ARM Linux内核

提纲

- 1. ARM系统结构简介
- 2. ARM-Linux内存管理
- 3. ARM-Linux进程管理和调度
- 4. ARM-Linux 的中断响应和处理
- 5. ARM-Linux系统调用

1. ARM系统结构简介

- ARM有7种运行状态:
  - 用户状态（User）
  - 中断状态（IRQ, Imterrupt Request） (0x18)
  - 快中断状态（FIQ,Fast Imterrupt Request）(0x1c)
  - 监管状态（Supervisor）
  - 终止状态（Abort）
  - 无定义状态（Undefined）
  - 系统状态（System）

- ARM系统结构中各个寄存器的使用方式

寄存器	使用方式
程序计数器pc（r15）	由所有运行状态共用
通用寄存器r0-r7	由所有运行状态共用
通用寄存器r8-r12	除快中断以外所有其他运行状态共用（快中断状态有自己专用的r8-r12）
当前程序状态寄存器CPSR	由所有运行状态共用
保存程序状态寄存器SPSR	除用户状态以外的6种运行状态，各有自己的保存程序状态寄存器SPSR
堆栈指针sp（r13）和链接寄存器lr（r14）	7种运行状态各有自己的sp和lr

2 ARM-Linux内存管理

- 存储管理是一个很大的范畴
  - 地址映射、空间分配、保护机制
- 存储管理机制的实现和具体的CPU以及MMU的结构关系非常紧密
- 操作系统内核的复杂性相当程度上来自内存管理，对整个系统的结构有着根本性的深远影响

2.1内存管理和MMU

- MMU，也就是“内存管理单元”，其主要作用是两个方面：
  - 地址映射
  - 对地址访问的保护和限制
- MMU就是提供一组寄存器
- MMU可以做在芯片中，也可以作为协处理器

2.2 冯·诺依曼结构和哈佛结构

- 冯·诺依曼结构：程序只是一种数据，对程序也可以像对数据一样加以处理，并且可以和数据存储在同一个存储器中
- 嵌入式系统中往往采用程序和数据两个存储器、两条总线的系统结构，称为“哈佛结构”

2.3 ARM存储管理机制

- ARM系统结构中，地址映射可以是单层的按“段（section）”映射，也可以是二层的页面映射
- 采用单层的段映射的时候，内存中有个“段映射表”，当CPU访问内存的时候：
  - 其32位虚地址的高12位用作访问段映射表的下标，从表中找到相应的表项
  - 每个表项提供一个12位的物理段地址，以及对这个段的访问许可标志，将这12位物理段地址和虚拟地址中的低20位拼接在一起，就得到了32位的物理地址

- 如果采用页面映射，“段映射表”就成了“首层页面映射表”，映射的过程如下(以页面大小＝4KB为例)：
  - 以32位虚地址的高12位（bit20-bit31）作为访问首层映射表的下标，从表中找到相应的表项，每个表项指向一个二层映射表。
  - 以虚拟地址中的次8位（bit12-bit19）作为访问所得二层映射表的下标，进一步从相应表项中取得20位的物理页面地址。
  - 最后，将20位的物理页面地址和虚拟地址中的最低12位拼接在一起，就得到了32位的物理地址。

- 凡是支持虚存的CPU必须为有关的映射表提供高速缓存，使地址映射的过程在不访问内存的前提下完成，用于这个目的高速缓存称为TLB
- 高速缓存 (I/O的特殊性)
- ARM系统结构中配备了两个地址映射TLB和两个高速缓存

- ARM处理器中，MMU是作为协处理器CP15的一部分实现的
- MMU相关的最主要的寄存器有三个：
  - 控制寄存器，控制MMU的开关、高速缓存的开关、写缓冲区的开关等
  - 地址转换表基地址寄存器
  - 域访问控制寄存器

- 控制寄存器中有S位（表示System）和R位（表示ROM），用于决定了CPU在当前运行状态下对目标段或者页面的访问权限，如果段或者页面映射表项中的2位的“访问权限” AP为00，那么S位和R位所起的作用如表

S	R	CPU运行在特权状态	CPU运行在用户状态
0	0	不能访问	不能访问
1	0	只读	不能访问
0	1	只读	只读
1	1	不确定	不确定

- 如果AP为01，则和S位R位无关，特权状态可读可写，用户状态不能访问。
- 如果AP为10，则和S位R位无关，特权状态可读可写，用户状态只读。
- 如果AP为11，则和S位R位无关，特权状态、用户状态都可读可写。

