**linux-mips启动分析 （1）**

系统加电起动后，MIPS 处理器默认的程序入口是0xBFC00000，此地址在无缓存的KSEG1的地址区域内，对应的物理地址是 0x1FC00000，即CPU从0x1FC00000开始取第一条指令，这个地址在硬件上已经确定为FLASH的位置，Bootloader将 Linux 内核映像拷贝到  RAM  中某个空闲地址处，然后一般有个内存移动操作，目的地址在 arch/mips/Makefile 内指定：   
load-$(CONFIG\_MIPS\_PB1550) += 0xFFFFFFFF80100000，  
则最终bootloader定会将内核移到物理地址   0x00100000  处。  
  
上面Makefile 里指定的的 load 地址，最后会被编译系统写入到 arch/mips/kernel/vmlinux.lds 中：  
  
OUTPUT\_ARCH(mips)  
ENTRY(kernel\_entry)  
jiffies = jiffies\_64;  
SECTIONS  
{  
. = 0xFFFFFFFF80100000;  
/\* read-only \*/  
\_text = .; /\* Text and read-only data \*/  
.text : {  
    \*(.text)  
...  
  
这个文件最终会以参数 -Xlinker --script -Xlinker vmlinux.lds 的形式传给 gcc，并最终传给链接器 ld 来控制其行为。  
ld 会将 .text 节的地址链接到 0xFFFFFFFF80100000 处。  
  
关于内核 ELF 文件的入口地址(Entry point)，即 bootloader 移动完内核后，直接跳转到的地址，由ld 写入 ELF的头中，其会依次用下面的方法尝试设置入口点，当遇到成功时则停止：  
  
a. 命令行选项 -e entry  
b. 脚本中的 ENTRY(symbol)  
c. 如果有定义 start 符号，则使用start符号（symbol）  
d. 如果存在 .text 节，则使用第一个字节的地址。  
e. 地址0  
  
注意到上面的 ld script 中，用 ENTRY 宏设置了内核的 entry point 是 kernel\_entry，因此内核取得控制权后执行的第一条指令是在 kernel\_entry 处。  
＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊  
  
linux  内核启动的第一个阶段是从  /arch/mips/kernel/head.s文件开始的。  
而此处正是内核入口函数kernel\_entry(),该函数定义在 /arch/mips/kernel/head.s文件里。  
  
kernel\_entry()函数是体系结构相关的汇编语言，它首先初始化内核堆栈段，来为创建系统中的第一个进程进行准备，  
接着用一段循环将内核映像的未初始化数据段（bss段，在\_edata和\_end之间）清零，  
最后跳转到  /init/main.c 中的 start\_kernel()初始化硬件平台相关的代码。  
＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊  
  
NESTED(kernel\_entry, 16, sp)            # kernel entry point  
声明函数   kernel\_entry，函数的堆栈为 16 byte，返回地址保存在  $sp 寄存器中。  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
声明函数入口  
#define NESTED(symbol, framesize, rpc)                  \  
        .globl  symbol;                         \  
        .align  2;                              \  
        .type   symbol,@function;               \  
        .ent    symbol,0;                       \  
symbol:     .frame  sp, framesize, rpc  
  
汇编伪指令  frame 用来声明堆栈布局。  
它有三个参数：  
    1）第一个参数  framereg：声明用于访问局部堆栈的寄存器，一般为  $sp。  
    2）第二个参数  framesize：申明该函数已分配堆栈的大小，应该符合  $sp ＋ framesize ＝ 原来的  $sp。  
    3）第三个参数  returnreg：这个寄存器用来保存返回地址。  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
    kernel\_entry\_setup          # cpu specific setup  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
这个宏一般为空的，在 include/asm-mips/mach-generic/kernel-entry-init.h 文件中定义。  
  
某些MIPS CPU需要额外的设置一些控制寄  
存器，和具体的平台相关，一般为空宏；某些多核MIPS，启动时所  
有的core的入口一起指向   kernel\_entry，然后在该宏里分叉，  
boot core 继续往下，其它的则不停的判断循环，直到boot core 唤醒之  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
  
    setup\_c0\_status\_pri  
设置   cp0\_status 寄存器  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
    .macro  setup\_c0\_status\_pri  
#ifdef CONFIG\_64BIT  
    setup\_c0\_status ST0\_KX 0  
#else  
    setup\_c0\_status 0 0  
#endif  
    .endm  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
    ARC64\_TWIDDLE\_PC  
除非 CONFIG\_ARC64，否则为空操作  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
  
#ifdef CONFIG\_MIPS\_MT\_SMTC  
    mtc0    zero, CP0\_TCCONTEXT\_\_bss\_start  
  
    mfc0    t0, CP0\_STATUS  
    ori t0, t0, 0xff1f  
    xori    t0, t0, 0x001e  
    mtc0    t0, CP0\_STATUS  
#endif /\* CONFIG\_MIPS\_MT\_SMTC \*/  
  
宏定义   CONFIG\_MIPS\_MT\_SMTC 是使用多核的  SMTC Linux 时定义的。一般情况下不考虑。  
MIPS已经开发出  SMP Linux的改进版，叫做SMTC(线程上下文对称多处理) Linux。  
SMTC Linux能理解轻量级  TC 的概念，并能因此减少某些与SMP Linux相关的开销。  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
    PTR\_LA      t0, \_\_bss\_start     # clear .bss  
    LONG\_S      zero, (t0)  
    PTR\_LA      t1, \_\_bss\_stop - LONGSIZE  
1:  
    PTR\_ADDIU   t0, LONGSIZE  
    LONG\_S      zero, (t0)  
    bne     t0, t1, 1b  
  
