUMA系统的内存访问策略为默认

current->signal->flags |= SIGNAL\_UNKILLABLE;

设置当前进程（init）为不可以杀进程（忽略致命的信号）

if (ramdisk\_execute\_command) {

run\_init\_process(ramdisk\_execute\_command);

printk(KERN\_WARNING "Failed to execute %s\n",

ramdisk\_execute\_command);

}

如果ramdisk\_execute\_command有指定的init程序，就执行它。

/\*

\* 我们尝试以下的每个函数，直到函数成功执行.

\*

\* 如果我们试图修复一个真正有问题的设备，

\* Bourne shell 可以替代init进程。

\*/

if (execute\_command) {

run\_init\_process(execute\_command);

printk(KERN\_WARNING "Failed to execute %s. Attempting "

"defaults...\n", execute\_command);

}

如果execute\_command有指定的init程序，就执行它。

run\_init\_process("/sbin/init");

run\_init\_process("/etc/init");

run\_init\_process("/bin/init");

run\_init\_process("/bin/sh");

panic("No init found. Try passing init= option to kernel. "

"See Linux Documentation/init.txt for guidance.");

在检查完ramdisk\_execute\_command和execute\_command为空的情况下，顺序执行以下初始化程序：如果都没有找到就打印错误信息。这也是我们做系统移植的时候经常碰到的错误信息，出现这个信息很有可能是：

1、你的启动参数配置有问题，通过 指定了init程序，但是没有找到，且默认的那四个程序也不在文件系统中。

2、文件系统挂载有问题，文件不存在

3、init程序没有执行权限

**至此，内核的初始化结束，正式进入了用户空间的初始化过程！！**

}

### do\_basic\_setup

在基本分析完内核启动流程的之后，还有一个比较重要的初始化函数没有分析，那就是do\_basic\_setup。在内核init线程中调用了do\_basic\_setup，这个函数也做了很多内核和驱动的初始化工作，详解如下：

/\*

\* 好了, 设备现在已经初始化完成。 但是还没有一个设备被初始化过，

\* 但是 CPU 的子系统已经启动并运行，

\* 且内存和处理器管理系统已经在工作了。

\*

\* 现在我们终于可以开始做一些实际的工作了..

\*/

static void \_\_init do\_basic\_setup(void)

{

#### cpuset\_init\_smp();

1. 针对SMP系统，初始化内核control group的cpuset子系统。如果非SMP，此函数为空。
2. **cpuset是在用户空间中操作cgroup文件系统来执行进程与cpu和进程与内存结点之间的绑定。**
3. 本函数将cpus\_allowed和mems\_allwed更新为在线的cpu和在线的内存结点，并为内存热插拨注册了钩子函数，最后创建一个单线程工作队列cpuset。

#### usermodehelper\_init();

创建一个单线程工作队列khelper。运行的系统中只有一个，主要作用是指定用户空间的程序路径和环境变量, 最终运行指定的user space的程序，属于关键线程，不能关闭。

#### init\_tmpfs();

初始化内核tmpfs文件系统

#### driver\_init();

初始化驱动模型中的各子系统，可见的现象是在/sys中出现的目录和文件

#### init\_irq\_proc();

在proc文件系统中创建irq目录，并在其中初始化系统中所有中断对应的目录。

#### do\_ctors();

调用链接到内核中的所有构造函数，也就是链接进.ctors段中的所有函数。

在Linux-2.6.31开始内核启动增加了对构造函数的支持。

git提交：

git提交：

点击(此处)折叠或打开

commit b99b87f70c7785ab1e253c6220f4b0b57ce3a7f7

Author: Peter Oberparleiter

Date: Wed Jun 17 16:28:03 2009 -0700

kernel: constructor support

内核：构造函数支持

Call constructors (gcc-generated initcall-like functions) during kernel

start and module load. Constructors are e.g. used for gcov data

initialization.