2.4 ARM-Linux存储机制的建立

- ARM-Linux内核也将这4GB虚拟地址空间分为两个部分，系统空间和用户空间
- ARM将I/O也放在内存地址空间中，所以系统空间的一部分虚拟地址不是映射到物理内存，而是映射到一些I/O设备的地址

```
#define TASK_SIZE       (0xc0000000UL)
#define PAGE_OFFSET     (0xc0000000UL)
#define PHYS_OFFSET     (0xa0000000UL)
```

```
#define __virt_to_phys(x)       ((x) - PAGE_OFFSET + PHYS_OFFSET)
#define __phys_to_virt(x)       ((x) - PHYS_OFFSET + PAGE_OFFSET)
```

- ARM处理器上的实现和x86的既相似又有很多不同：
  - 在ARM处理器上，如果整个段（1MB，并且和1MB边界对齐）都有映射，就采用单层映射；而在x86上总是采用二层映射
  - ARM处理器上所谓的“段（section）”是固定长度的，实质上就是超大型的页面；而x86上的“段（segment）”则是不定长的
- Linux在启动初始化的时候依次调用：start_kernel()>setup_arch()>pageing_init()>memtable_init()>create_mapping()

Linux的启动

- head.S是linux运行的第一个文件。
- 内核的入口是stext,这是在arch/arm/kernel/vmlinux.lds.S中定义的
  - ENTRY(stext)
- vmlinux.lds.S是ld script文件,ENTRY(stext)表示程序的入口是在符号stext。 而符号stext是在arch/arm/kernel/head.S中定义的

启动的主线

- 确定process type
- 确定machine type
- 创建页表
- 调用平台特定的__CPU_flush函数
- 开启mmu
- 切换数据

确定processor type

- 确保kernel运行在SVC模式下,并且IRQ和FIRQ中断已经关闭。
- 通过cp15协处理器的c0寄存器来获得processor id的指令。
- 跳转到 ,在 __lookup_processor_type 中会 把存储在r5中。
- 判断r5中的process_type是否是0,如果是0,说明

__lookup_processor_type

- __lookup_processor_type函数主要是根据从CPU中获得的process id和系统中的proc_info进行匹配
  - adr r3, __lookup_processor_type_data
  - ldmia r3, {r4 - r6}
- adr指令取指获得的是基于PC的一个地址,由于此时MMU还没有打开,也可以 理解成物理

创建页表

- kernel里面的所有符号在链接时,都使用了虚拟地址值。在完成基本的初始化后,kernel代码将跳到第一个C语言函数start_kernl来执行行,在那时候,这些虚拟地址必须能够对应到它所存放在真正内存位置,否则运行会出错。为此,CPU必须开启MMU,但在开启MMU前,必须为虚拟地址到物理地址的映射建立相应的页表。在开启MMU后,kernel并不马上将PC值指向start_kernl,而是要做一些C语言运行期的设置,如堆栈等工作后才跳到start_kernel去执行。在此过程中,PC值还是物理地址,因此还需要为这段内存空间建立va = pa的内存映射关系。当然,此时建立的所有页表都会在将来

