# ARM 汇编

## ARM体系

**ARM处理器工作模式：**

用户模式(user)：正常执行状态

快速中断模式(fiq)：高速数据传输等

外部中断模式(irq)：通用的中断处理

管理模式(svc)：操作系统使用的保护模式

中止模式(abt)：数据或指令预取终止时进入的模式

未定义指令模式(und)：未定义指令执行时进入的模式

系统模式(sys)：运行具有特权的操作系统任务

**ARM处理器工作状态：**

ARM状态：执行32位ARM指令集

Thumb状态：执行16位Thumb指令集

**ARM寄存器**：

31个通用寄存器：

未分组寄存器：R0~R7

分组寄存器：R8~R14

R13常用作堆栈指针SP

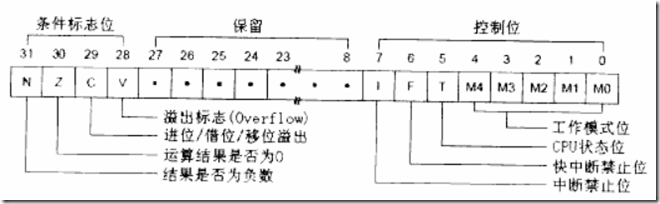
R14程序连接寄存器LR

6个状态寄存器：

R15用作程序计数器PC

R16程序状态寄存器CPSR

1：ARM每种工作模式除R0~R15共16个寄存器外，还有第17个寄存器CPSR，叫做 “当前程序状态寄存器”，CPSR中一些位被用于标识各种状态，一些位被用于标识当前处于什么工作模式。

[](http://images2015.cnblogs.com/blog/1092407/201701/1092407-20170125181623987-870666130.png)

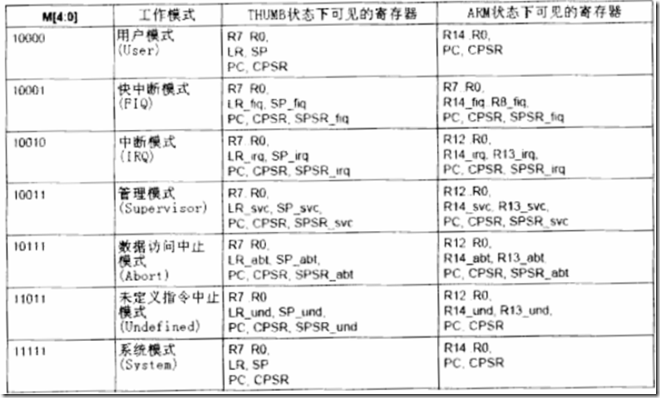
2：这个状态寄存器的32位可以分为四个8位的域。

[](http://images2015.cnblogs.com/blog/1092407/201701/1092407-20170125181628659-785937281.png)

其中cpsr\_c代表的是cpsr寄存器的低8位，也就是控制位

有些程序里这样写  
msr cpsr\_c, #0xd2       @ 进入中断模式

这样的语句时，就是在更改这8位的值为0xd2，根据下图，知道是进入中断模式

[](http://images2015.cnblogs.com/blog/1092407/201701/1092407-20170125181630362-2026780902.png)

**ARM存储：**

arm从零地址开始存储，

数据类型：

字节(8位)，半字(16位)，字(32位)3种数据类型，字需要4字节

对齐(地址的低2位为0即是4的倍数)，半字需要2字节对齐(地址的

最低位为0即是2的倍数)

存储格式：

大端格式，字数据的高字节在低地址，低字节在高地址

小端格式，字数据的低字节在低地址，高字节在高地址

默认为小端格式

ARM非对齐的存储访问：

非对齐值ARM状态低2位不为0，Thumb状态最低位不为0

非对齐的指令预取操作：

忽略地址中相应的位（ARM状态下是低2位，Thumb状态下最低

位）

非对齐的数据访问操作：

执行结果不可知

忽略相应的地址位

**ATPCS**

寄存器使用规则：



**ARM协处理器：**

ARM 微处理器可支持多达 16 个协处理器，用于各种协处理操作，在程序执行的过程中，每个协处理器只执行针对自身的协处理指令，忽略 ARM 处理器和其他协处理器的指令。ARM 的协处理器指令主要用于 ARM 处理器初始化 ARM 协处理器的数据处理操作，以及在ARM 处理器的寄存器和协处理器的寄存器之间传送数据，和在 ARM 协处理器的寄存器和存储器之间传送数据。 ARM 协处理器指令包括以下 5 条：

— CDP 协处理器数操作指令

— LDC 协处理器数据加载指令

— STC 协处理器数据存储指令

— MCR ARM 处理器寄存器到协处理器寄存器的数据传送指令

— MRC 协处理器寄存器到ARM 处理器寄存器的数据传送指令

1、CDP 指令

CDP 指令的格式为：

CDP{条件} 协处理器编码，协处理器操作码1，目的寄存器，源寄存器1，源寄存器2，协处理 器操作码2。 CDP 指令用于ARM 处理器通知ARM 协处理器执行特定的操作,若协处理器不能成功完成特定的操作，则产生未定义指令异常。其中协处理器操作码1 和协处理器操作码2 为协处理器将要执行的操作，目的寄存器和源寄存器均为协处理器的寄存器，指令不涉及ARM 处理器的寄存器和存储器。

指令示例：

CDP P3 ， 2 ， C12 ， C10 ， C3 ， 4 ；该指令完成协处理器 P3 的初始化

2、LDC 指令

LDC 指令的格式为：

LDC{条件}{L} 协处理器编码,目的寄存器，[源寄存器]

LDC 指令用于将源寄存器所指向的存储器中的字数据传送到目的寄存器中，若协处理器不能成功完成传送操作，则产生未定义指令异常。其中，{L}选项表示指令为长读取操作，如用于双精度数据的传输。

指令示例：

LDC P3 ， C4 ， [R0] ；将 ARM 处理器的寄存器 R0 所指向的存储器中的字数据传送到协处理器 P3 的寄存器 C4 中。

3、STC 指令

STC 指令的格式为：

STC{条件}{L} 协处理器编码,源寄存器，[目的寄存器]

STC 指令用于将源寄存器中的字数据传送到目的寄存器所指向的存储器中，若协处理器不能成功完成传送操作，则产生未定义指令异常。其中，{L}选项表示指令为长读取操作，如用于双精度数据的传输。

指令示例：

STC P3 ， C4 ， [R0] ；将协处理器 P3 的寄存器 C4 中的字数据传送到 ARM 处理器的寄存器R0 所指向的存储器中。

4、MCR 指令

MCR 指令的格式为：

MCR{条件} 协处理器编码，协处理器操作码1，源寄存器，目的寄存器1，目的寄存器2，协处理器操作码2。

MCR 指令用于将ARM 处理器寄存器中的数据传送到协处理器寄存器中,若协处理器不能成功完成操作，则产生未定义指令异常。其中协处理器操作码1 和协处理器操作码2 为协处理器将要执行的操作，源寄存器为ARM 处理器的寄存器，目的寄存器1 和目的寄存器2 均为协处理器的寄存器。

指令示例：

MCR P3 ， 3 ， R0 ， C4 ， C5 ， 6 ；该指令将 ARM 处理器寄存器 R0 中的数据传送到协处理器 P3 的寄存器 C4 和 C5 中。

5、MRC 指令

MRC 指令的格式为：

MRC{条件} 协处理器编码，协处理器操作码1，目的寄存器，源寄存器1，源寄存器2，协处理器操作码2。

MRC 指令用于将协处理器寄存器中的数据传送到ARM 处理器寄存器中,若协处理器不能成功完成操作，则产生未定义指令异常。其中协处理器操作码1 和协处理器操作码2 为协处理器将要执行的操作，目的寄存器为ARM 处理器的寄存器，源寄存器1 和源寄存器2 均为协处理器的寄存器。

指令示例：

MRC P3 ， 3 ， R0 ， C4 ， C5 ， 6 ；该指令将协处理器 P3 的寄存器中的数据传送到 ARM 处理器寄存器中.

The ARM920T 有两个具体协处理器

CP14（略）

CP15 —系统控制协处理器 （the system control coprocessor）他通过协处理器指令MCR和MRC提供具体的寄存器来配置和控制caches、MMU、保护系统、配置时钟模式（在bootloader时钟初始化用到）……

CP15的寄存器只能被MRC和MCR（Move to Coprocessor from ARM Register ）指令访问

他包括15个具体的寄存器如下：

-R0：ID号寄存器 这是一个只读寄存器，返回一个32位的设备ID号，32具体功能参考2410 datesheet的 table 2-5

-R0：缓存类型寄存器（CACHE TYPE REGISTER），注意有2个R0，根据MCR操作数的不同传送不同的值，这也一个只读寄存器，包含了caches的信息。读这个寄存器的方式是通过设置协处理操作码为1.

如：

MRC p15,0,Rd,c0,c0,1; 返回caches的详细信息

详见：table 2-6

-R1：控制寄存器 2-10

-R2：转换表基址寄存器（Translation Table Base --TTB） 2-12

-R3：域访问控制寄存器（Domain access control ）2-13

-R4：保留 2-14

-R5：异常状态寄存器（fault status -FSR）2-14

-R6：异常地址寄存器（fault address -FAR）2-15

-R7：缓存操作寄存器 2-15

-R8：TLB操作寄存器 2-18

-R9：缓存锁定寄存器 2-19

-R10：TLB 锁定寄存器 2-21

-R11-12&14：保留

-R13：处理器ID 2-22

-R15：测试配置寄存器 2-24

## GNU汇编

GCC内联汇编的一般格式：

asm(

代码列表

: 输出运算符列表

: 输入运算符列表

: 被更改资源列表

);

   \_\_asm\_\_ \_\_violate\_\_ ("movl %1,%0" : "=r" (result) : "m" (input));     
  
    movl %1,%0  是指令模板；%0  和 %1 代表指令的操作数，称为占位符，内嵌汇编靠它们将C 语言表达式与指令操作数相对应。

指令模板后面用小括号括起来的是 C 语言表达式，本例中只有两个：result 和 input ，他们按照出现的顺序分别与指令操作数 %0 、%1 对应；注意对应顺序：第一个 C 表达式对应 %0 ；第二个表达式对应 %1 ，依次类推，操作数至多有10个，分别用 %0, %1 …. %9 表示

Linux 平台的标准汇编器是 GAS，它是 GCC 所依赖的后台汇编工具，通常包含在 binutils 软件包中。GAS 使用标准的 AT&T 汇编语法，可以用来汇编用 AT&T 格式编写的程序

@ 表示注释

在 AT&T 汇编格式中，寄存器名要加上 '%' 作为前缀

AT&T 格式 pushl %eax

Intel 格式 push eax

在 AT&T 汇编格式中，用 '$' 前缀表示一个立即操作数；而在 Intel 汇编格式中，立即数的表示不用带任何前缀

AT&T 格式 pushl $1

Intel 格式 push 1

AT&T 和 Intel 格式中的源操作数和目标操作数的位置正好相反。在 Intel 汇编格式中，目标操作数在源操作数的左边；而在 AT&T 汇编格式中，目标操作数在源操作数的右边

AT&T 格式 addl $1, %eax

Intel 格式 add eax, 1

在 AT&T 汇编格式中，操作数的字长由操作符的最后一个字母决定，后缀'b'、'w'、'l'分别表示操作数为字节（byte，8 比特）、字（word，16 比特）和长字（long，32比特）；而在 Intel 汇编格式中，操作数的字长是用 "byte ptr" 和 "word ptr" 等前缀来表示的。例如：

AT&T 格式 movb val, %al

Intel 格式 mov al, byte ptr val

在 AT&T 汇编格式中，绝对转移和调用指令（jump/call）的操作数前要加上'\*'作为前缀，而在 Intel 格式中则不需要。

远程转移指令和远程子调用指令的操作码，在 AT&T 汇编格式中为 "ljump" 和 "lcall"，而在 Intel 汇编格式中则为 "jmp far" 和 "call far"，即：

AT&T 格式 ljump $section, $offset lcall $section, $offset

Intel 格式 jmp far section:offset call far section:offset

与之相应的远程返回指令则为：

AT&T 格式 lret $stack\_adjust

Intel 格式 ret far stack\_adjust

在 AT&T 汇编格式中，内存操作数的寻址方式是

section:disp(base, index, scale)

而在 Intel 汇编格式中，内存操作数的寻址方式为：

section:[base + index\*scale + disp]

下面是一些内存操作数的例子：

AT&T 格式 Intel 格式

movl -4(%ebp), %eax mov eax, [ebp - 4]

movl array(, %eax, 4), %eax mov eax, [eax\*4 + array]

movw array(%ebx, %eax, 4), %cx mov cx, [ebx + 4\*eax + array]

movb $4, %fs:(%eax) mov fs:eax, 4

# SMP

## bootup

1、当系统加电或reset，ARM11 MPCore的所有CPUs取下一条指令（从 restet 向量地址，即NOR flash 的

0x00000000地址）到它们自己的PC寄存器，这个地址放了引导监视器的程序；

2、只有CPU0 继续执行引导监视器代码，并且其它CPUs执行WFI 指令，这是一个循环，检测SYS\_FLAGS

寄存器。其它CPUs在Linux 内核引导过程中，才开始执行有意义的代码

setup\_arch()

setup\_processor()

cpuinfo\_store\_boot\_cpu()

smp\_init\_cpus() //初始化CPU映射

cpu\_logical\_map()

cpu\_read\_ops()

setup\_per\_cpu\_areas() //分配和初始化各个核分别所需要的内存

//允许正在引导的处理器（CPU0）访问自己的初始化过的数据

smp\_prepare\_boot\_cpu()

sched\_init()

init\_idle()

rest\_init()

kernel\_init()

kernel\_init\_freeable()

smp\_prepare\_cpus()

smp\_init()

idle\_threads\_init()

sched\_init\_smp()

scheduler\_tick()

trigger\_load\_balance(rq)

raise\_softirq(SCHED\_SOFTIRQ)

run\_rebalance\_domains()

rebalance\_domains()

rebalance\_domains()

load\_balance()

find\_busiest\_group()

find\_busiest\_queue()

detach\_tasks()

attach\_tasks()

attach\_task()

smp\_call\_function\_single()

generic\_exec\_single()

arch\_send\_call\_function\_single\_ipi()

smp\_cross\_call()