在内核启动和模块挂载时，调用构造函数（gcc生成的类初始化函数）。构造函数就是

比如用于初始化gcov数据的函数

Disable constructor support for usermode Linux to prevent conflicts with

host glibc.

对于Linux的用户模式禁用构造函数支持，以避免和glibc冲突。

Signed-off-by: Peter Oberparleiter

Acked-by: Rusty Russell

Acked-by: WANG Cong

Cc: Sam Ravnborg

Cc: Jeff Dike

Cc: Andi Kleen

Cc: Huang Ying

Cc: Li Wei

Cc: Michael Ellerman

Cc: Ingo Molnar

Cc: Heiko Carstens

Cc: Martin Schwidefsky

Cc: Al Viro

Signed-off-by: Andrew Morton

Signed-off-by: Linus Torvalds

$ git tag --contains b99b87f7

v2.6.31

v2.6.31-rc1

v2.6.31-rc2

v2.6.31-rc3

v2.6.31-rc4

v2.6.31-rc5

v2.6.31-rc6

v2.6.31-rc7

v2.6.31-rc8

v2.6.31-rc9

v2.6.32

v2.6.32-rc1

v2.6.32-rc2

v2.6.32-rc3

v2.6.32-rc4

v2.6.32-rc5

v2.6.32-rc6

v2.6.32-rc7

v2.6.32-rc8

#### do\_initcalls();

调用所有编译内核的驱动模块中的初始化函数。

这里就是驱动程序员需要关心的步骤，其中按照各个内核模块初始化函数所自定义的启动级别（1～7），按顺序调用器初始化函数。

对于同一级别的初始化函数，安装编译是链接的顺序调用，也就是和内核Makefile的编写有关。

在编写内核模块的时候需要知道这方面的知识，比如你编写的模块使用的是I2C的API，那你的模块的初始化函数的级别必须低于I2C子系统初始化函数的级别（也就是级别数（1~7）要大于I2C子系统）。如果编写的模块必须和依赖的模块在同一级，那就必须注意内核Makefile的修改了。

这方面的知识会在有空的时候总结下，网上也有相关的文章。

### driver\_init

上面的函数调用了driver\_init函数，作用是驱动模型子系统的初始化，对于内核驱动工程师来说比较重要，详解如下：

drivers/base/init.c：

/\*\*

\* driver\_init - 初始化驱动模型.

\*

\* 调用驱动模型初始化函数来初始化它们的子系统。

\* 由早期的init/main.c中调用。

\*/

void \_\_init driver\_init(void)

{

/\* 它们为核心部件 \*/

devtmpfs\_init();

初始化devtmpfs文件系统，驱动核心设备将在这个文件系统中添加它们的设备节点。

这个文件系统可以由内核在挂载根文件系统之后自动挂载到/dev下，也可以在文件系统的启动脚本中手动挂载。

devices\_init();

初始化驱动模型中的部分子系统和kobject：

devices

dev

dev/block

dev/char

buses\_init();

初始化驱动模型中的bus子系统

classes\_init();

初始化驱动模型中的class子系统

firmware\_init();

初始化驱动模型中的firmware子系统

hypervisor\_init();

初始化驱动模型中的hypervisor子系统

/\* 这些也是核心部件, 但是必须

\* 在以上核心中的核心部件之后调用。

\*/

platform\_bus\_init();

初始化驱动模型中的bus/platform子系统

system\_bus\_init();

初始化驱动模型中的devices/system子系统

cpu\_dev\_init();

初始化驱动模型中的devices/system/cpu子系统

memory\_dev\_init();

初始化驱动模型中的devices/system/memory子系统

虽然从代码上看这样，但是我在实际的系统中并没有找到/sys/devices/system/memory这个目录。

### setup\_arch

在分析start\_kernel函数的时候，其中有构架相关的初始化函数setup\_arch。

此函数根据构架而异，对于ARM构架的详细分析如下：

void \_\_init setup\_arch(char \*\*cmdline\_p)

{

struct machine\_desc \*mdesc;