调用平台特定的__cpu_flush 函数

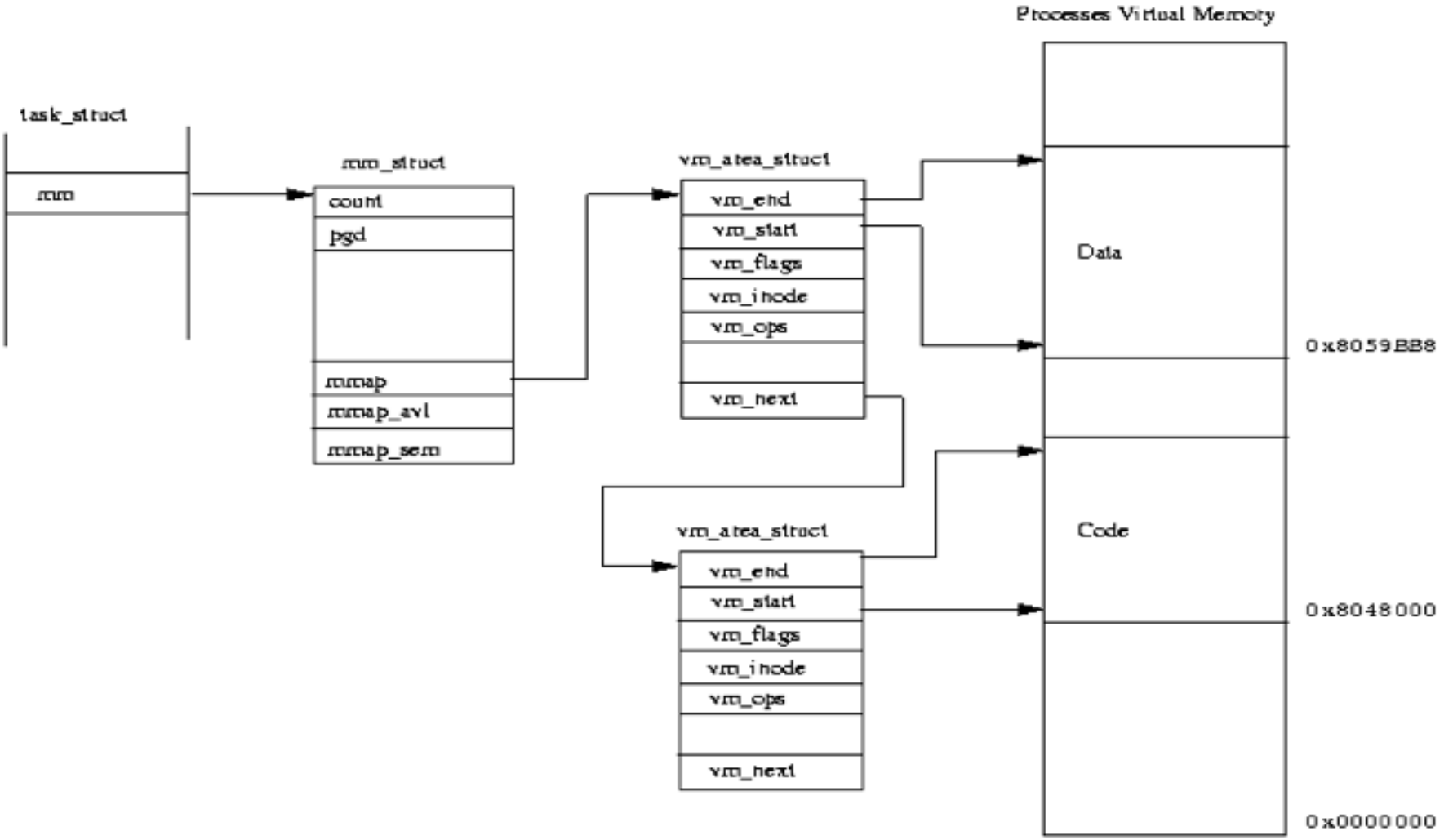
- 在我们需要在开启mmu之前,做一些必须的工作:清除ICache, 清除 DCache, 清除 Writebuffer, 清除TLB等.这些一般是通过cp15协处理器来实现的,并且是平台相关的. 这就是 __cpu_flush 需要做的工作
- 然后才能开启MMU并切换数据

2.5 ARM-Linux进程的虚存空间

- Linux虚拟内存的实现需要6种机制的支持：
  - 地址映射机制
  - 内存分配回收机制
  - 缓存和刷新机制
  - 请求页机制
  - 交换机制
  - 内存共享机制

- 系统中的每个进程都各有自己的首层映射表，这就是它的空间，没有独立的空间的就只是线程而不是进程
- Linux内核需要管理所有的虚拟内存地址，每个进程虚拟内存中的内容在其task_struct结构中指向的 vm_area_struct结构中描叙