\_\_smp\_cross\_call()

csd\_lock\_wait()

# 进程管理

## 进程创建与执行

long do\_fork(unsigned long clone\_flags, //一些标志

unsigned long stack\_start, //子进程用户态堆栈的开始地址

struct pt\_regs \*regs, //指向通用寄存器值的指针，

unsigned long stack\_size, //未使用

int \_\_user \*parent\_tidptr, //父进程在用户态下pid的地址

int \_\_user \*child\_tidptr) //子进程在用户态下pid的地址

copy\_process(regs)

dup\_task\_struct()

tsk = alloc\_task\_struct() //使用内核的slab分配器去为所要创建

的进程分配struct task\_struct的空间

ti = alloc\_thread\_info(tsk) //使用内核的伙伴系统去为所要创

建的进程分配内核栈

tsk->stack = ti //关联了struct task\_struct和内核栈

setup\_thread\_stack(tsk, orig)

prepare\_to\_copy()

copy\_flags()

copy\_thread(regs)

struct thread\_info \*thread = task\_thread\_info(p)

\*childregs = \*regs

thread->cpu\_context.sp = (unsigned long)childregs

// ret\_from\_fork为汇编语言首地址

thread->cpu\_context.pc = (unsigned long)ret\_from\_fork

sched\_fork()

wake\_up\_new\_task

int do\_execve(char \* filename,

char \_\_user \*\_\_user \*argv,

char \_\_user \*\_\_user \*envp,

struct pt\_regs \* regs)

//arch/arm/kernel/process.c 创建内核线程

pid\_t kernel\_thread(int (\*fn)(void \*), void \*arg, unsigned long flags)

struct pt\_regs regs

regs.ARM\_r1 = (unsigned long)arg //内核线程函数的参数

regs.ARM\_r2 = (unsigned long)fn //内核线程函数的起始地址

regs.ARM\_r3 = (unsigned long)do\_exit

do\_fork(&regs)

struct pt\_regs {

long uregs[18];

}

## 进程相关数据结构

### task\_struct

task\_struct{

void \*stack //进程内核栈，内核态的进程访问处于内核数据段

的栈

state //进程的当前动态

prio //进程动态优先级 普通进程100-139，是进程调度程序

在选择新进程时所要使用的数

static\_prio //进程静态优先级 普通进程100-139，

sleep\_avg //平均睡眠时间

/\*prio = max(100, min(static\_prio – bonus + 5, 139))

bonus 为0 ~10，分别对应sleep\_avg 0~100, 100 ~200……\*/

/\*交互式进程判断：

prio <= 3\*static\_prio/4 + 28

或者 bonus – 5 >= static\_prio/4 + 28\*/

rt\_priority //实时优先级 实时进程1-99

normal\_prio //值取决于静态优先级和调度策略

time\_slice //基本时间片

/\* time\_slice =(140 - static\_prio) \* 20 static\_prio < 120

(140 - static\_prio) \* 5 static\_prio >= 120\*/

policy //进程调度类型

/\*SCHED\_FIFO 先进先出的实时进程

SCHED\_RR 时间片轮转的实时进程

SCHED\_NORMAL 普通的分时进程\*/

mm\_struct \*mm //用户空间内存描述符，内核线程为NULL

mm\_struct \*active\_mm //

struct prio\_array \*array //指向包含进程运行队列的集合

pid\_t pid //进程标识符

pid\_t tgid //线程组领头进程的pid

struct task\_struct \*real\_parent

struct task\_struct \*parent //指向进程的当前父进程

struct list\_head children //子进程链表

struct list\_head sibling //兄弟进程的链表

struct thread\_struct thread //进程切换时保存硬件上下文

struct nsproxy \*nsproxy{ //命名空间集合

struct uts\_namespace \*uts\_ns //内核名称、版本等信息

struct ipc\_namespace \*ipc\_ns //与进程通信相关的信息

struct mnt\_namespace \*mnt\_ns //已装载的文件系统视图

struct pid\_namespace \*pid\_ns //进程ID的信息

}

struct files\_struct \*files //与打开文件有关的信息

struct fs\_struct \*fs //进程文件系统信息

}

### thread\_info

//Linux把thread\_info（线程描述符）和内核态的线程堆栈存放在一起，

这块区域通常是8192K（占两个页框），其实地址必须是8192的整数

倍

thread\_info{

unsigned long flags

int preempt\_count

mm\_segment\_t addr\_limit

struct task\_struct \*task //进程描述符的指针

struct exec\_domain \*exec\_domain

\_\_u32 cpu

\_\_u32 cpu\_domain

struct cpu\_context\_save cpu\_context{ //cpu上下文

\_\_u32 sp //堆栈

\_\_u32 pc

}

\_\_u32 syscall

\_\_u8 used\_cp[16]

unsigned long tp\_value

struct crunch\_state crunchstate

union fp\_state fpstate \_\_attribute\_\_((aligned(8)))

union vfp\_state vfpstate

struct restart\_block restart\_block

}

union thread\_union {

struct thread\_info thread\_info

unsigned long stack[THREAD\_SIZE/sizeof(long)]

}

### file\_struct

file\_struct{ //打开文件信息

struct file \* fd\_array[NR\_OPEN\_DEFAULT]

}

### fs\_struct

fs\_struct{ //文件系统信息

atomic\_t count

rwlock\_t lock

int umask

struct dentry \* root, \* pwd, \* altroot //根目录，当前目录

struct vfsmount \* rootmnt, \* pwdmnt, \* altrootmnt

}

### pid

pid{

atomic\_t count

int nr //pid的数值

struct hlist\_node pid\_chain //链接散列链表的下一个和前一个元素

struct hlist\_head tasks[PIDTYPE\_MAX];

struct rcu\_head rcu

}

## 运行队列

rq{

struct task\_struct \*curr //当前正在运行的进程

struct task\_struct \*idle //当前CPU上的swapper进程

struct prio\_array \*active //活动进程链表

struct prio\_array \*expired //过期进程链表

}

## 等待队列

wait\_queue\_head\_t{ 等待队列头

spinlock\_t lock

struct list\_head task\_list

}

wait\_queue\_t{

unsigned int flags

#define WQ\_FLAG\_EXCLUSIVE 0x01

void \*private

wait\_queue\_func\_t func

struct list\_head task\_list

}

## 进程调度

调度类型：

SCHED\_FIFO: 先进先出的实时调度

SCHED\_RR: 时间片轮转的实时调度

SCHED\_NORMAL: 普通的分时进程

普通进程的调度

静态优先级：100~139

动态优先级：100~139

静态优先级 = MAX\_RT\_PRIO(为99) +nice+ 20

基本时间片 = (140 – 静态优先级) \* 20 若静态优先级<120

(单位为ms) (140 – 静态优先级) \* 5 若静态优先级>= 120

动态优先级 = max(100, min(静态优先级 – bonus + 5, 139))

bonus为0~10的值，

实时优先级：0~99

### scheduler\_tick函数

scheduler\_tick()

sched\_clock() //获取当前TSC值并转化为纳秒

smp\_processor\_id() //获得当前CPU的ID。如果不是SMP，那么

就返回0

idle\_cpu() //

cpu\_rq() //返回给定处理器可执行队列的指针

update\_cpu\_clock() //

task\_running\_tick()

//递减进程时间片

if(!--p->time\_slice){ //如果时间片已用完

requeue\_task(p, rq->active);

set\_tsk\_need\_resched(p);

}

else{ //如果时间片没有用完

}

update\_load()

trigger\_load\_balance()

### try\_to\_wake\_up函数

try\_to\_wake\_up()

task\_rq\_lock() //禁用本地中断

activate\_task() //

sched\_clock()

recalc\_task\_prio() //

task\_rq\_unlock()

### recalc\_task\_prio函数

recalc\_task\_prio() //更新进程的平均睡眠时间和动态优先级

### schedule函数

schedule() //从运行对列的链表中找到一个进程并执行

rq->curr = next

context\_switch()

switch\_to()

\_\_switch\_to() //arch/arm/kernel/entry-armv.S

add ip, r1, #TI\_CPU\_SAVE

…

ldmia r4, {r4 - sl, fp, sp, pc}

## 进程地址空间

进程的地址空间即用户空间中进程的内存，可以被进程访问的合法地

址空间称为内存区域。

内存描述符，表示进程的地址空间

### 内核堆栈

所有进程（包括内核进程和普通进程）都有一个内核栈，在x86的32位机器上内核栈大小可以为4KB或8KB，这个可以在编译内核的时候配置。内核栈的用途有两个：1）当进程陷入内核态，即内核代表进程执行系统调用时，系统调用的参数就放在内核栈上，内核栈记录着进程的在内核中的调用链；2）在内核栈被配置成8KB大小的情况下，当中断服务程序中断当前进程时，它将使用当前被中断进程的内核栈。

对于用户进程，其既有用户地址空间中的栈，也有它自己的内核栈。而内核进程就只有内核栈

每一个进程（包括普通进程和内核进程）的地址空间都分为用户地址空间和内核地址空间两部分，在32位的x86机器上，用户地址空间的范围是0~3G，内核地址空间的范围是3G~4G.对于不同的进程，其用户地址空间会随着进程不同而不同，但所有进程的内核地址空间则都是一样的。对于内核进程，由于其始终运行在内核态，所以没有用户地址空间，其对应的tast\_struct结构体中的mm域也就被赋值为NULL。而堆的概念应该是只存在于进程的用户地址空间中，所以内核进程是没有堆一说的。内核线程可以用kmalloc 或vmalloc在运行时申请内存。kmalloc或vmalloc申请到的内存在整个内核中都可以使用。比方说 内核线程a申请到了一块内存A, 只要把该内存的首地址传给另一个内核线程b，则在b中同样也可以使用这块内存。

### mm\_struct

// 进程地址空间有关的全部信息

mm\_struct{

start\_code, end\_code //代码段的起始地址和结束地址

start\_data, end\_data //数据段的起始地址和结束地址

pgd\_t \* pgd //指向页全局目录

start\_brk //堆的起始地址

brk //堆的当前最后地址

start\_stack //用户态栈的起始地址

arg\_start, arg\_end //命令行参数的起始和结束地址

env\_start, env\_end //环境变量的起始和结束地址

atomic\_t mm\_users //正在使用该地址的进程数目

atomic\_t mm\_count //主引用计数

//下面两个描述该地址空间的全部内存区域，可访问的地址空间，

超出限制后会引起段错误

struct vm\_area\_struct \* mmap { //以链表形式存放

struct mm\_struct \* vm\_mm;

unsigned long vm\_start //内存区域起始地址

unsigned long vm\_end //内存区域结束地址

}

struct rb\_root mm\_rb //以红黑树形式存放

//所有的mm\_struct结构都以自身的mmlist域连接在一个双向链

表中，链表首元素为init\_mm，即init进程的地址空间

struct list\_head mmlist

}

vm\_area\_struct{ //内存区域结构

struct mm\_struct \* vm\_mm //相关的mm\_struct

unsigned long vm\_start //有权访问的内存区间的首地址

unsigned long vm\_end //有权访问的内存区间的尾地址

pgprot\_t vm\_page\_prot //访问控制权限

unsigned long vm\_flags //标志

struct vm\_operations\_struct \* vm\_ops //与内存区域相关的操作函数

表

}

vm\_operations\_struct{

void (\*open)(struct vm\_area\_struct \* area)

void (\*close)(struct vm\_area\_struct \* area)

}

//创建一个新的线性地址空间

static inline unsigned long do\_mmap(struct file \*file, unsigned long addr,

unsigned long len, unsigned long prot,

unsigned long flag, unsigned long offset)

### vm\_area\_struct

// 进程的用户空间地址

struct vm\_area\_struct {

}

### vm\_struct

//进程的内核空间地址，所描述的内核空间地址范围

//: (3G+896M+8M)~4G。3G~(3G+896M)用来映射物理内存的前896M，//它们之间有简单的线性对应关系，(3G+896M)~(3G+896M+8M)的

//8M空间是安全保护区，用来检测非法指针

struct vm\_struct {

}

### 页表

//顶级页表是页全局目录，arm架构

typedef u32 pmdval\_t

typedef pmdval\_t pgd\_t[2]

//二级页表是中间页目录，arm架构

typedef unsigned long pmd\_t

//最后一级页表简称页表，arm架构

typedef unsigned long pte\_t

pgd\_t \*pgd\_alloc(struct mm\_struct \*mm)

pgd\_t \*new\_pgd = \_\_pgd\_alloc()

\_\_get\_free\_pages(GFP\_KERNEL | \_\_GFP\_REPEAT, 2) //4个页

//将前((3\*1024M – 16M)/2M)\*8置0，用户空间地址置0

memset(new\_pgd, 0, USER\_PTRS\_PER\_PGD \* sizeof(pgd\_t))

init\_pgd = pgd\_offset\_k(0) //init\_mm->pgd

//拷贝init\_mm->pgd映射的内核地址空间部分，所有进程共用相

//同的内核地址空间

memcpy(new\_pgd + (3\*1024M – 16M)/2M , init\_pgd +, size )

// if (!vectors\_high()) start

new\_pud = pud\_alloc(mm, new\_pgd, 0)

new\_pmd = pmd\_alloc(mm, new\_pud, 0)

new\_pte = pte\_alloc\_map(mm, NULL, new\_pmd, 0)