此为设备描述结构体，对于任何板子都定义了这样的一个结构体，我以前的文章有介绍：

《Uncompressing Linux... done, booting the kernel》 1、machine type 不匹配

unwind\_init();

初始化基於ARM EABI的Backtrace Unwind機制(栈回退)，此函数主要用于地址转换（arch/arm/kernel/unwind.c）

setup\_processor();

再次检测处理器类型，并初始化处理器相关的底层变量。内核启动时的处理器信息（包括cache）就是通过这个函数打印的，例如：

CPU: ARMv7 Processor [413fc082] revision 2 (ARMv7), cr=10c53c7f

CPU: VIPT nonaliasing data cache, VIPT aliasing instruction cache

mdesc = setup\_machine\_fdt(\_\_atags\_pointer);

if (!mdesc)

mdesc = setup\_machine\_tags(machine\_arch\_type);

在此处通过bootloader传递过来的设备ID来匹配一个 struct machine\_desc 结构体

（这个结构体就是在arch/arm/mach-\*/mach-\*.c中定义的结构体：MACHINE\_START～MACHINE\_END ）

如果没有匹配上就死循环。

如果匹配上了就打印机器名 ，并处理bootloader传递过来的tagged\_list，将所有的tag信息保存到相应的全局变量或结构体中。

内核启动时的机器信息就是这里打印的，例如：

Machine: ti8168evm

最后返回结构体指针。

machine\_desc = mdesc;

machine\_name = mdesc->name;

通过匹配的struct machine\_desc 结构体数据，初始化一些全局变量

if (mdesc->soft\_reboot)

reboot\_setup("s");

通过struct machine\_desc 中的soft\_reboot数据来设置重启类型：

如果存在就为“s”：softreset；如果不存在就为“h”：hardreset

init\_mm.start\_code = (unsigned long) \_text;

init\_mm.end\_code = (unsigned long) \_etext;

init\_mm.end\_data = (unsigned long) \_edata;

init\_mm.brk = (unsigned long) \_end;

这里通过连接脚本中得到的Linux代码位置数据来初始化一个mm\_struct结构体init\_mm中的部分数据

ps：每一个任务都有一个mm\_struct结构以管理内存空间，init\_mm是内核自身的mm\_struct

/\* 同时填充cmd\_line以备后用, 保护boot\_command\_line数据 \*/

strlcpy(cmd\_line, boot\_command\_line, COMMAND\_LINE\_SIZE);

\*cmdline\_p = cmd\_line;

将boot\_command\_line复制到cmd\_line中。这里关键是要知道系统启动的时候的cmdline是如何传递的。

parse\_early\_param();

处理在 struct obs\_kernel\_param 中定义为early的启动参数（主要是内存配置部分的参数）

其中就分析了mem=size@start参数初始化了struct meminfo meminfo;

同时如果有vmalloc=size参数也会初始化 vmalloc\_min

参考：《Linux内核高-低端内存设置代码跟踪（ARM构架）》

这里需要注意的是内核的cmdline中的参数按照其被需要的先后，分为early和非early的。

include/linux/init.h：

struct obs\_kernel\_param {

const char \*str; //在cmdline中相应参数名。

int (\*setup\_func)(char \*); //对于此参数的专用处理函数

int early; //是否为早期需要处理的参数

};

两种不同的参数在内核中用了不同的宏来定义：

early： #define early\_param(str, fn) \

\_\_setup\_param(str, fn, fn, 1)

非early： #define \_\_setup(str, fn) \

\_\_setup\_param(str, fn, fn, 0)

使用这两个宏定义的参数和构架相关，一些构架或者板子可以定义自己特定的参数和处理函数。对于比较重要的“men”参数就是early参数。

sanity\_check\_meminfo();

在此处设置struct meminfo meminfo中每个bank中的highmem变量，

通过vmalloc\_min确定每个bank中的内存是否属于高端内存

arm\_memblock\_init(&meminfo, mdesc);