- task_struct结构分析图：

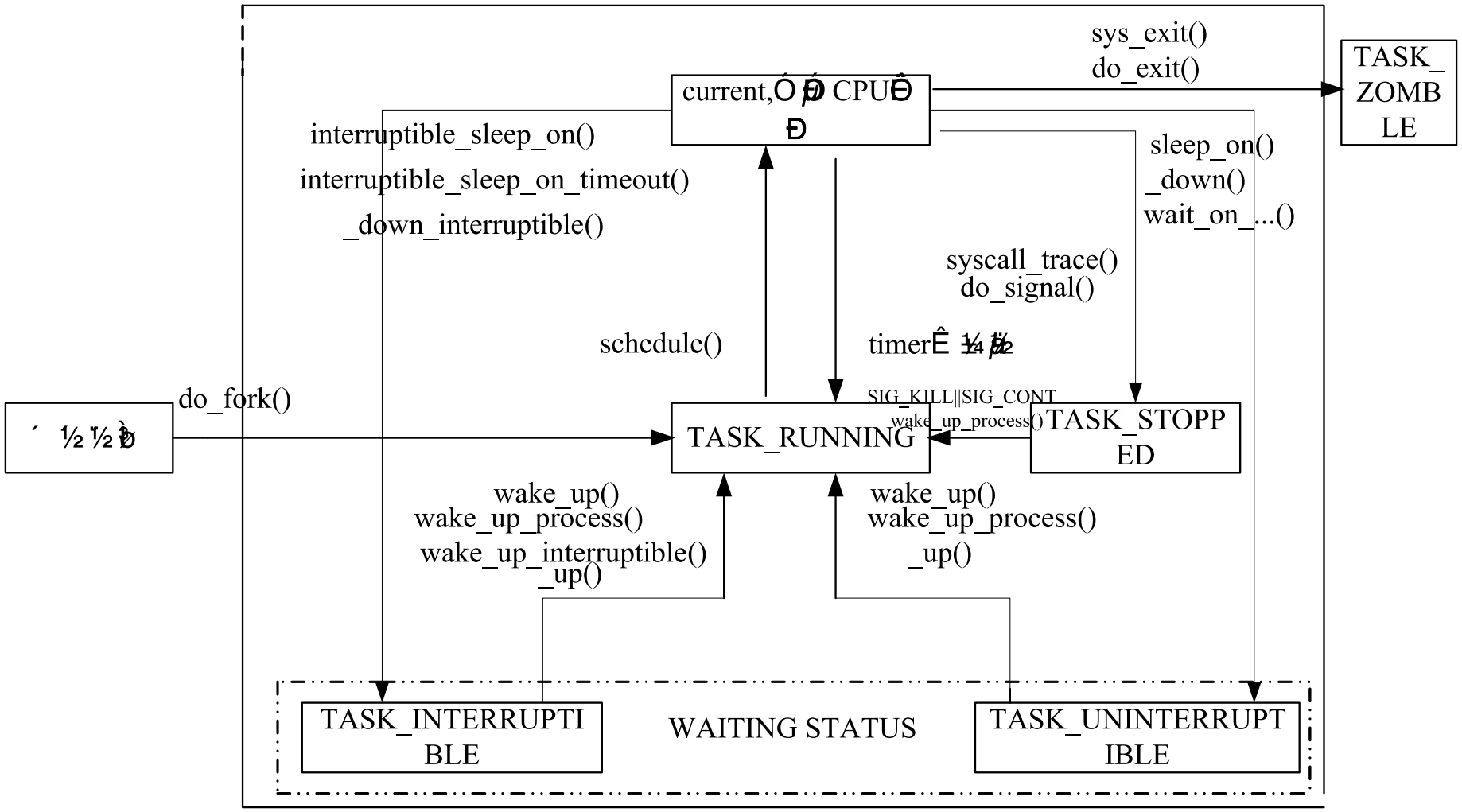


- 由于那些虚拟内存区域来源各不相同，Linux使用vm_area_struct中指向一组虚拟内存处理过程的指针来抽象此接口
- 为进程创建新的虚拟内存区域或处理页面不在物理内存中的情况下，Linux内核重复使用进程的vm_area_struct数据结构集合。采用AVL树来减少查找时间。
- 当进程请求分配虚拟内存时，Linux并不直接

3 ARM-Linux进程管理和调度

- Linux进程有5种状态，分别是：
  - TASK_RUNNING
  - TASK_INTERRUPTIBLE
  - TASK_UNINTERRUPTIBLE
  - TASK_ZOMBIE
  - TASK_STOPPED

进程间状态变换


sys_exit()
do_exit()
TASK_ZOMBLE
current,Ó Ð CPUÊ Ð
interruptible_sleep_on()
interruptible_sleep_on_timeout()
_down_interruptible()
sleep_on()
_down()
wait_on_...()
syscall_trace()
do_signal()
schedule()
timerÊ ¼ß
SIG_KILL||SIG_CONT
wake_up_process()
do_fork()
´ ½ ½ è
TASK_RUNNING
TASK_STOPPED
wake_up()
wake_up_process()
wake_up_interruptible()
_up()
wake_up()
wake_up_process()
_up()
TASK_INTERRUPTIBLE
WAITING STATUS
TASK_UNINTERRUPTIBLE

3.1 Linux进程的创建、执行和消亡

- 1. Linux进程的创建
- 系统的第一个真正的进程，init内核线程（或进程）的标志符为1
- 新进程通过克隆老进程或当前进程来创建，系统调用fork或clone可以创建新任务
- 复制完成后，Linux允许两个进程共享资源而不是复制各自的拷贝