// if (!vectors\_high()) end

### 内存映射

mmap()

SYSCALL\_DEFINE6(mmap\_pgoff)

vm\_mmap\_pgoff()

do\_mmap\_pgoff()

get\_unmapped\_area()

case MAP\_PRIVATE //flags & MAP\_TYPE

mmap\_region()

find\_vma\_links() //查找最靠近申请区的vma

//当前要申请的vma能否跟已经存在prev vma合并

vma\_merge()

//如果没法合并新申请一个vma，然后插入mm vma list

中

kmem\_cache\_zalloc()

file->f\_op->mmap(file, vma) //fs mmap

munmap()

SYSCALL\_DEFINE2(munmap)

profile\_munmap(addr)

vm\_munmap(addr, len)

do\_munmap()

find\_vma()

unmap\_region()

# 系统调用

ARM Linux系统利用SWI指令来从用户空间进入内核空间。SWI指

令用于产生软件中断，从而实现从用户模式到管理模式的变换。X86

上系统调用号通过eax寄存器传递给内核，ebx、ecx、edx、esi、和edi按照顺序传递前5个系统调用的函数参数。

## 系统调用的实现

1. 把系统调用sys\_foo加入到系统调用表，对于大多数体系结构，该表位于entry.s。arm体系中为

arch/arm/kernel/call.S，CALL(sys\_foo)

1. 把系统调用号加入到include/asm-arm/unistd.h中，

#define \_\_NR\_foo 338

1. 实现系统调用，系统调用必须被编译进内核，不能编译成模块，asmlinkage限定词表示仅从栈中提取该函数的参数

asmlinkage long sys\_foo(void)

{

}

用户空间使用：\_syscalln，n从0~6，库函数syscall()

#define \_\_NR\_foo 338

\_syscall0(long,fool)

int main()

{

foo();

}

## glibc系统调用

### fork

weak\_alias (\_\_libc\_fork, fork)

\_\_libc\_fork

pid = INLINE\_SYSCALL (fork, 0)

INTERNAL\_SYSCALL() //嵌入汇编实现

### pthread\_create

compat\_symbol (libpthread, \_\_pthread\_create\_2\_0, pthread\_create,

GLIBC\_2\_0)

\_\_pthread\_create\_2\_0

\_\_pthread\_create\_2\_1()

create\_thread()

do\_clone()

INTERNAL\_SYSCALL()

# 内存管理

## 页

page{

atomic\_t \_count //页的引用计数器

atomic\_t flags //页框状态的标志

void \*virtual //页的虚拟地址

}

//分配2的order次方个连续的物理页

static inline struct page \* alloc\_pages(gfp\_t gfp\_mask, unsigned int order)

//释放物理页

void free\_pages(unsigned long addr, unsigned int order

//将页框永久映射到高端内存

kmap\_high()

map\_new\_virtual()

// no NUMA

alloc\_pages(gfp\_mask, order)

alloc\_pages\_node(numa\_node\_id(), gfp\_mask, order)

\_\_alloc\_pages(gfp\_mask, order, node\_zonelist(nid, gfp\_mask))

\_\_alloc\_pages\_nodemask(gfp\_mask, order, zonelist, NULL)

## 区描述符

zone{

pages\_min //区中保留页的数目

pages\_low, pages\_high //回收页框使用的下界和上界

const char \*name //区的名字

}

## slab

//高速缓存描述符，每个高速缓存由多个slab组成

kmem\_cache{

struct kmem\_cache \*slabp\_cache //包含slab描述符的普通slab高

速缓存

unsigned int slab\_size //单个slab的大小

//高速缓存中的所有slab，partial、full和free

struct kmem\_list3 \*nodelists[MAX\_NUMNODES]

}

//slab描述符，slab由一个或多个物理上连续的页组成

slab{

struct list\_head list //满，部分满或空链表

unsigned long colouroff //slab着色的偏移量

void \*s\_mem //slab中的第一个对象

unsigned int inuse //slab中已分配的对象数

kmem\_bufctl\_t free //第一个空闲对象

unsigned short nodeid

}

//创建高速缓存

struct kmem\_cache \*kmem\_cache\_create (const char \*name,

size\_t size, size\_t align,

unsigned long flags,

void (\*ctor)(void\*, struct kmem\_cache \*, unsigned long),

void (\*dtor)(void\*, struct kmem\_cache \*, unsigned long))

//撤销高速缓存

void kmem\_cache\_destroy(struct kmem\_cache \*cachep)

//从缓存中分配

void \*kmem\_cache\_alloc(struct kmem\_cache \*cachep, gfp\_t flags)

void kmem\_cache\_free(struct kmem\_cache \*cachep, void \*objp)

## 内存分配

//分配的内存在物理地址上连续，虚拟地址页连续flags分配器标志

void \*kmalloc(size\_t size, gfp\_t flags)

\_\_kmalloc()

\_\_do\_kmalloc() //分配slab对象

kmalloc\_slab()

return kmalloc\_dma\_caches[index] //返回高速缓存

slab\_alloc()

\_\_do\_cache\_alloc()

\_\_\_\_cache\_alloc()

cache\_alloc\_refill()

cache\_grow() //给高速缓存分配slab

kmem\_getpages() //给slab分配页框

void kfree(const void \*objp)

//虚拟地址连续，物理地址无需连续

void \*vmalloc(unsigned long size)

\_\_vmalloc\_node\_flags()

\_\_vmalloc\_node()

\_\_vmalloc\_node\_range()

\_\_get\_vm\_area\_node()

kzalloc\_node()

kmalloc\_node()

\_\_kmalloc\_node()

\_\_do\_kmalloc\_node()

alloc\_vmap\_area()

kmalloc\_node()

\_\_insert\_vmap\_area()

setup\_vmalloc\_vm()

\_\_vmalloc\_area\_node()

map\_vm\_area()

kmemleak\_alloc()

void vfree(void \*addr)

## 用户空间

malloc

malloc小于128k的内存，使用brk分配内存，将\_edata往高地址推(只分配虚拟空间，不对应物理内存(因此没有初始化)，第一次读/写数据时，引起内核缺页中断，内核才分配对应的物理内存，然后虚拟地址空间建立映射关系。

malloc大于128k的内存，使用mmap分配内存，在堆和栈之间找一块空闲内存分配(对应独立内存，而且初始化为0。

malloc一次性能申请小内存（小于128KB），分配的是在堆区（heap），用sbrk()进行对齐生长。而malloc一次性申请大内存（大于128KB时）分配到的是在映射区，而不是在堆区，采用的mmap()系统调用进行映射

无论malloc通过sbrk还是mmap实现，分配到的内存只是虚拟内存，而且只是虚拟内存的页号，代表这块空间进程可以用，实际上还没有分配到实际的物理页面。等你的进程访问到这个新分配的内存空间的时候，如果其还没有对应的物理页面分配，就会产生缺页中断，内核这个时候会给进程分配实际的物理页面，以与这个未被映射的虚拟页面对应起来。

# 文件系统

## 超级块对象

super\_block //超级块链表头

//存放已安装文件系统的有关信息，一个超级块对应一个文件系统

super\_block{

dev\_t s\_dev //具体文件系统的块设备标识符

file\_system\_type \*s\_type //文件系统的类型

struct list\_head s\_list //指向所有的超级块的列表

unsigned long s\_blocksize //文件系统中以字节为单位的数据

块的大小

unsigned char s\_dirt //标识超级块是否被修改

unsigned long long s\_maxbytes //允许的最大文件大小(字节)

struct dentry \*s\_root //文件系统安装目录的目录项

struct list\_head s\_files //已打开文件的链表

struct block\_device \*s\_bdev //文件系统被安装的块设备

const struct super\_operations \*s\_op //超级块方法

}

//超级块操作

super\_operations{

struct inode \*(\*alloc\_inode)(struct super\_block \*sb)

void (\*destroy\_inode)(struct inode \*)

}

## 索引节点对象

//存放具体文件的一般信息，inode有两种，一种是VFS的inode，一

种是具体文件系统的inode。前者在内存中，后者在磁盘中

inode{

struct list\_head i\_dentry //目录项列表头

i\_ino //索引节点号，每个inode都是唯一的

i\_mode //文件类型与访问权限

i\_uid //所有者的标识符

dev\_t i\_rdev //实际的设备标识

umode\_t i\_mode //文件的类型和访问权限

i\_atime //上次访问文件的时间

blkcnt\_t i\_blocks //文件所占的块数

super\_block \*i\_sb //指向超级块对象的指针

struct address\_space \*i\_mapping

struct address\_space i\_data

const struct inode\_operations \*i\_op //索引节点操作表

const struct file\_operations \*i\_fop

}

//索引节点操作表

inode\_operations{

int (\*create) (struct inode \*,struct dentry \*,int, struct nameidata \*)

int (\*mkdir) (struct inode \*,struct dentry \*,int)

int (\*mknod) (struct inode \*,struct dentry \*,int,dev\_t)

}

//地址空间

address\_space{

struct inode \*host

const struct address\_space\_operations \*a\_ops

struct address\_space \*assoc\_mapping

}

## 文件对象

存放打开文件与进程之间进行交互的有关信息，描述的是进程已经打

开的文件

file{

mode\_t f\_mode //进程的访问模式

loff\_t f\_pos //当前的文件位移量（文件指针）

struct path f\_path;

#define f\_dentry f\_path.dentry //与文件相关的目录项对象

#define f\_vfsmnt f\_path.mnt //含有该文件的已安装文件系统

const struct file\_operations \*f\_op //指向文件操作表的指针

atomic\_t f\_count //文件对象的引用计数器

struct file\_ra\_state f\_ra //文件预读状态

}

//文件对象操作

file\_operations{

struct module \*owner //拥有者，THIS\_MODULE

ssize\_t (\*read) (struct file \*, char \_\_user \*, size\_t, loff\_t \*)

ssize\_t (\*write) (struct file \*, const char \_\_user \*, size\_t, loff\_t \*)

}

## 目录项对象

存放目录项（也就是文件的特定名称）与对应文件进行链接的有关信

息，只存在于内存中，并没有实际对应的磁盘上的描述，更确切的说

是存在于内存的目录项缓存

dentry{

inode \* d\_inode //与文件名关联的索引节点

dentry \*d\_parent //父目录的目录项对象

super\_block \*d\_sb //文件的超级块对象

struct dentry\_operations \*d\_op //目录项操作指针

}

//目录项操作

dentry\_operations{

int (\*d\_revalidate)(struct dentry \*, struct nameidata \*)

int (\*d\_delete)(struct dentry \*)

}

## 路径名查找

int fastcall path\_lookup(const char \*name, unsigned int flags,

struct nameidata \*nd)

do\_path\_lookup()

//绝对路径查找

nd->mnt = mntget(fs->rootmnt)

nd->dentry = dget(fs->root)

//相对路径查找

nd->mnt = mntget(fs->pwdmnt)

nd->dentry = dget(fs->pwd)

retval = path\_walk(name, nd)

link\_path\_walk(name, nd)

\_\_link\_path\_walk(name, nd)

do\_lookup(nd, &this, &next)

\_\_d\_lookup(nd->dentry, name)

struct nameidata {

struct dentry \*dentry //目录项对象的地址

struct vfsmount \*mnt //已安装文件系统对象的地址

struct qstr last //路径名的最后一个分量

unsigned int flags //查找标志

int last\_type //路径名最后一个分量的类型

unsigned depth //符号链接嵌套的级别

char \*saved\_names[] //与嵌套的符号链接关联的路径名

union {

struct open\_intent open

} intent

}

## 与进程相关的结构

files\_struct{

atomic\_t count

struct fdtable \*fdt

struct fdtable fdtab

int next\_fd

struct embedded\_fd\_set close\_on\_exec\_init

struct embedded\_fd\_set open\_fds\_init

struct file \* fd\_array[NR\_OPEN\_DEFAULT]

}

fs\_struct{

atomic\_t count

rwlock\_t lock

int umask

struct dentry \* root, \* pwd, \* altroot

struct vfsmount \* rootmnt, \* pwdmnt, \* altrootmnt

}

mnt\_namespace{

atomic\_t count

struct vfsmount \* root

struct list\_head list

wait\_queue\_head\_t poll

int event

}

## 文件系统安装

### 文件系统类型

file\_systems //系统中所有文件系统类型链表头

file\_system\_type{

char \*name //文件系统名

int fs\_flags //文件系统类型标志

get\_sb，kill\_sb //函数指针，读超级块和删除超级块方法

struct module \*owner //实现文件系统的模块的指针

struct list\_head fs\_supers //具有相同文件系统类型的超级块对

象链表的头

}

//注册文件系统类型和卸载文件系统类型

struct file\_system\_type ext2\_fs\_type {

.name = "ext2"

.mount = ext2\_mount

.fs\_flags = FS\_REQUIRES\_DEV

}

ext2\_mount(type. dev\_name)

mount\_bdev(dev\_name, ext2\_fill\_super)

struct block\_device \*bdev = blkdev\_get\_by\_path(dev\_name)

lookup\_bdev(dev\_name)

super\_block \*s = sget(test\_bdev\_super, set\_bdev\_super,bdev)

super\_block \*s = alloc\_super()

set\_bdev\_super(s, bdev)

s->s\_type = type

ext2\_fill\_super(s, data /\* fs specific data\*/)

sb->s\_op = &ext2\_sops

struct inode \*root = ext2\_iget(sb, EXT2\_ROOT\_INO)

ext2\_inode \*raw\_inode = ext2\_get\_inode()

inode->i\_op = &ext2\_file\_inode\_operations

inode->i\_fop = &ext2\_file\_operations

int register\_filesystem(&ext2\_fs\_type)

// 添加到文件系统链表，以file\_systems开头

p = find\_filesystem(ext2\_fs\_type.name)