在此处按地址数据从小到大排序meminfo中的数据，并初始化全局的memblock数据。

paging\_init(mdesc);

设置内核的参考页表。

此页表不仅用于物理内存映射，还用于管理vmalloc区。

此函数中非常重要的一点就是初始化了bootmem分配器！

request\_standard\_resources(mdesc);

通过获取设备描述结构体（struct machine\_desc）中的数据和编译时产生的地址数据，初始化内存相关的全局结构体变量。

unflatten\_device\_tree();

通过启动参数中的“非平坦设备树”信息(如果有),获取内存相关信息

#ifdef CONFIG\_SMP

if (is\_smp())

smp\_init\_cpus();

#endif

针对SMP处理器，初始化可能存在的CPU映射 - 这描述了可能存在的CPU

reserve\_crashkernel();

用于内核崩溃时的保留内核

此功能通过内核command line参数中的"crashkernel="保留下内存用于主内核崩溃时获取内核信息的导出。

cpu\_init();

初始化一个CPU，并设置一个per-CPU栈

tcm\_init();

初始化ARM内部的TCM（紧耦合内存）。

参考资料：《对ARM紧致内存TCM的理解》

ARM官网也有介绍文档

#ifdef CONFIG\_MULTI\_IRQ\_HANDLER

handle\_arch\_irq = mdesc->handle\_irq;

#endif

调用设备描述结构体中的mdesc->handle\_irq函数，目的未知。

#ifdef CONFIG\_VT

#if defined(CONFIG\_VGA\_CONSOLE)

conswitchp = &vga\_con;

#elif defined(CONFIG\_DUMMY\_CONSOLE)

conswitchp = &dummy\_con;

#endif

#endif

early\_trap\_init();

对中断向量表进行早期初始化

if (mdesc->init\_early)

mdesc->init\_early();

如果设备描述结构体定义了init\_early函数（应该是早期初始化之意），则在这里调用。

}

这个函数主要是检查处理器的类型是否匹配，并获取处理器信息来设置处理器的相关底层参数。

static void \_\_init setup\_processor(void)

{

struct proc\_info\_list \*list;

/\*

\* 在支持处理器列表中定位处理器

\* 连接器为我们创建这个列表，从 \* arch/arm/mm/proc-\*.S中的入口

\*/

list = lookup\_processor\_type(read\_cpuid\_id());

if (!list) {

printk("CPU configuration botched (ID %08x), unable "

"to continue.\n", read\_cpuid\_id());

while (1);

}

这里再次核对处理器类型，虽然这个已经在汇编代码中执行过一遍了

cpu\_name = list->cpu\_name;

#ifdef MULTI\_CPU

processor = \*list->proc;

#endif

#ifdef MULTI\_TLB

cpu\_tlb = \*list->tlb;

#endif

#ifdef MULTI\_USER

cpu\_user = \*list->user;

#endif

#ifdef MULTI\_CACHE

cpu\_cache = \*list->cache;

#endif

通过从struct proc\_info\_list获取的数据初始化CPU相关的全局变量

printk("CPU: %s [%08x] revision %d (ARMv%s), cr=%08lx\n",

cpu\_name, read\_cpuid\_id(), read\_cpuid\_id() & 15,

proc\_arch[cpu\_architecture()], cr\_alignment);

打印内核启动时的处理器信息

sprintf(init\_utsname()->machine, "%s%c", list->arch\_name, ENDIANNESS);

sprintf(elf\_platform, "%s%c", list->elf\_name, ENDIANNESS);

elf\_hwcap = list->elf\_hwcap;

#ifndef CONFIG\_ARM\_THUMB

elf\_hwcap &= ~HWCAP\_THUMB;

#endif

feat\_v6\_fixup();

针对特定的ARM核软件屏蔽一些功能

cacheid\_init();

初始化ARM核中的缓存

cpu\_proc\_init();

宏：

#define cpu\_proc\_init \_\_glue(CPU\_NAME,\_proc\_init)

意在调用处理器特定的初始化函数。

}