- 2. Linux进程的执行
- 要让若干新进程按照需要处理不同的事情，就必须通过系统调用exec
- 函数sys_execve将可执行文件的名字从用户空间取入内核空间以后就调用do_execve( )执行具体的操作

- do_execve( )执行的流程：
  - 打开可执行文件,获取该文件的 file结构。
  - 获取参数区长度,将存放参数的页面清零。
  - 对linux_binprm结构的其它项作初始化
  - 通过对参数和环境个数的计算来检查是否在这方面有错误
  - 调用prepare_binprm() 对数据结构linux_binprm作进一步准备
  - 把一些参数(文件名、环境变量、文件参数)从用户空间复制到内核空间
  - 调用search_binary_handler()，搜寻目标文件的处理模块并执行

- 3. Linux进程的消亡
- 进程终止由可终止进程的系统调用通过调用do_exit（）实现
- do_exit(long code)带一个参数code，用于传递终止进程的原因

- **do_exit(long code)流程**：
  - （1）如果进程在中断服务程序中调用do_exit（），则打印提示信息　　。
  - （2）记录进程的记帐信息。
  - （3）进程标志置为PF_EXITING。
  - （4）释放定时器链表。
  - （5）释放临界区数据。
  - （6）将消息队列中和current进程有关项删除。
  - （7）释放进程的存储管理信息。
  - （8）释放进程已打开文件的信息。
  - （9）释放进程的文件系统。
  - （10）释放进程的信号响应函数指针数组等管理信息。
  - （11）释放进程的LDT。
  - （12）进程状态置为TASK_ZOMBIE。
  - （13）置上退出信息，通知所有相关进程，它要退出了。
  - （14）exec_domain结构共享计数减1，binfmt结构共享计数减1。
  - （15）重新调度，将current进程从run-queue中删除，交出CPU控制权

- 以下情况要调用do_exit()函数：
  - 具体对应的系统调用出错，不得不终止进程，如：
    - do_page_fault（）
    - sys_sigreturn（）
    - setup_frame（）
    - save_v86_state（）
  - 其他终止进程的情况，通过调用以下函数实现终止：
    - sys_exit（）
    - sys_reboot()
    - do_signal（）

- LINUX系统进程的切换包括三个层次:
  - 用户数据的保存:
    - 正文段、数据段、栈段、共享内存段
  - 寄存器数据的保存
    - PC、PSW、SP、PCBP、FP…
  - 系统层次的保存
    - proc、u、虚拟存储空间管理表格、中断处理栈

3.2 ARM-Linux进程的调度

- Linux进程调度由函数schedule()实现的，其基本流程可以概括为五步：
  - 清理当前运行中的进程
  - 选择下一个投入运行的进程
  - 设置新进程的运行环境
  - 执行进程上下文切换
  - 后期整理
- Linux调度的时机有两种：
  - 在内核应用中直接调用schedule()

4 ARM-Linux 的中断响应和处理

- 中断是一个流程，一般来说要经过三个环节：
  - 中断响应
  - 中断处理
  - 中断返回
- 中断响应是第一个环节，主要是确定中断源，在整个中断机制中起着枢纽的作用

- 使CPU在响应中断的时候能迅速的确定中断源，**且尽量减少引脚数量**，辅助手段主要有下列几种：
  - 中断源通过数据总线提供一个代表具体设备的数值，称为“中断向量”
  - 在外部提供一个“集线器”，称为“中断控制器”
  - 将中断控制器集成在CPU芯片中，但是设法“挪用”或“复制”原有的若干引线，而并不实际增加

- ARM是将中断控制器集成在CPU内部的，由外设产生的中断请求都由芯片上的中断控制器汇总成一个IRQ中断请求
- 中断控制器还向CPU提供一个中断请求寄存器和一个中断控制寄存器
- GPIO是一个通用的可编程的I/O接口，其接口寄存器中的每一位都可以分别在程序的控

- ARM Linux将中断源分为三组：
  - 第一组是针对外部中断源；
  - 第二组中是针对内部中断源，它们都来自集成在芯片内部的外围设备和控制器，比如LCD控制器、串行口、DMA控制器等等。
  - 第三组中断源使用的是一个两层结构。