\*p = fs

int unregister\_filesystem(struct file\_system\_type \* fs)

### vfsmount

vfsmount{ //已安装文件系统描述符

struct vfsmount \*mnt\_parent //父文件系统

struct dentry \*mnt\_mountpoint //安装点目录

struct dentry \*mnt\_root //根目录

struct mnt\_namespace \*mnt\_ns //安装了文件系统的进程命名空间

的指针

}

### 安装普通文件系统

//进程命名空间

mnt\_namespace{ //命名空间—已安装文件系统树

atomic\_t count //引用计数器，命名空间的进程数

struct vfsmount \* root //根目录已安装文件系统描述符

struct list\_head list //所有已安装文件系统描述符链表的头

wait\_queue\_head\_t poll //等待队列

int event //事件

}

//安装普通文件系统，mount系统调用，namespace.c

SYSCALL\_DEFINE5(mount (char \_\_user \* dev\_name, char \_\_user \* dir\_name,char \_\_user \* type, unsigned long flags,void \_\_user \* data)

do\_mount(const char \*dev\_name)

user\_path(dir\_name, &path)

user\_path\_at()

user\_path\_at\_empty()

filename\_lookup()

path\_lookupat()

security\_sb\_mount()

do\_new\_mount(&path, dev\_name)

type = get\_fs\_type()

vfs\_kern\_mount(type, dev\_name)

mount\_fs(struct file\_system\_type \*type, dev\_name)

root = type->mount2 //或者

root = type->mount()

ext2\_mount(dev\_name)

mount\_bdev(dev\_name)

do\_add\_mount()

### 安装根文件系统

笼统的来说，虚拟的根文件系统包括三种类型，即Initramfs、cpio-initrd和image-initrd

下面还是先来澄清一下initrd和initramfs的关系（个人理解）：

1. initrd是启动时加载的，initramfs是编译到内核中的

2. initramfs只支持cpio格式，initrd支持cpio格式和传统的image-initrd格式

3. initramfs被解析处理后原始的cpio包(压缩或非压缩)所占的空间 (&\_\_initramfs\_start - &\_\_initramfs\_end)

是作为系统的一部分直接保留在系统中，不会被释放掉， 而对于initrd镜像文件，如果没有在命令行中设置"keepinitd"

命令，那么initrd镜像文件被处理后其原始文件所占的空间 (initrd\_end - initrd\_start)将被释放掉。

4. cpio-initrd的处理和initramfs极其相似，cpio-initrd就是相当于把initramfs处理机制用initrd机制加载

5. 现在普遍用initramfs（嵌入式linux内核）或cpio-initrd（发行版linux的grub），

存在多种类型的Initrd，实际应用中包括无Initrd、Linux Kernel和Initrd打包、Linux Kernel和Initrd分离以及RAMDisk Initrd。

在系统启动的时候，U-Boot会将Linux Kernel和Rootfs加载到内存，并跳转到Linux Kernel的入口地址执行程序

dev\_t ROOT\_DEV //包含根文件系统的磁盘设备号

1. 安装rootfs文件系统

start\_kernel()

setup\_arch(&command\_line)

parse\_tags(tags)

parse\_tag(t)

//等于执行parse\_tag\_core()函数

t->parse(tag)

//给包含根文件系统的磁盘设备号赋值

ROOT\_DEV = old\_decode\_dev

(tag->u.core.rootdev)

vfs\_caches\_init()

mnt\_init()

init\_rootfs() //fs/ramfs/inode.c 第一步 安装rootfs

//注册根文件系统类型，下面会引用

register\_filesystem(&rootfs\_fs\_type)

init\_mount\_tree() //第二步

do\_kern\_mount("rootfs", 0, "rootfs", NULL)

//获取前面注册的根文件系统类型

get\_fs\_type(fstype="rootfs")

vfs\_kern\_mount()

//等同于rootfs\_get\_sb()函数

type->get\_sb() // 分配新的超级块

s = sget(fs\_type)

get\_sb\_nodev(ramfs\_fill\_super)

// 调用格式为fill\_super()

ramfs\_fill\_super()

inode = ramfs\_get\_inode(sb,

S\_IFDIR | 0755, 0)

//inode节点赋值，S\_IFDIR

inode->i\_op = &ramfs\_dir\_inode\_operations

inode->i\_fop = &simple\_dir\_operations

d\_alloc\_root(inode)

//根目录名字

struct qstr name = { .name = "/", }

d\_alloc()

init\_waitqueue\_head()

set\_fs\_pwd()

set\_fs\_root()

1. 安装实际根文件系统

start\_kernel()

rest\_init()

kernel\_init()

prepare\_namespace() //实际根文件系统的安装

root\_device\_name = saved\_root\_name

ROOT\_DEV = name\_to\_dev\_t(root\_device\_name)

initrd\_load()

mount\_root() //挂载

/\*用系统调用创建"/dev/root"\*/

create\_dev("/dev/root", ROOT\_DEV)

sys\_mknod(name, S\_IFBLK|0600,)

mount\_block\_root("/dev/root", root\_mountflags)

/\*获得文件系统类型，如果在bootoption里

有，则就为这个文件系统类型，如果没有指

定，则返回ilesytem链上所有类型，下面再

对每个进行尝试.\*/

get\_fs\_names(fs\_names)

do\_mount\_root() //逐个匹配挂载根设备

/\*mount系统调用来做实际的安装文件系

统工作\*/

sys\_mount(name, "/root")

/\*改变当前路径到根目录\*/

sys\_chdir("/root")

/\*将挂载点从当前目录【/root】（在mount\_root函数

中设置的）移到根目录\*/

sys\_mount(".", "/", NULL, MS\_MOVE, NULL)

do\_mount()

/\*当前目录即【/root】（真正文件系统挂载的目录）

做为系统根目录\*/

sys\_chroot(".")

### 卸载文件系统

## VFS系统调用

### sys\_open

long sys\_open(const char \_\_user \*filename, int flags, int mode)

do\_sys\_open()

getname(filename) //从进程的地址空间读取文件的路径名

get\_unused\_fd() //获取空的文件描述符

do\_filp\_open(dfd, tmp, flags, mode)

open\_namei(&nd)

path\_lookup\_open() //打开文件

\_\_path\_lookup\_intent\_open()

do\_path\_lookup(nd)

path\_lookup\_create() //look up parent

\_\_path\_lookup\_intent\_open()

do\_path\_lookup()

path\_walk (name, nd)

open\_namei\_create() //create

vfs\_create(dir->d\_inode, path->dentry, mode, nd)

//使用父索引节点创建新的索引节点

//struct inode \*dir

dir->i\_op->create(dir, dentry, mode, nd)

return nameidata\_to\_filp(&nd, flags) //返回file对象

filp = nd->intent.open.file 或者：

filp = \_\_dentry\_open(nd->dentry, nd->mnt, flags, filp)

f->f\_op = fops\_get(inode->i\_fop) //赋值f->f\_op

fd\_install(fd, f) //加载file对象，将file对象和fd绑定

struct files\_struct \*files = current->files

fdt = files\_fdtable(files)

rcu\_assign\_pointer(fdt->fd[fd], file)

### sys\_write

sys\_write()

vfs\_write()

if (file->f\_op->write)

ret = file->f\_op->write(file, buf, count, pos)

else

ret = do\_sync\_write(file, buf, count, pos)

ret = filp->f\_op->aio\_write()

// ext2

ext2\_file\_operations::write()

new\_sync\_write(struct file \*filp)

filp->f\_op->write\_iter()

generic\_file\_write\_iter()

\_\_generic\_file\_write\_iter()

generic\_perform\_write()

mapping->a\_ops ->write\_begin()

ext2\_write\_begin()

iov\_iter\_copy\_from\_user\_atomic

mapping->a\_ops ->write\_end()

ext2\_write\_end()

generic\_write\_sync()

vfs\_fsync\_range()

file->f\_op->fsync()

ext2\_fsync()

generic\_file\_fsync()

\_\_generic\_file\_fsync()

blkdev\_issue\_flush()

submit\_bio\_wait()

submit\_bio()

generic\_make\_request()

### sys\_read

SYSCALL\_DEFINE3(read, fd)

struct fd f = fdget\_pos(fd)

vfs\_read()

ret = file->f\_op->read(file, buf, count, pos)

ext2\_file\_operations::read()

// ext2\_file\_operations::read

new\_sync\_read(struct file \*filp)

filp->f\_op->read\_iter()

generic\_file\_read\_iter()

do\_generic\_file\_read()

find\_get\_page()

// if find\_get\_page() return NULL

page\_cache\_sync\_readahead

// if page cache OK

mapping->a\_ops->readpage(filp, page)

ext2\_readpage()

mpage\_readpage()

do\_mpage\_readpage()

mpage\_alloc()

bio\_alloc()

mpage\_bio\_submit()

submit\_bio()

generic\_make\_request()

i\_size\_read()

mark\_page\_accessed()

copy\_page\_to\_iter() // copy to user space

## 文件加锁

//劝告锁和强制锁

flock() 只产生劝告锁

fcntl()

//文件锁数据结构

file\_lock{

struct list\_head fl\_link //活动或阻塞列表的指针

struct list\_head fl\_block //锁的等待者列表的指针

fl\_owner\_t fl\_owner //文件的所有者

unsigned int fl\_pid //进程拥有者的pid

wait\_queue\_head\_t fl\_wait //阻塞进程的等待队列

struct file \*fl\_file //指向文件对象的指针

unsigned char fl\_flags //锁标志

unsigned char fl\_type //锁类型

loff\_t fl\_start //被锁区字段的起始偏移量

loff\_t fl\_end //被锁区字段的末尾偏移量

struct fasync\_struct \* fl\_fasync //租借锁的中断通知

unsigned long fl\_break\_time //租借结束前额剩余时间

struct file\_lock\_operations \*fl\_ops //指向文件锁操作的指针

struct lock\_manager\_operations \*fl\_lmops //指向文件锁管理的指

针

union {

struct nfs\_lock\_info nfs\_fl;

struct nfs4\_lock\_info nfs4\_fl;

} fl\_u //具体文件系统的信息

}

### FL\_FLOCK锁

### FL\_POSIX锁

## devfs

static struct file\_system\_type dev\_fs\_type = {

.name = "devtmpfs",

.mount = dev\_mount,

.kill\_sb = kill\_litter\_super,

}

dev\_mount()

mount\_single(fs\_type, flags, data, shmem\_fill\_super)

shmem\_fill\_super()

shmem\_get\_inode(S\_IFDIR | S\_IRWXUGO | S\_ISVTX)

init\_special\_inode(inode, mode, dev)

if (S\_ISCHR(mode)) {

inode->i\_fop = &def\_chr\_fops

inode->i\_rdev = rdev

const struct file\_operations def\_chr\_fops = {

.open = chrdev\_open,

.llseek = noop\_llseek,

};

chrdev\_open()

kobj\_lookup(cdev\_map, inode->i\_rdev, &idx)

filp->f\_op->open(inode, filp) // misc\_fops.open()

//start\_kernel()

do\_basic\_setup()

driver\_init()

devtmpfs\_init()

register\_filesystem(&dev\_fs\_type)

kthread\_run(devtmpfsd, &err, "kdevtmpfs")

sys\_mount("devtmpfs", "/", "devtmpfs", MS\_SILENT,

options)

do\_new\_mount()

## yaffs2

yaffs\_AllocateChunk() //存储空间的分配

yaffs\_FindBlockForAllocation()

yaffs\_GetBlockInfo()

yaffs\_CheckSpaceForAllocation() //检查Flash上是否有足够的可用

空间

yaffs\_DeleteChunk() //chunk的释放

yaffs\_BlockBecameDirty() //删除

yaffs\_GetTnode()

yaffs\_CreateTnodes()

yaffs\_FreeTnode()

# I/O设备

## 设备驱动相关

### 设备对象

device{ //设备对象，一个设备只有一个驱动

device\_driver \*driver //指向控制设备驱动程序

bus\_type \* bus //指向所连总线

struct device \*parent //指向父设备的指针

struct kobject kobj //内嵌的kobject结构

void \*platform\_data //平台数据，无确定类型

void (\*release)(struct device \* dev) //释放设备描述符的会掉函

数

}

//设备注册

int device\_register(struct device \*dev)

device\_add() //添加设备

//设备注销

void device\_unregister(struct device \* dev)

**device\_add函数**

int device\_add(struct device \*dev)

dev = get\_device(dev) //增加该设备的引用计数

kobject\_add() //加入kobject

//创建sys目录下设备的uevent属性文件

device\_create\_file(dev, &dev->uevent\_attr)

bus\_attach\_device(dev) //寻找设备对应的驱动

device\_attach(dev) //尝试为设备找到一个驱动

//如果设备结构里面没有驱动则查找驱动

\_\_device\_attach()

driver\_probe\_device(drv, dev)

really\_probe(dev, drv)

dev->bus->probe(dev)

drv->probe(dev)

driver\_bound(dev) //将设备加入驱动的设备链表

### 驱动程序

device\_driver{ //驱动程序，一个驱动可对应多个设备

klist klist\_devices //驱动程序支持的所有设备列表的首部

const char \* name //设备驱动程序名称

struct bus\_type \* bus //指向总线描述符的指针

int (\*probe) (struct device \* dev) //检测驱动是否可以控制设备

int (\*remove) (struct device \* dev) //移走设备时所调用的方法

void (\*shutdown) (struct device \* dev) //断电关闭时调用的方法

int (\*suspend) (struct device \* dev, pm\_message\_t state) //低功率挂起

int (\*resume) (struct device \* dev) //恢复正常状态时的方法

}

//驱动程序注册

int driver\_register(struct device\_driver \* drv)

//驱动程序注销

void driver\_unregister(struct device\_driver \* drv)