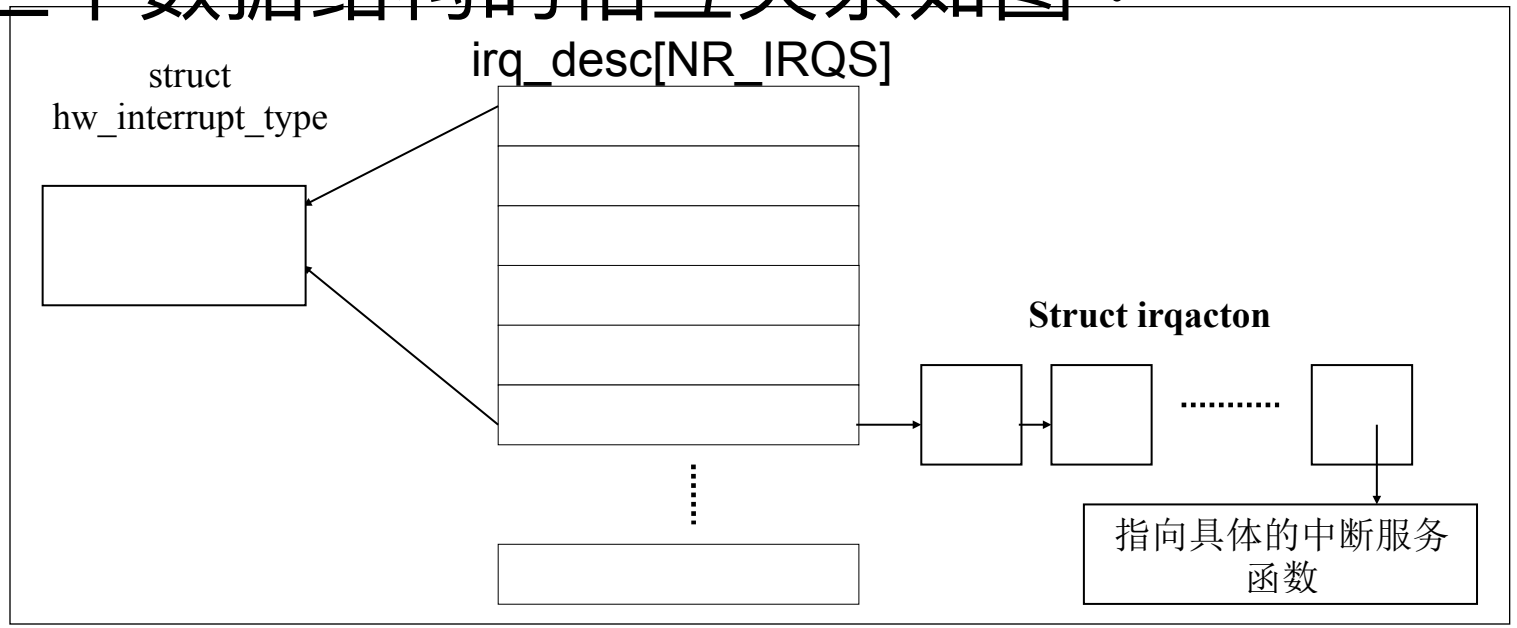
- 在Linux中，每一个中断控制器都由strcut hw_interrut_type数据结构表示：

```
struct hw_interrupt_type {
const char * typename;
unsigned int (*startup)(unsigned int irq);
void (*shutdown)(unsigned int irq);
void (*enable)(unsigned int irq);
void (*ack)(unsigned int irq);
void (*end)(unsigned int irq);
void (*set_affinity)(unsiged int irq,unsigned long mask);
};
```

- 每一个中断请求线都有一个struct irqdesc 数据结构表示：

```
typedef struct {
unsigned int status;   /* IRQ status */
hw_irq_controller *handler;
struct irqaction *action;  /*IRQ action list */
unsigned int depth;    /* nested irq disables */
spinlock_t lock;
}_cacheline_aligned irq_desc_t;
```

- 具体中断处理程序则在数据结构 struct irqaction
- 三个数据结构的相互关系如图：



- ARM Linux的中断初始化。
  - 在ARM Linux存储管理中，内核中DRAM区间的虚拟地址和物理地址是相同的。系统加电引导以后，CPU进入内核的总入口，即代码段的起点stext，CPU首先从自身读出CPU的型号以及其所在的开发板，把有关的信息保存在全局变量中；
  - 然后就转入start_kernel()函数进行初始化；
  - 接着是执行函数trap_init（）
    - 这个函数做的第一件事是将下列指令搬运到虚拟地址0处：

```
.LCvectors:     swi     SYS_ERROR0
                b       __real_stubs_start + (vector_undefinstr - __stubs_start)
                ldr     pc, __real_stubs_start + (.LCvswi - __stubs_start)
                b       __real_stubs_start + (vector_prefetch - __stubs_start)
                b       __real_stubs_start + (vector_data - __stubs_start)
                b       __real_stubs_start + (vector_addrexcptn - __stubs_start)
                b       __real_stubs_start + (vector_IRQ - __stubs_start)
                b       __real_stubs_start + (vector_FIQ - __stubs_start)
```