### 总线

bus\_type{

const char \* name //总线类型的名称

struct kset drivers //驱动程序的kobject集合

struct kset devices //设备的kobjectj集合

struct bus\_attribute \* bus\_attrs

struct device\_attribute \* dev\_attrs

struct driver\_attribute \* drv\_attrs

int (\*match)(struct device \* dev, struct device\_driver \* drv) //检验

//给定的设备驱动程序是否支持特定设备的方法

}

//总线注册

int bus\_register(struct bus\_type \* bus)

//总线注销

void bus\_unregister(struct bus\_type \* bus)

### 平台设备

//资源

resource{

resource\_size\_t start //设备在cpu总线上的线性起始物理地址

resource\_size\_t end //设备在cpu总线上的线性结尾物理地址

const char \*name //设备的名称

unsigned long flags //设备的一些共性和特性的标志位

struct resource \*parent, \*sibling, \*child

}

//平台设备

platform\_device {

const char \* name

u32 id //

struct device dev //设备对象

u32 num\_resources //资源个数

struct resource \* resource //资源

}

//注册平台设备

int platform\_device\_register(struct platform\_device \* pdev)

platform\_device\_add(pdev)

pdev->dev.bus = &platform\_bus\_type

device\_add(pdev->dev)

bus\_add\_device(dev) //总线添加设备

struct bus\_type \* bus = get\_bus(dev->bus)

//注册多个平台设备

int platform\_add\_devices(struct platform\_device \*\*devs, int num)

platform\_device\_register(devs[i])

//注销平台设备

void platform\_device\_unregister(struct platform\_device \* pdev)

platform\_device\_del(pdev)

//平台驱动

platform\_driver{

struct device\_driver driver //设备驱动

int (\*probe)(struct platform\_device \*)

int (\*remove)(struct platform\_device \*)

}

platform\_driver\_register(struct platform\_driver \*drv) //平台驱动注册

drv->driver.bus = &platform\_bus\_type //平台总线类型

drv->driver.probe = platform\_drv\_probe //平台驱动的probe

driver\_register(drv->driver) //驱动注册

bus\_add\_driver()

driver\_attach()

bus\_for\_each\_dev() //遍历设备，查找和驱动匹配的设备

\_\_driver\_attach(struct device \* dev, void \* data)

struct device\_driver \* drv = data

driver\_probe\_device(drv, dev)

really\_probe(dev, drv)

dev->driver = drv //关联驱动

ret = drv->probe(dev)

//对于平台驱动，相当于执行：

platform\_drv\_probe()

platform\_driver::probe()

dm9000\_probe()

platform\_driver\_unregister(struct platform\_driver \*drv) //平台驱动注销

driver\_unregister()

### kobject

kobject {

const char \* k\_name

char name[KOBJ\_NAME\_LEN]

struct kref kref //容器的引用计数

struct list\_head entry //所插入的链表

struct kobject \* parent

struct kset \* kset

struct kobj\_type \* ktype //类型描述

struct dentry \* dentry //对应的sysfs文件的dentry

wait\_queue\_head\_t poll

}

### 设备类

//设备类，是指提供的用户接口相似的一类设备的集合，常见的设备

//类的有block、tty、input、usb等等

class{ //设备类

const char \* name

struct module \* owner

struct kset subsys

struct list\_head children

struct list\_head devices

struct list\_head interfaces

}

class\_device{ //设备类的设备

struct list\_head node

struct kobject kobj

struct class \* class

dev\_t devt

struct device \* dev //设备对象

}

struct class \*class\_create(struct module \*owner, const char \*name)

void class\_destroy(struct class \*cls)

struct class\_device \*class\_device\_create(struct class \*cls,

struct class\_device \*parent, dev\_t devt,

struct device \*device, const char \*fmt, ...)

class\_device\_add()

class\_device\_unregister(struct class\_device \*class\_dev)

### 设备号

主设备号编码12位，次设备号编码20位，把两个参数合成一个32位dev\_t变量，高20位为主设备号号，低20位为次设备号。MAJOR宏提取主设备号，MINOR宏提取次设备号，MKDEV宏把主设备号和次设备号合并成一个dev\_t值

### 模块

module{

}

当代码编译成模块时，会定义MODULE宏，否则不会

//include/linux/init.h

**编译进内核时的宏定义：**

#define module\_init(x) \_\_initcall(x)

#define module\_exit(x) \_\_exitcall(x)

模块编译进内核原理：

驱动初始化函数就会带有.initcall6.init属性，那么在其编译时将会被

安排到\_\_initcall\_start和\_\_initcall\_end范围之间，在kernel启动时，会执行kernel\_init->do\_basic\_setup->do\_initcalls，do\_initcalls中会

把.initcall.init中的函数依次执行一遍。

### 驱动加载

start\_kernel()

rest\_init(void)

kernel\_thread(kernel\_init, NULL, CLONE\_FS)

kernel\_init()

kernel\_init\_freeable()

do\_basic\_setup()

// do\_initcalls中会把.initcall.init中的函数依次执

行一遍，进行驱动加载

do\_initcalls()

**编译成模块时的宏定义**：

#define module\_init(initfn) \

static inline initcall\_t \_\_inittest(void) \

{ return initfn; } \ //对参数做静态类型检查

//通过gcc的别名特性将init\_module与initfn等同起来，下同

int init\_module(void) \_\_attribute\_\_((alias(#initfn)));

/\* This is only required if you want to be unloadable. \*/

#define module\_exit(exitfn) \

static inline exitcall\_t \_\_exittest(void) \

{ return exitfn; } \

void cleanup\_module(void) \_\_attribute\_\_((alias(#exitfn)));

//内核中加载和卸载模块的系统调用

sys\_init\_module(void \_\_user \*umod,unsigned long len,

const char \_\_user \*uargs)

load\_module(umod, len, uargs)

sys\_delete\_module(const char \_\_user \*name\_user, unsigned int flags)

insmod：模块加载命令

rmmod：模块卸载命令

加载和卸载模块的固定函数名字：

init\_module() 和 cleanup\_module()

## 字符设备

cdev{ //字符设备结构

struct kobject kobj //内嵌的kobject

struct module \*owner //指向实现驱动程序模块的指针

dev\_t dev //主设备号和次设备号

struct file\_operations \*ops //指向设备驱动程序文件操作表

struct list\_head list //与字符设备文件对应的索引节点链表的头

unsigned int count //给设备驱动程序分配的设备号范围的大小

}

char\_device\_struct{

struct char\_device\_struct \*next

unsigned int major //设备号范围的主设备号

unsigned int baseminor //设备号范围的次设备号

int minorct //设备号范围的大小

char name[64]

struct file\_operations \*fops

struct cdev \*cdev

}

char\_device\_struct \*chrdevs[] //字符设备结构链表头，少于255

kobj\_map \*cdev\_map //字符设备映射链表

//注册字符设备

int register\_chrdev(unsigned int major, const char \*name,

const struct file\_operations \*fops)

\_\_register\_chrdev\_region() //返回char\_device\_struct结构

cdev\_alloc() //动态的分配cdev描述符

cdev\_add() ////在设备驱动程序模型中注册一个cdev描述符

kobj\_map() //把指定的设备号范围加入散列表中

int unregister\_chrdev(unsigned int major, const char \*name)

cdev\_del()

//动态分配一个主设备号

int alloc\_chrdev\_region(dev\_t \*dev, unsigned baseminor, unsigned count,

const char \*name)

int chrdev\_open(struct inode \* inode, struct file \* filp)

## 块设备

block\_device{

dev\_t bd\_dev //块设备的主设备号和次设备号

}

//注册块设备驱动

int register\_blkdev(unsigned int major, const char \*name)

//注销块设备驱动

int unregister\_blkdev(unsigned int major, const char \*name)

mtd\_partition{

char \*name

u\_int32\_t size

u\_int32\_t offset

u\_int32\_t mask\_flags

struct nand\_ecclayout \*ecclayout

struct mtd\_info \*\*mtdp

}

int add\_mtd\_partitions(struct mtd\_info \*master,

const struct mtd\_partition \*parts,

int nbparts)

### emmc

#### card 层

mmc\_blk\_init()

register\_blkdev()

mmc\_register\_driver(&mmc\_driver)

mmc\_blk\_probe()

mmc\_blk\_probe()

mmc\_blk\_alloc()

mmc\_blk\_alloc\_req()

mmc\_init\_queue()

mmc\_init\_queue()

mq->queue = blk\_init\_queue(mmc\_request\_fn, lock)

blk\_init\_queue\_node(mmc\_request\_fn)

blk\_init\_allocated\_queue(mmc\_request\_fn)

q->request\_fn = mmc\_request\_fn

blk\_queue\_make\_request(q, blk\_queue\_bio)

q->make\_request\_fn = blk\_queue\_bio

generic\_make\_request(struct bio \*bio)

request\_queue \*q = bdev\_get\_queue(bio->bi\_bdev)

q->make\_request\_fn(q, bio)

blk\_queue\_bio(q)

\_\_blk\_run\_queue(q)

\_\_blk\_run\_queue\_uncond(q)

q->request\_fn(q)

mmc\_request\_fn()

#### core 层

mmc\_init()

mmc\_register\_bus()

sdio\_register\_bus()

#### host 层

//android-goldfish.c

goldfish\_mmc\_probe()

mmc\_alloc\_host()

mmc->ops = &goldfish\_mmc\_ops

mmc\_add\_host(mmc)

## 网络接口

net\_device{ //网络设备结构

char name[IFNAMSIZ]

struct hlist\_node name\_hlist

unsigned long mem\_end

unsigned long mem\_start

int (\*open)(struct net\_device \*dev)

struct device dev

}

//注册网络设备

int register\_netdev(struct net\_device \*dev)

//注销网络设备

void unregister\_netdevice(struct net\_device \*dev)

## 设备文件系统

//用来创建设备文件，把IO设备当做设备文件这种特殊的文件来处理

//设备文件分为块和字符：

块设备的数据可以被随机访问，字符设备不一定能被随机访问

long sys\_mknod(const char \_\_user \*filename, int mode, unsigned dev)

sys\_mknodat(AT\_FDCWD, filename, mode, dev)

getname(filename)

do\_path\_lookup()

lookup\_create()

do\_path\_lookup()

vfs\_mknod()

dir->i\_op->mknod(dir, dentry, mode, dev)

## MTD Nand

B、MTD原始设备

//mtd结构

mtd\_info{

u\_char type

u\_int32\_t flags

u\_int32\_t size //mtd大小

u\_int32\_t erasesize

}

//设备分区的结构

mtd\_part{

}

add\_mtd\_device()

del\_mtd\_device()

C、MTD设备层

## i2c驱动

i2c\_driver {

int (\*probe)(struct i2c\_client \*, const struct i2c\_device\_id \*)

int (\*remove)(struct i2c\_client \*)

struct device\_driver driver

const struct i2c\_device\_id \*id\_table

int (\*detect)(struct i2c\_client \*, struct i2c\_board\_info \*)

}

device\_driver {

of\_device\_id \*of\_match\_table

}

i2c\_add\_driver(i2c\_driver \*driver)

i2c\_register\_driver(THIS\_MODULE, driver)

driver->driver.bus = &i2c\_bus\_type

driver\_register(&driver->driver)

bus\_add\_driver()

module\_add\_driver()

i2c\_for\_each\_dev(driver, \_\_process\_new\_driver)

bus\_for\_each\_dev(&i2c\_bus\_type, NULL, driver,

\_\_process\_new\_driver)

\_\_process\_new\_driver(struct device \*dev, void \*data)

i2c\_do\_add\_adapter(data, to\_i2c\_adapter(dev))

i2c\_detect(adap, driver)

i2c\_detect\_address()

i2c\_default\_probe()

driver->detect(temp\_client, &info)

driver->attach\_adapter(adap)

i2c\_master\_send

i2c\_master\_send(const struct i2c\_client \*client, const char \*buf, int

count)

msg.buf = (char \*)buf

msg.addr = client->addr

i2c\_transfer(adap, &msg, 1)

\_\_i2c\_transfer(adap, msgs,1)

trace\_i2c\_write()

i2c\_master\_recv(const struct i2c\_client \*client, char \*buf, int count)

i2c\_transfer(adap, &msg, 1)

# 同步与加锁

## 每CPU变量

per\_cpu(var, cpu) //为CPU选择一个每CPU数组元素

## 原子操作

typedef struct {

int counter

} atomic\_t //x86\_64体系下原子操作类型

atomic\_read(v) //返回\*v

ACCESS\_ONCE((v)->counter)

atomic\_add(int i, atomic\_t \*v) //给\*v增加i

atomic\_add\_return(i, v)

ATOMIC\_OP\_RETURN(add, +)

cmpxchg()

\_\_cmpxchg\_mb()

smp\_mb()

\_\_cmpxchg()

asm volatile()

smp\_mb()

## 优化和内存屏障

// 优化屏障

barrier() 展开为\_\_asm\_\_ \_\_volatile\_\_("": : :"memory")

表示禁止编译器重排barrier()之前和之后的指令

// 内存屏障

mb(), smp\_mb()

表示原语之后的操作开始执行前原语之前的操作已经完成，由硬件实现

## 自旋锁

spinlock\_t{

break\_lock //表示进程正在忙等自旋锁

}

spin\_lock() //循环，直到自旋锁变为1（未锁），把自旋锁置为0（锁上）

raw\_spin\_lock()

\_raw\_spin\_lock()

\_\_raw\_spin\_lock()

preempt\_disable() //禁用内核抢占

spin\_acquire()