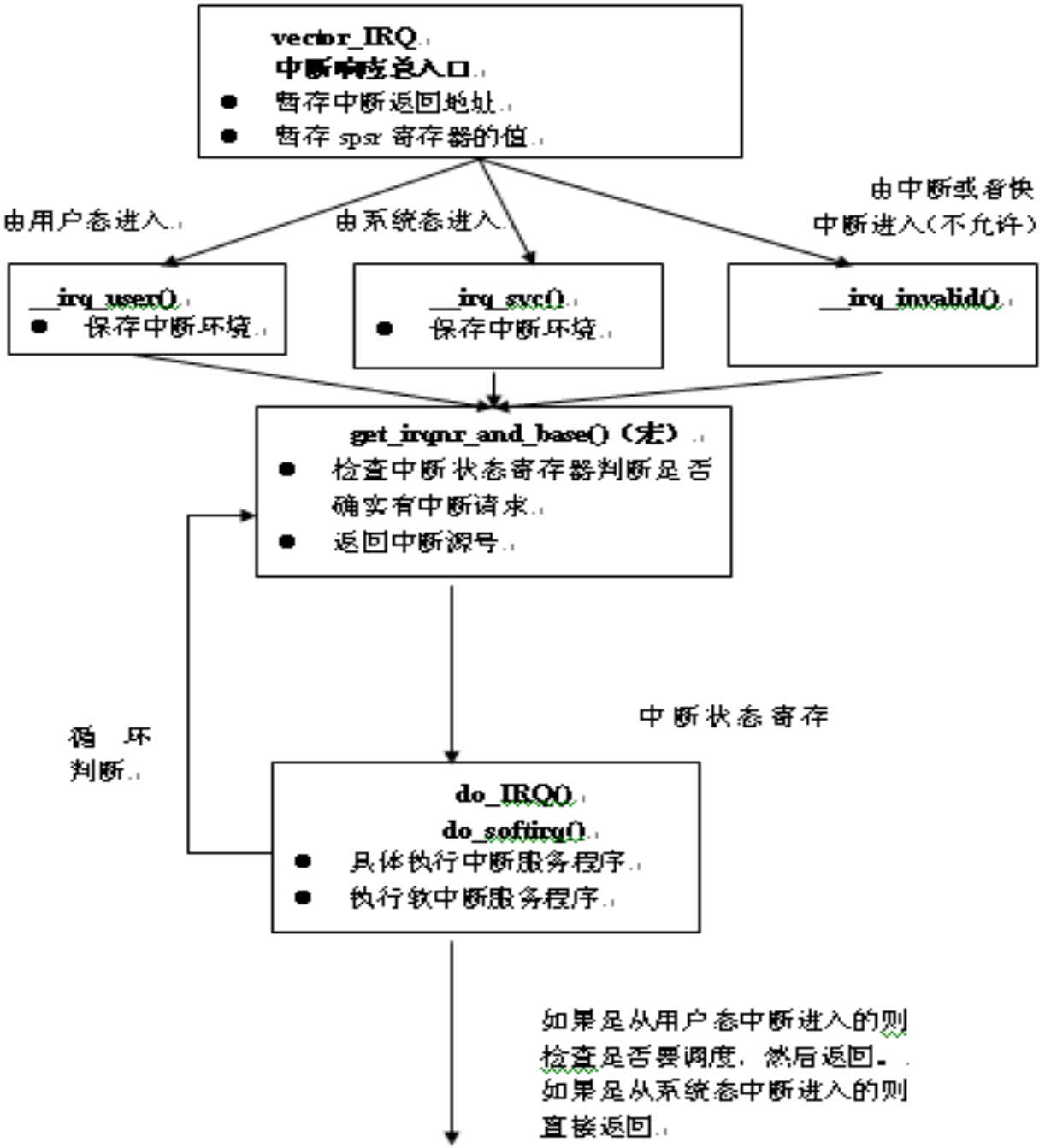
- 第二件事是搬运底层中断响应程序的代码（如下所示）到0x200处：

```
__stubs_start:
vector_IRQ:
                ...
vector_data:
...
vector_prefetch:
                ...
vector_undefinestr:
                ...
vector_FIQ:
                ...
vector_addrexcptn:
                ...
.LCvswi:        .word           vector_swi
.LCsirq:        .word           __temp_irq
.LCsund:        .word           __temp_und
.LCsabt:        .word           __temp_abt
__stubs_end:
```