LOCK\_CONTENDED()

do\_raw\_spin\_trylock()

arch\_spin\_trylock()

do\_raw\_spin\_lock()

arch\_spin\_lock

asm()

spin\_unlock() //把自旋锁置为1（未锁）

## 内核信号量

semaphore{

atomic\_t count //大于0，资源是空闲的；等于0，信号量忙，但

没有进程等待；小于0，资源不可用，并至少有

一个进程等待

wait\_queue\_head\_t wait //存放等待队列列表的地址，存放所有等

待资源的睡眠进程，如果count大于等于

1. 等待队列为空

int sleepers //表示是否有一些进程在信号量上睡眠的标志

}

//初始化信号量

sema\_init(struct semaphore \*sem, int val)

//创建互斥信号量

DECLARE\_MUTEX(name)

//初始化一个动态创建的信号量

init\_MUTEX(struct semaphore \*sem)

up() //释放信号量

down() //获取信号量

raw\_spin\_lock\_irqsave()

sem->count

\_\_down

raw\_spin\_unlock\_irqrestore()

### 读写信号量

## IPC信号量

semget() //c创建IPC信号量

semop()

semctl()

sem\_queue{ //挂起请求队列，用来标识正在等待数组中的一个或多

个信号量的进程

struct sem\_queue \* next /\* next entry in the queue \*/

struct sem\_queue \*\* prev //previous entry in the queue,

\*(q->prev) == q

struct task\_struct \*sleeper

int pid

struct sem\_array \* sma //IPC信号量的描述符指针

}

## 互斥体

mutex{

atomic\_t count

spinlock\_t wait\_lock

struct list\_head wait\_list

}

asm-generic/mutex.h

asm-generic/mutex-dec.h

mutex\_lock()

\_\_mutex\_fastpath\_lock(\_\_mutex\_lock\_slowpath)

atomic\_dec\_return()

atomic\_sub\_return()

asm()

//如果获取锁不成功，则睡眠线程

\_\_mutex\_lock\_slowpath()

\_\_mutex\_lock\_common()

# 进程通信

## 管道

pipe\_inode\_info{

wait\_queue\_head\_t wait //管道等待队列

struct pipe\_buffer bufs[PIPE\_BUFFERS] //管道缓存区描述字

符

}

pipe() //创建管道系统调用

read() //从管道读取数据

write() //向管道写数据

## IPC消息

msg\_queue {

kern\_ipc\_perm q\_perm

list\_head q\_ messages //队列中的消息列表

list\_head q\_receivers //接收消息的进程列表

list\_head q\_senders //发送消息的进程列表

}

msg\_msg{

struct list\_head m\_list //用于消息链表的指针

long m\_type //消息类型

int m\_ts //消息正文的大小

struct msg\_msgseg \* next //消息的下一部分

void \*security //安全数据结构指针

}

do\_msgsnd() //发送消息，msgsnd()

do\_msgrcv() //接收消息，msgrcv()

## IPC共享内存

## 信号

# 中断与定时器

## 中断

### 中断描述符表IDT

是一个系统表，与每一个中断或异常向量相联系，每一个向量在表中

有相应的中断或异常处理程序的入口地址。IDT包含3种类型的描述符：任务门，中断门和陷阱门

linux IDT分类

中断门：

void set\_intr\_gate(int nr, void \*func) //

系统门：

void set\_system\_gate(int nr, void \*func)

系统中断门：

void set\_system\_intr\_gate(unsigned int n, void \*addr)

陷阱门：

void \_\_init set\_trap\_gate(unsigned int n, void \*addr)

任务门：

void \_\_init set\_task\_gate(unsigned int n, unsigned int gdt\_entry)

extern struct desc\_struct idt\_table[]

中断向量描述符

irq\_desc\_t{

irq\_flow\_handler\_t handle\_irq //指向PIC对象，当前中断处理

函数入口

void \*handler\_data //指向PIC方法所使用的数据

struct irq\_chip \*chip //底层硬件访问

struct irqaction \*action //标识当出现IRQ时要调用的中断服务

例程，中断处理函数链表

irq\_count //中断计数器

spinlock\_t lock //用于串行访问IRQ描述符和PIC的自旋锁

status //描述IRQ线状态的一组标志

const char \*name //中断名称

} irq\_desc

irq\_chip{ //操作底层硬件

const char \*name

unsigned int (\*startup)(unsigned int irq)

void (\*shutdown)(unsigned int irq)

}

irqaction{

irq\_handler\_t handler //指向一个I/O设备的中断服务例程，中

断处理函数

flags //描述IRQ与I/O设备之间的关系，中断标志

const char \*name //I/O设备名，中断名字

void \*dev\_id //I/O 设备的私有字段（可能为主设备号和次设

备号），传给中断处理函数的参数

irq //IRQ线，中断号

struct proc\_dir\_entry \*dir //指向与IRQn相关的/proc/irq/n

目录的描述符

}

### 中断相关的处理函数

request\_irq() //注册中断处理程序

int request\_irq(unsigned int irq, //表示要分配的中断号

irq\_handler\_t handler, //中断处理函数指针

unsigned long irqflags, //中断处理程序标志，

eg:IRQF\_DISABLED

const char \*devname, //与中断相关的设备名

void \*dev\_id) //用于共享中断线

setup\_irq(irq, action)

irq\_chip\_set\_defaults(desc->chip)

free\_irq() //释放中断处理程序

free\_irq(unsigned int irq, void \*dev\_id)

编写中断处理程序的格式

static irqreturn\_t intr\_handler(int irq,void \*dev)

//中断处理

bne asm\_do\_IRQ //中断向量表的汇编程序

//arm处理器的中断总入口函数

void asm\_do\_IRQ(unsigned int irq, struct pt\_regs \*regs)

desc\_handle\_irq (irq, desc)

//逐个调用irqaction链表中的处理函数

desc->handle\_irq(irq, desc)

unsigned int do\_IRQ(struct pt\_regs \*regs)

//处理中断事件

irqreturn\_t handle\_IRQ\_event(unsigned int irq, struct irqaction \*action)

local\_irq\_enable() //使能中断

//中断初始化

init\_IRQ()

// init\_arch\_irq = mdesc->init\_irq

init\_arch\_irq() //函数指针，void (\*init\_arch\_irq)(void) \_\_initdata

= NULL

//MACHINE\_START(S3C2440, "SMDK2440")

// .init\_irq = s3c24xx\_init\_irq,

s3c24xx\_init\_irq() //struct machine\_desc \_\_mach\_desc\_S3C2440

放在.arch.info.init段里面

// extern struct irq\_desc irq\_desc[NR\_IRQS]全局结构

set\_irq\_handler(irqno, handle\_edge\_irq)

handle\_edge\_irq (unsigned int irq, struct irq\_desc \*desc)

handle\_IRQ\_event(irq, action)

### 软中断

static struct softirq\_action softirq\_vec[32]

softirq\_action{

void (\*action)(struct softirq\_action \*)

void \*data

}

do\_softirq()

open\_softirq() //注册软中断

raise\_softirq() //挂起软中断

### tasklet

tasklet\_struct{

struct tasklet\_struct \*next //下一个tasklet

unsigned long state //tasklet状态

atomic\_t count //引用计数器

void (\*func)(unsigned long) //tasklet处理函数

unsigned long data //处理函数的参数

}

//tasklet调度

tasklet\_schedule(struct tasklet\_struct \*t)

tasklet\_hi\_schedule(struct tasklet\_struct \*t)

void tasklet\_init(struct tasklet\_struct \*t,

void (\*func)(unsigned long), unsigned long data)

### 工作队列

//工作者线程

workqueue\_struct{

struct cpu\_workqueue\_struct \*cpu\_wq;

struct list\_head list;

const char \*name;

int singlethread;

int freezeable;

}

//表示工作的数据结构

work\_struct{

atomic\_long\_t data

struct list\_head entry

work\_func\_t func

}

worker\_thread()

schedule\_work(struct work\_struct \*work) //调度工作队列

flush\_scheduled\_work() //刷新工作队列

create\_workqueue() //创建工作队列

## 定时器

HZ //节拍率，时钟中断的频率，在asm-arm/param.h中定义

jiffies //用来记录自系统启动以来产生的节拍总数，linux/jiffies.h

get\_jiffies\_64() //获取jiffies或jiffies\_64的值

struct timespec xtime //墙上时间，存放当前时间和日期

timespec{

time\_t tv\_sec //存放自UTC午夜以来经过的秒数

long tv\_nsec //存放自上一秒开始经过的纳秒数

}

time\_init() //在内核初始化时建立计时体系结构

时钟中断处理程序：

static void tick\_periodic(int cpu)

do\_timer(1)

update\_times(ticks) //更新xtime值

update\_process\_times()

scheduler\_tick()

timer\_interrupt()

### 定时器使用

timer\_list{

struct list\_head entry //定时器列表的入口

void (\*function)(unsigned long) //定时器到期时执行函数的地址

data //传递给定时器函数的参数

expires //定时器到期时间，以jiffies为单位的定时值

}

//初始化一个定时器结构

init\_timer(struct timer\_list \*timer)

//添加一个定时器

void add\_timer(struct timer\_list \*timer)

//修改一个定时器

int mod\_timer(struct timer\_list \*timer, unsigned long expires)

//删除一个定时器

int del\_timer(struct timer\_list \*timer)

//定时器运行

void tick\_periodic(int cpu)

update\_process\_times(user\_mode(get\_irq\_regs()))

run\_local\_timers()

raise\_softirq() //处理软中断

### 进程延迟执行

//将任务设置为可中断睡眠状态

set\_current\_state(TASK\_INTERRUPTIBLE)

//让相应的进程睡眠到指定时间后重新执行

schedule\_timeout()

### 系统调用

sys\_time()

sys\_gettimeofday()

sys\_adjtimex()

sys\_setitimer()

POSIX

# 虚拟化技术

## container

linux中称谓的container在内核层面由两个独立的机制保证，一个保证资源的隔离性，名为namespace；一个进行资源的控制，名为cgroup

namespace：

inux现有的namespace有6种： uts, pid, ipc, mnt, net和user

struct task\_struct{

struct nsproxy \*nsproxy{

struct uts\_namespace \*uts\_ns

struct ipc\_namespace \*ipc\_ns

struct mnt\_namespace \*mnt\_ns

struct pid\_namespace \*pid\_ns

}

}

copy\_process()

copy\_namespaces()

create\_new\_namespaces()

copy\_mnt\_ns()

copy\_utsname()

copy\_ipcs()

copy\_pid\_ns()

cgroup：

## Docker

## KVM和QEMU

# 网络协议栈

struct socket {

socket\_state state

unsigned long flags

const struct proto\_ops \*ops

struct fasync\_struct \*fasync\_list

struct file \*file

struct sock \*sk

wait\_queue\_head\_t wait

short type

}

// ipv4

struct inet\_connection\_sock\_af\_ops ipv4\_specific = {

.queue\_xmit = ip\_queue\_xmit

}

struct proto tcp\_prot = {

.init = tcp\_v4\_init\_sock

}

struct proto\_ops inet\_stream\_ops = {

.connect = inet\_stream\_connect

}

struct inet\_protosw inetsw\_array[] {

{

.type = SOCK\_STREAM

.protocol = IPPROTO\_TCP

.prot = &tcp\_prot

.ops = &inet\_stream\_ops

}

}

struct net\_proto\_family inet\_family\_ops {

.family = PF\_INET,

.create = inet\_create,

}

inet\_bind(struct socket \*sock, struct sockaddr \*uaddr, int addr\_len)

inet->inet\_rcv\_saddr = inet->inet\_saddr = addr->sin\_addr.s\_addr

struct sock \*sk = sock->sk;

struct inet\_sock \*inet = inet\_sk(sk)

inet\_stream\_connect(struct socket \*sock, struct sockaddr \*uaddr, int

addr\_len, int flags)

\_\_inet\_stream\_connect(sock, uaddr, addr\_len, flags)

sk->sk\_prot->connect(sk, uaddr, addr\_len)

tcp\_v4\_connect(struct sock \*sk, struct sockaddr \*uaddr, int

addr\_len) // for ipv4 tcp

ip\_route\_connect()

tcp\_connect(sk)

tcp\_connect\_queue\_skb(sk, buff)

tcp\_ecn\_send\_syn(sk, buff)

tcp\_send\_syn\_data(sk, buff)

tcp\_transmit\_skb(sk, buff, 1, sk->sk\_allocation)

icsk->icsk\_af\_ops->queue\_xmit(sk, skb,

&inet->cork.fl)

ip\_queue\_xmit(struct sock \*sk, struct sk\_buff

\*skb, struct flowi \*fl)

inet\_wait\_for\_connect(sk, timeo, writebias)

prepare\_to\_wait()

finish\_wait()

tcp\_v4\_init\_sock(struct sock \*sk)

struct inet\_connection\_sock \*icsk = inet\_csk(sk)

icsk->icsk\_af\_ops = &ipv4\_specific

inet\_create(struct net \*net, struct socket \*sock, int protocol, int kern)

struct inet\_protosw \*answer

struct sock \*sk

list\_for\_each\_entry\_rcu(answer, &inetsw[sock->type], list) {

answer =

}

sock->ops = answer->ops //for ipv4 it's may be inet\_stream\_ops

answer\_prot = answer->prot

sk = sk\_alloc(net, PF\_INET, GFP\_KERNEL, answer\_prot)

struct sock \*sk

sk->sk\_prot = sk->sk\_prot\_creator = prot

sk->sk\_prot->init(sk)

tcp\_v4\_init\_sock(struct sock \*sk)

icsk->icsk\_af\_ops = &ipv4\_specific

inet\_init()

proto\_register(&tcp\_prot, 1)

sock\_register(&inet\_family\_ops)

for (q = inetsw\_array; q < &inetsw\_array[INETSW\_ARRAY\_LEN];

++q)

inet\_register\_protosw(q)

sock\_register(struct net\_proto\_family \*ops)

rcu\_dereference\_protected(net\_families[ops->family], lockdep\_is\_held(&net\_family\_lock))

rcu\_assign\_pointer(net\_families[ops->family], ops)

SYSCALL\_DEFINE3(socket, ...)