– trap_init()函数执行完了以后，再执行init_IRQ()。通过函数init_IRQ()建立上面提及的3个数据结构及其相互联系的框架。

- 在进入中断响应之前，CPU自动完成下列操作：
  - 将进入中断响应前的内容装入r14_irq，即中断模式的lr，使其指向中断点。
  - 将cpsr原来的内容装入spsr_irq，即中断模式的spsr；同时改变cpsr的内容使CPU运行于中断模式，并关闭中断。
  - 将堆栈指针sp切换成中断模式的sp_irq。

- 中断流程图


vector_IRQ
中断响应总入口
暂存中断返回地址
暂存 spsr 寄存器的值
由用户态进入
由系统态进入
由中断或者快中断进入(不允许)
__irq_user()
保存中断环境
__irq_svc()
保存中断环境
__irq_invalid()
get_irqnr_and_base()（宏）
检查中断状态寄存器判断是否确实有中断请求
返回中断源号
循环判断
中断状态寄存
do_IRQ()
do_softirq()
具体执行中断服务程序
执行软中断服务程序
如果是从用户态中断进入的则检查是否要调度，然后返回。
如果是从系统态中断进入的则直接返回

5 ARM-Linux系统调用

5 ARM-Linux系统调用

- LIBC和直接调用

5 ARM-Linux系统调用

- LIBC和直接调用
- X86有INT 0x80

5 ARM-Linux系统调用

- LIBC和直接调用
- X86有INT 0x80
- arm处理器有自陷指令SWI

5 ARM-Linux系统调用

- LIBC和直接调用
- X86有INT 0x80
- arm处理器有自陷指令SWI
- cpu遇到自陷指令后，跳转到内核态

5 ARM-Linux系统调用

- LIBC和直接调用
- X86有INT 0x80
- arm处理器有自陷指令SWI
- cpu遇到自陷指令后，跳转到内核态
- 操作系统首先保存当前运行的信息，然后根据系统调用号查找相应的函数去执行

5 ARM-Linux系统调用

- LIBC和直接调用
- X86有INT 0x80
- arm处理器有自陷指令SWI
- cpu遇到自陷指令后，跳转到内核态
- 操作系统首先保存当前运行的信息，然后根据系统调用号查找相应的函数去执行
- 执行完了以后恢复原先保存的运行信息返回

创建和使用一个新的系统调用(1)

- 在 arch/arm/kernel/目录下创建一个新的文件mysyscall.c

```
void hello(void) {
                    printk("hello world\n");
}
```

- 在 arch/arm/kernel/call.S 中添加新的系统调用，新的系统调用号0x000000+326

```
                .long   SYMBOL_NAME(sys_gettid)
                .long   SYMBOL_NAME(sys_readahead)
                .long   SYMBOL_NAME(hello)
__syscall_end:
                .rept   NR_syscalls - (__syscall_end - __syscall_start) / 4
                .long   SYMBOL_NAME(sys_ni_syscall)
                .endr
```

创建和使用一个新的系统调用(2)

- 修改arch/arm/kernel/目录下的Makefile文件，在obj-y后面添加mysyscall.o

```
obj-y := arch.o compat.o dma.o $(ENTRY_OBJ) entry-common.o irq.o \
                  process.o ptrace.o semaphore.o setup.o signal.o sys_arm.o \
                time.o traps.o $(O_OBJS_$(MACHINE)) mysyscall.o
```

创建和使用一个新的系统调用(3)

- 写一个测试程序来使用新的系统调用：

```
test.h:
#define    sys_hello()            {__asm__ __volatile__ ("swi
0x900000+226\n\t")}while(0)
test.c:
#include <stdio.h>
#include  “test.h”
int main(void)
{
            printf("start hello\n");
sys_hello();
            printf("end hello\n");
}
```

创建和使用一个新的系统调用(4)

- 然后执行

```
# arm-linux-gcc test.c -o test
```

- 启动开发板，将应用程序test通过zmodem协议下载到开发板的文件系统目录下，在板子上运行test程序所得结果如下：

```
# ./test
start hello
hello world
end hello
```

注意，上面的例子是直接用汇编使用系统调用的，而不是使用*libc*，因为*test*应用程序使用的是新添加的系统调

- 思考:

  如何增加一个带参数的系统调用?