sock\_create(family, type, protocol, &sock)

\_\_sock\_create()

sock->type = type

pf = rcu\_dereference(net\_families[family])

err = pf->create(net, sock, protocol, kern)

inet\_create() // ipv4

sock\_map\_fd()

sock\_map\_fd(sock, flags & (O\_CLOEXEC | O\_NONBLOCK))

SYSCALL\_DEFINE3(bind, ...)

sock = sockfd\_lookup\_light(fd, &err, &fput\_needed)

move\_addr\_to\_kernel()

sock->ops->bind(sock, (struct sockaddr \*)&address, addrlen)

inet\_bind() // for ipv4

SYSCALL\_DEFINE3(connect, ...)

sock->ops->connect(sock, (struct sockaddr \*)&address, addrlen, sock->file->f\_flags)

inet\_stream\_connect() // for ipv4

// TCP --> IP

ip\_queue\_xmit()

ip\_local\_out(skb)

ip\_local\_out\_sk(skb->sk, skb)

\_\_ip\_local\_out(skb)

nf\_hook(NFPROTO\_IPV4, NF\_INET\_LOCAL\_OUT,

skb, NULL, skb\_dst(skb)->dev, dst\_output)

nf\_hook\_thresh(pf, hook, skb, indev, outdev, okfn,

INT\_MIN)

nf\_hook\_slow(pf, hook, skb, indev, outdev, okfn,

thresh)

dst\_output\_sk(sk, skb)

skb\_dst(skb)->output(sk, skb)

ip\_output() // ???

ip\_output(struct sock \*sk, struct sk\_buff \*skb)

ip\_finish\_output(struct sk\_buff \*skb)

ip\_finish\_output\_gso(skb)

ip\_finish\_output2()

ip\_finish\_output2(skb)

dst\_neigh\_output(dst, neigh, skb)

neigh\_hh\_output(hh, skb)

dev\_queue\_xmit(skb)

n->output(n, skb)

ip\_build\_and\_send\_pkt()

ip\_send\_unicast\_reply()

// UDP 🡪 IP

ip\_append\_data()

ip\_ufo\_append\_data()

ip\_push\_pending\_frames()

// IP --> device

dev\_queue\_xmit(struct sk\_buff \*skb)

\_\_dev\_queue\_xmit(skb, NULL)

dev\_hard\_start\_xmit(skb, dev, txq, &rc)

xmit\_one(skb, dev, txq, next != NULL)

netdev\_start\_xmit(skb, dev, txq, more)

\_\_netdev\_start\_xmit(ops, skb, dev, more)

ops->ndo\_start\_xmit(skb, dev)

wlanHardStartXmit() // for wlan

# U-boot

## 数据结构

### gd\_t

//全局数据区的结构

typede struct global\_data {

bd\_t \*bd //与开发板相关的数据结构

unsigned long flags

unsigned long baudrate

unsigned long have\_console

unsigned long reloc\_off

unsigned long env\_addr

unsigned long env\_valid

unsigned long fb\_base

void \*\*jt

} gd\_t

### bd\_t

//与开发板相关的数据结构

typedef struct bd\_info{

int bi\_baudrate //串口波特率

unsigned long bi\_ip\_addr //IP地址

unsigned char bi\_enetaddr[6] //MAC地址

struct environment\_s \*bi\_env //环境变量开始的地址

ulong bi\_arch\_number //开发板的机器码

ulong bi\_boot\_params //内核参数的起始地址

struct{ //RAM配置信息

ulong start;

ulong size;

} bi\_dram[CONFIG\_NR\_DRAM\_BANKS]

#ifdef CONFIG\_HAS\_ETH1

unsigned char bi\_enet1addr[6]

#endif

} bd\_t

//启动参数的数据结构，linux2.6之后用tag

struct param\_struct{

union {

struct {

unsigned long page\_size /\* 0 \*/

unsigned long nr\_pages /\* 4 \*/

unsigned long ramdisk\_size /\* 8 \*/

unsigned long flags /\* 12 \*/

#define FLAG\_READONLY 1

#define FLAG\_RDLOAD 4

#define FLAG\_RDPROMPT 8

unsigned long rootdev /\* 16 \*/

unsigned long video\_num\_cols /\* 20 \*/

unsigned long video\_num\_rows /\* 24 \*/

unsigned long video\_x /\* 28 \*/

unsigned long video\_y /\* 32 \*/

unsigned long memc\_control\_reg /\* 36 \*/

unsigned char sounddefault /\* 40 \*/

unsigned char adfsdrives /\* 41 \*/

unsigned char bytes\_per\_char\_h /\* 42 \*/

unsigned char bytes\_per\_char\_v /\* 43 \*/

unsigned long pages\_in\_bank[4] /\* 44 \*/

unsigned long pages\_in\_vram /\* 60 \*/

unsigned long initrd\_start /\* 64 \*/

unsigned long initrd\_size /\* 68 \*/

unsigned long rd\_start /\* 72 \*/

unsigned long system\_rev /\* 76 \*/

unsigned long system\_serial\_low /\* 80 \*/

unsigned long system\_serial\_high /\* 84 \*/

unsigned long mem\_fclk\_21285 /\* 88 \*/

}s;

char unused[256]

} u1

union {

char paths[8][128]

struct {

unsigned long magic

char n[1024 - sizeof(unsigned long)]

}s;

} u2;

char commandline[COMMAND\_LINE\_SIZE]

}

### tag

linux2.6之后用tag

struct tag\_header {

u32 size //表示tag数据结构的联合u实质存放的数据的大小

u32 tag //标记的类型

}

struct tag {

struct tag\_header hdr //表明tag的类型和大小

union {

struct tag\_core core

struct tag\_mem32 mem

struct tag\_videotext videotext

struct tag\_ramdisk ramdisk

struct tag\_initrd initrd

struct tag\_serialnr serialnr

struct tag\_revision revision

struct tag\_videolfb videolfb

struct tag\_cmdline cmdline

struct tag\_acorn acorn

struct tag\_memclk memclk

} u

}

struct tagtable {

u32 tag

int (\*parse)(const struct tag \*)

}

#define \_\_tag \_\_attribute\_\_((unused, \_\_section\_\_(".taglist")))

#define \_\_tagtable(tag, fn) \

static struct tagtable \_\_tagtable\_##fn \_\_tag = { tag, fn }

Linux内核将tagtable也组成了一个静态的链表放入.taglist.init节中，

这是通过\_\_tagtable宏来实现的。tagtable 列表编译连接后被存放

在.taglist.init中。这个静态的链表必须以tag\_header.tag = ATAG\_CORE

开始，并以tag\_header.tag = ATAG\_NONE结束

U\_BOOT\_CMD(

bootm, CFG\_MAXARGS, 1, do\_bootm,…

)

## 启动流程

//board/smdk2410/u-boot.lds 连接器脚本

OUTPUT\_ARCH(arm)

ENTRY(\_start)

SECTIONS

{

. = 0x00000000 //起始地址

. = ALIGN(4) //4字节对齐

.text : //test指代码段

{

cpu/arm920t/start.o (.text) //表示吧start.S编译放在最开始

\*(.text)

}

。。。。。。

. = ALIGN(4);

\_\_bss\_start = .; //bss表示归零段

.bss : { \*(.bss) }

\_end = .;

}

//cpu/arm920t/start.S

\_start: b reset

……

bl lowlevel\_init

//board/smdk2410/lowlevel\_init.S

lowlevel\_init:

ldr r0, =SMRDATA

ldr r1, \_TEXT\_BASE

……

ldr pc, \_start\_armboot

\_start\_armboot: .word start\_armboot

### start\_armboot

//lib\_arm/board.c

start\_armboot()

for (init\_fnc\_ptr = init\_sequence; \*init\_fnc\_ptr; ++init\_fnc\_ptr)

board\_init()

/\* arch number of SMDK2410-Board \*/

gd->bd->bi\_arch\_number = MACH\_TYPE\_SMDK2410

/\* adress of boot parameters \*/

gd->bd->bi\_boot\_params = 0x30000100

dram\_init()

for (;;) {

main\_loop () //查询命令和执行命令

}

//include/command.h uboot命令的格式

#define U\_BOOT\_CMD(name,maxargs,rep,cmd,usage,help) \

cmd\_tbl\_t \_\_u\_boot\_cmd\_##name Struct\_Section = {#name, maxargs, rep,

cmd, usage, help}

name：命令的名字

maxargs：最大的参数个数

rep：命令是否可重复，重复指在执行完一次后敲回车可再次执行

cmd：对应的函数指针

usage：简短的使用说明

help：详细的使用说明

//common/cmd\_bootm.c

U\_BOOT\_CMD(

bootm, CFG\_MAXARGS, 1, do\_bootm,

"bootm - boot application image from memory\n",

"[addr [arg ...]]\n - boot application image stored in memory\n"

"\tpassing arguments 'arg ...'; when booting a Linux kernel,\n"

"\t'arg' can be the address of an initrd image\n"

)

### do\_bootm

// U\_BOOT\_CMD，bootm命令执行函数

do\_bootm() //cmd\_bootm.c

do\_bootm\_linux() //启动linux内核

//启动linux内核的函数的函数指针

void (\*theKernel)(int zero, int arch, uint params);

image\_header\_t \*hdr = &header

bd\_t \*bd = gd->bd

//指向kernel的启动地址

theKernel = (void (\*)(int, int, uint))ntohl(hdr->ih\_ep)

setup\_start\_tag (bd) //设置ATAG\_CORE标志

// static struct tag \*params

params = (struct tag \*) bd->bi\_boot\_params

params->hdr.tag = ATAG\_CORE

params = tag\_next (params)

setup\_memory\_tags (bd) //设置内存标记

setup\_commandline\_tag (bd, commandline) //设置命令行标记

setup\_end\_tag(bd) //设置ATAG\_NONE标志

params->hdr.tag = ATAG\_NONE

//函数最后执行，启动内核并传递参数

//该函数调用则会将0放入r0，机器码machid

放入r1，内核参数地址bd->bi\_boot\_params放入r2

theKernel (0, bd->bi\_arch\_number, bd->bi\_boot\_params)

## 传递参数

简单的讲，uboot利用函数指针及传参规范，它将

R0: 0x0

R1: 机器号，机器类型ID，开发板型号

R2: 参数地址，tag列表地址

三个参数传递给内核。

其中，R2寄存器传递的是一个指针，这个指针指向一个TAG区域

U-boot向内核传递参数的具体实现过程

### gd

1. 在include/asm-arm/global\_data.h中声名一个gd全局指针变量宏定义，并指定存放在r8寄存器中，在后面要用到gd全局指针变量时，只须要在文件开头引用这个宏就可以了

#define DECLARE\_GLOBAL\_DATA\_PTR register volatile

gd\_t \*gd asm ("r8")

### start\_armboot

1. 在start\_armboot(lib\_arm/board.c)主函数中计算全局数据结构的地址并赋值给指针gd，并对struct tag数据结构里参数赋值

gd = (gd\_t\*)(\_armboot\_start - CFG\_MALLOC\_LEN - sizeof(gd\_t))

for (init\_fnc\_ptr = init\_sequence; \*init\_fnc\_ptr; ++init\_fnc\_ptr) {

if ((\*init\_fnc\_ptr)() != 0) {

hang ();

}

}

board\_init()

/\* arch number of SMDK2410-Board \*/

gd->bd->bi\_arch\_number = MACH\_TYPE\_SMDK2410

/\* adress of boot parameters \*/

gd->bd->bi\_boot\_params = 0x30000100

dram\_init() //对struct tag数据结构里内存参数赋值

gd->bd->bi\_dram[0].start = PHYS\_SDRAM\_1;

gd->bd->bi\_dram[0].size = PHYS\_SDRAM\_1\_SIZE

c、传递到内核的参数列表的构建

./common/cmd\_bootm.c文件中，bootm命令对应的do\_bootm函数，当分析uImage中信息发现OS是Linux时，调用./lib\_arm/armlinux.c文件中的do\_bootm\_linux函数来启动Linux kernel

### do\_bootm\_linux

do\_bootm\_linux()

bd\_t \*bd = gd->bd

setup\_start\_tag (bd) //设置ATAG\_CORE标志

//内核的参数的开始地址static struct tag \*params,

params = (struct tag \*) bd->bi\_boot\_params

params->hdr.tag = ATAG\_CORE

…

params = tag\_next (params)

setup\_memory\_tags (bd) //设置内存标记

setup\_commandline\_tag (bd, commandline) //设置命令行标记

setup\_end\_tag (bd) //设置ATAG\_NONE标志

params->hdr.tag = ATAG\_NONE

d、最后do\_bootm\_linux函数调用theKernel (0, machid, bd->bi\_boot\_params)去启动内核并传递参数，可以看见r1是machid，r2是bi\_boot\_params参数的地址

theKernel (0, bd->bi\_arch\_number, bd->bi\_boot\_params)

# 内核启动和关机

## 宏定义

PHYS\_OFFSET RAM起始物理地址

#define PHYS\_OFFSET PLAT\_PHYS\_OFFSET

#define PLAT\_PHYS\_OFFSET UL(CONFIG\_PHYS\_OFFSET)

PAGE\_OFFSET RAM起始虚拟地址, 值为0xC0000000，为进程内

核空间起始虚拟地址

//arch/arm/include/asm/memory.h

#define PAGE\_OFFSET UL(CONFIG\_PAGE\_OFFSET)

TEXT\_OFFSET kernel相对于RAM起始地址的偏移（虚拟地址）

值为0x00008000

/arch/arm/Makefile

textofs-y := 0x00008000

TEXT\_OFFSET := $(textofs-y)

KERNEL\_RAM\_VADDR = (PAGE\_OFFSET + TEXT\_OFFSET) kernel

加载地址(虚拟地址)，值为0xC0008000

KERNEL\_RAM\_PADDR 内核在RAM中的物理地址

PG\_DIR\_SIZE kernel用的一级页表大小，一般为16KB，放在kernel之前

swapper\_pg\_dir = (KERNEL\_RAM\_VADDR - PG\_DIR\_SIZE) kernel用的页表基址地址(虚拟地址)，值为0xC0004000

## 基本流程

//arch/arm/kernel/head.S

// PAGE\_OFFSET: UL(CONFIG\_PAGE\_OFFSET)，内核镜像起始虚拟

//地址

// TEXT\_OFFSET: arch/arm/Makefile textofs-y = 0x00008000

// KERNEL\_RAM\_VADDR为kernel加载运行地址

#define KERNEL\_RAM\_VADDR (PAGE\_OFFSET + TEXT\_OFFSET)

// swapper\_pg\_dir 初始页表虚拟地址，大小为16K

.globl swapper\_pg\_dir

.equ swapper\_pg\_dir, KERNEL\_RAM\_VADDR - PG\_DIR\_SIZE

ENTRY(stext)

safe\_svcmode\_maskall r9

// \_\_LINUX\_ARM\_ARCH\_\_ = 4, r9没有使用

setmode PSR\_F\_BIT | PSR\_I\_BIT | SVC\_MODE, r9

// cpsr\_c代表的是cpsr寄存器的低8位，也就是控制位

//禁止IRQ, FIQ中断，设置SVC模式

msr cpsr\_c, #( PSR\_F\_BIT | PSR\_I\_BIT | SVC\_MODE)

msr cpsr\_c, #PSR\_F\_BIT | PSR\_I\_BIT | SVC\_MODE

mrc p15, 0, r9, c0, c0 //@r9 = cpuid

@从协处理器cp15的寄存器c0获得处理器信息cpuid（存入r9）与内核支持的处理器比较，确定是否支持当前CPU，输入r9=cupid，输出r5=procinfo

bl \_\_lookup\_processor\_type @ r5=procinfo r9=cupid

// r3为\_\_lookup\_processor\_type\_data标号处物理的地址

// 注：mmu enable之前的代码的虚拟地址和物理地址之间可

// 能有一个固定的差值，也可能相等

adr r3, \_\_lookup\_processor\_type\_data

ldmia r3, {r4 - r6}

sub r3, r3, r4 @ get offset between virt&phys

add r5, r5, r3 @ convert virt addresses to

add r6, r6, r3 @ physical address space

…

movs r10, r5 //r10 = procinfo

adr r3, 2f

ldmia r3, {r4, r8}

sub r4, r3, r4

add r8, r8, r4 // r8 = PHYS\_OFFSET RAM启始物理地址

bl \_\_vet\_atags // r2 = atags or dtb

bl \_\_fixup\_pv\_table

bl \_\_create\_page\_tables

pgtbl r4, r8 // r4 = physical page table base address

Clear the swapper page table to 0s

ldr r7, [r10, #PROCINFO\_MM\_MMUFLAGS] // r7 = mmu\_flag

…

ldr r13, =\_\_mmap\_switched

b \_\_enable\_mmu

b \_\_turn\_mmu\_on

mov r3, r13

ret r3 // r3 = \_\_mmap\_switched

b start\_kernel

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

//不再使用

@输出r5=machinfo，确定内核是否支持当前开发板

bl \_\_lookup\_machine\_type @ r5=machinfo

@和r1中存放的machine ID比较

teq r3, r1 @ matches loader number?

……

ldr r13, \_\_switch\_data //进入arch/arm/kernel/head-common.S

……

b start\_kernel //进入init/main.c

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

### start\_kernel

start\_kernel() //init/main.c

page\_address\_init()

for (i = 0; i < ARRAY\_SIZE(page\_address\_htable); i++)

INIT\_LIST\_HEAD(&page\_address\_htable[i].lh)

setup\_arch() //arch/arm/kernel/setup.c

mdesc = setup\_machine\_fdt(\_\_atags\_pointer) //or

setup\_machine\_tags(\_\_atags\_pointer, \_\_machine\_arch\_type)

for\_each\_machine\_desc(p)

tags = phys\_to\_virt(mdesc->boot\_params) //tag列表首地址

save\_atags(tags);

parse\_tags //解析每个tag

early\_paging\_init(mdesc)

mdesc ->init\_meminfo()

setup\_dma\_zone(mdesc)

sanity\_check\_meminfo()

// memblock初始化

arm\_memblock\_init(mdesc)

// 负责建立只能用于内核的页表

paging\_init() //重新初始化页表，arch/arm/mm/mmu.c

devicemaps\_init(mdesc)

mdesc->map\_io() // 即smdk2440\_map\_io()

early\_alloc()

bootmem\_init()

zone\_sizes\_init()

free\_area\_init\_node()

free\_area\_init\_core()

init\_currently\_empty\_zone()

zone\_init\_free\_lists()

memmap\_init()

memmap\_init\_zone()

page\_alloc\_init()

console\_init()

rest\_init()

kernel\_thread(kernel\_init) //创建了一个内核线程

kernel\_init()

kernel\_init\_freeable()

do\_basic\_setup()

do\_initcalls() //加载驱动

async\_synchronize\_full()

free\_initmem()

/\*init\_post()\*/

run\_init\_process(execute\_command) //可以指定init

run\_init\_process("/sbin/init") //启动init应用程序，init

是由内核启动的用户进程，常见的init有sysvinit、

upstart、systemd三种（busybox init也是一种）

kernel\_execve()

do\_execve()

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

kthreadd()

for (;;) {

}

schedule()

cpu\_idle()

while (1){

}

## Async and Workqueue

async\_schedule(async\_func\_t func, void \*data)

\_\_async\_schedule(func, data, &async\_dfl\_domain)

entry->func = func;

entry->data = data;

entry->domain = domain

queue\_work(system\_unbound\_wq, &entry->work)

queue\_work\_on()

\_\_queue\_work()

## 数据结构

### proc\_info\_list

//表示内核支持的CPU，都被定义在.proc.info.init段中

proc\_info\_list{ //include/asm-arm/procinfo.h

cpu\_val

cpu\_mask

}

如在arch/arm/mm/proc-arm920.S中line-451定义了arm920

.section ".proc.info.init", #alloc, #execinstr

.type \_\_arm920\_proc\_info,#object

\_\_arm920\_proc\_info:

.long 0x41009200

.long 0xff00fff0

### machine\_desc

//表示内核支持的开发板，都被定义在.arch.info.init段中

machine\_desc{

nr //体系类型ID

phys\_io

io\_pg\_offst

\*name //体系名字

boot\_params //boot参数，tag列表地址

void (\*map\_io)(void) //io映射函数

void (\*init\_irq)(void) //中断初始化函数

}

如arch/arm/match-s3c2440/match-smdk2440.c中line-192定义了

smdk2440

struct machine\_desc \_\_mach\_desc\_ S3C2440{

.nr = MACH\_TYPE\_ S3C2440, //开发板类型号

.name = "SMDK2440",

.phys\_io = S3C2410\_PA\_UART,

.io\_pg\_offst = (((u32)S3C24XX\_VA\_UART) >> 18) & 0xfffc,

.boot\_params = S3C2410\_SDRAM\_PA + 0x100,

.init\_irq = s3c24xx\_init\_irq,

.map\_io = smdk2440\_map\_io,

.init\_machine = smdk2440\_machine\_init

.timer = &s3c24xx\_timer,

}

## busybox init

//busybox1.7.0/init/init.c

init\_main() //init程序

parse\_inittab()

//打开配置文件，文档examples/inittab，位于/etc/inittab

file = fopen(INITTAB, "r") // #define INITTAB "/etc/inittab"

new\_init\_action(a->action, command, id)

run\_actions(SYSINIT)

waitfor(a, 0);

run(a)

waitpid(runpid, &status, 0)

delete\_init\_action(a);

run\_actions(WAIT)

run\_actions(ONCE)

run(a)

delete\_init\_action(a)

while(1)

run\_actions(RESPAWN)

run\_actions(ASKFIRST)

if (a->pid == 0) {

a->pid = run(a)

}

wpid = wait(NULL)

while (wpid > 0) {

for{

}

wpid = waitpid(-1, NULL, WNOHANG)

}

new\_init\_action(ASKFIRST, bb\_default\_login\_shell, VC\_2)

new\_init\_action(ASKFIRST, "-/bin/sh", "/dev/tty2")

创建init\_action结构，填充

放入init\_action\_list

//init配置文件 /etc/inittab

inittab格式：<id>:<runlevels>:<action>:<process>

id->/dev/id 用做终端，stdin,stdout,stderr，表示该进程使用的控制台，

省略则与init使用一样的控制台

runlevels->进程的运行级别，busybox的init不支持，可忽略

action –>何时执行，表示init如何控制该进程，可取值如下：

sysinit 系统启动后最先执行

respawn系统启动完once进程后

askfirst系统启动完respawn进程后

wait系统执行完sysinit进程后

once系统执行完wait进程后

restart当Busybox配置了

CONFIG\_FEATURE\_USE\_INNITTAB,

且init进程接收到了SIGHUP信号

ctrlaltdel 按下Ctrl+Alt\_Del组合键时

shutdown系统关机时

process –> 要执行的应用程序或脚本 前面加“-”说明是可交互的如

-/bin/sh

默认的inittab配置文件

#it has the following default behavior:

# ::sysinit:/etc/init.d/rcS //默认的启动脚本

# ::askfirst:/bin/sh

# ::ctrlaltdel:/sbin/reboot

# ::shutdown:/sbin/swapoff -a

# ::shutdown:/bin/umount -a -r

# ::restart:/sbin/init

系统启动最小根文件系统：

1./dev/console

2.init->busybox

3./etc/inittab配置文件

4.配置文件中指定的应用程序

5.C库

系统的配置文件

/etc/init.d/rcS文件

## 构建根文件系统

yaffs

## 机器类型

//定义了开发板相关信息的结构，所有的机器类型的信息都放在

.arch.info.init段中，match-smdk2440.c

struct machine\_desc \_\_mach\_desc\_ S3C2440{

.nr = MACH\_TYPE\_ S3C2440, //开发板类型号

.name = "SMDK2440",

.phys\_io = S3C2410\_PA\_UART,

.io\_pg\_offst = (((u32)S3C24XX\_VA\_UART) >> 18) & 0xfffc,

.boot\_params = S3C2410\_SDRAM\_PA + 0x100,

.init\_irq = s3c24xx\_init\_irq,

.map\_io = smdk2440\_map\_io,

.init\_machine = smdk2440\_machine\_init

.timer = &s3c24xx\_timer,

}

smdk2440\_machine\_init() //注册设备

platform\_add\_devices() //添加s3c2440设备

smdk\_machine\_init()

platform\_add\_devices() //添加smdk公共设备

start\_kernel()

setup\_arch()

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

lookup\_machine\_type() //汇编语言实现

\_\_lookup\_machine\_type //查找匹配的机器类型

init\_arch\_irq = mdesc->init\_irq

system\_timer = mdesc->timer

init\_machine = mdesc->init\_machine //全局初始化函数

customize\_machine()

init\_machine()

arch\_initcall(customize\_machine)

\_\_define\_initcall("3",fn,3) //如果module没有定义

module\_init(x) \_\_define\_initcall("6",fn,6)

## 内核关机

## Reboot系统调用

//reboot.c

SYSCALL\_DEFINE4(reboot, int, magic1, int, magic2, unsigned int, cmd,

void \_\_user \*, arg)

kernel\_restart(NULL)

kernel\_halt()

kernel\_power\_off()

kernel\_shutdown\_prepare(SYSTEM\_POWER\_OFF)

//向关心reboot事件的进程，发送SYS\_RESTART、

SYS\_HALT或者SYS\_POWER\_OFF事件, 将系统状态设置为相应的状态（SYS\_RESTART、SYS\_HALT或

SYS\_POWER\_OFF）

blocking\_notifier\_call\_chain()

//禁止User mode helper

usermodehelper\_disable()

//关闭所有的设备

device\_shutdown()

dev->bus->shutdown(dev)

//或者

dev->driver->shutdown(dev)

migrate\_to\_reboot\_cpu() //将当前的进程（task）移到一个CPU

上

syscore\_shutdown() //将系统核心器件关闭（例如中断等）

machine\_power\_off()

# Android

## binder

binder\_init()

binder\_deferred\_workqueue

= create\_singlethread\_workqueue("binder")

debugfs\_create\_dir("binder", NULL)

init\_binder\_device(“binder”)

binder\_fops

misc\_register()

struct file\_operations binder\_fops {

.owner = THIS\_MODULE,

.poll = binder\_poll,

.unlocked\_ioctl = binder\_ioctl,

.compat\_ioctl = binder\_ioctl,

.mmap = binder\_mmap,

.open = binder\_open,

.flush = binder\_flush,

.release = binder\_release,

}

binder\_open(struct inode \*nodp, struct file \*filp)

binder\_proc \*proc

filp->private\_data = proc

binder\_ioctl()

binder\_mmap(struct file \*filp, struct vm\_area\_struct \*vma)

struct binder\_proc \*proc = filp->private\_data

vma->vm\_private\_data = proc

binder\_alloc\_mmap\_handler(alloc = &proc->alloc, vma)

struct vm\_struct \*area

area = get\_vm\_area(vma->vm\_end - vma->vm\_start,

VM\_IOREMAP

(alloc)->buffer = area->addr

alloc->user\_buffer\_offset =vma->vm\_start –

(uintptr\_t)alloc->buffer

alloc->pages = kzalloc(sizeof(alloc->pages[0]) \*

((vma->vm\_end - vma->vm\_start) / PAGE\_SIZE),

GFP\_KERNEL)

alloc->buffer\_size = vma->vm\_end - vma->vm\_start

proc->files = get\_files\_struct(current)