清除  BSS 段，清 0。  
变量   \_\_bss\_start  和   \_\_bss\_stop 在连接文件arch/mips/kernel/vmlinux.lds 中定义。  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
    LONG\_S      a0, fw\_arg0     # firmware arguments  
    LONG\_S      a1, fw\_arg1  
    LONG\_S      a2, fw\_arg2  
    LONG\_S      a3, fw\_arg3  
把  bootloader 传递给内核的启动参数保存在 fw\_arg0，fw\_arg1，fw\_arg2，fw\_arg3 变量中。  
变量  fw\_arg0 为内核参数的个数，其余分别为字符串指针，为  ＊＊＊ ＝ XXXX  的格式。  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
    MTC0        zero, CP0\_CONTEXT   # clear context register  
清除  CP0 的 context register，这个寄存器用来保存页表的起始地址。  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
    PTR\_LA      $28, init\_thread\_union  
初始化  $gp 寄存器，这个寄存器的地址指向一个  union，  
THREAD\_SIZE  大小，最低处是一个thread\_info 结构  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
    PTR\_LI      sp, \_THREAD\_SIZE - 32  
    PTR\_ADDU    sp, $28  
设置  $sp 寄存器，堆栈指针。  $sp = (init\_thread\_union 的地址) ＋  \_THREAD\_SIZE - 32  
的得出  $sp 指向这个  union  结构的结尾地址  － 32 字节地址。  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
    set\_saved\_sp    sp, t0, t1  
把  这个 CPU 核的堆栈地址  $sp 保存到  kernelsp[NR\_CPUS] 数组。  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
    如果定义了  CONFIG\_SMP 宏，即多  CPU 核。  
        .macro  set\_saved\_sp stackp temp temp2  
#ifdef CONFIG\_MIPS\_MT\_SMTC  
        mfc0    \temp, CP0\_TCBIND  
#else  
        MFC0    \temp, CP0\_CONTEXT  
#endif  
        LONG\_SRL    \temp, PTEBASE\_SHIFT  
        LONG\_S  \stackp, kernelsp(\temp)  
        .endm  
如果没有定义  CONFIG\_SMP 宏，单  CPU 核。  
        .macro  set\_saved\_sp stackp temp temp2  
        LONG\_S  \stackp, kernelsp  
        .endm  
  
变量  kernelsp 的定义，在 arch/mips/kernel/setup.c 文件中。  
unsigned long kernelsp[NR\_CPUS];  
把  这个 CPU 核的堆栈地址  $sp 保存到  kernelsp[NR\_CPUS] 数组。  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
    PTR\_SUBU    sp, 4 \* SZREG       # init stack pointer  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
    j       start\_kernel  
    END(kernel\_entry)  
最后跳转到  /arch/mips/kernel/main.c 中的 start\_kernel()初始化硬件平台相关的代码。  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊  
这个   init\_thread\_union 变量在  arch/mips/kernel/init\_task.c 文件中定义。  
  
union thread\_union init\_thread\_union  
    \_\_attribute\_\_((\_\_section\_\_(".data.init\_task"),  
                   \_\_aligned\_\_(THREAD\_SIZE))) =  
        { INIT\_THREAD\_INFO(init\_task) };  
＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊  
问题：  
    1）这个  init\_thread\_union 结构体指针是怎么初始化的？

＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊

**linux-mips启动分析（2）**

linux  内核启动的第一个阶段是从  /arch/mips/kernel/head.s文件开始的。  
而此处正是内核入口函数kernel\_entry(),该函数定义在 /arch/mips/kernel/head.s文件里。  
  
kernel\_entry()函数是体系结构相关的汇编语言，它首先初始化内核堆栈段，来为创建系统中的第一个进程进行准备， 接着用一段循环将内核映像的未初始化数据段（bss段，在\_edata和\_end之间）清零， 最后跳转到  /arch/mips/kernel/main.c 中的 start\_kernel()初始化硬件平台相关的代码。  
下面讲述   start\_kernel() 函数。  
＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊  
  
asmlinkage void \_\_init start\_kernel(void)  
{  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
    char \* command\_line;  
    extern struct kernel\_param \_\_start\_\_\_param[], \_\_stop\_\_\_param[];  
    定义了核的参数数据结构  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
    smp\_setup\_processor\_id();  
    设置  SMP 多核的  CPU 核的 ID 号，单核不进行任何操作，我们不关心。  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
    unwind\_init();  
    在   MIPS 体系结构中，这个函数是个空函数（可能调用setup\_arch，配置核的相关函数）  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
    lockdep\_init();  
    初始化核依赖关系哈希表。  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
    local\_irq\_disable();  
    关闭当前  CPU 核的中断  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
    early\_boot\_irqs\_off();  
    通过一个静态全局变量    early\_boot\_irqs\_enabled 来帮助我们调试代码，  
    通过这个标记可以帮助我们知道是否在“early bootup code”，  
    也可以通过这个标志警告是否有无效的中断打开。  
    和     early\_boot\_irqs\_on() 函数配置使用，参考下面。  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
    early\_init\_irq\_lock\_class();  
    每一个中断都有一个    IRQ 描述符 （ struct irq\_desc ）来进行描述。  
    这个函数的主要作用是设置所有的   IRQ 描述符 （ struct irq\_desc ）的锁是统一的锁，  
    还是每一个    IRQ 描述符 （ struct irq\_desc ）都有一个小锁。  
    参考《linux-mips启动分析（2-1）》。  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
    lock\_kernel();  
    获取大内核锁，这种大内核锁锁定整个内核。  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
    tick\_init();  
    如果没有定义   CONFIG\_GENERIC\_CLOCKEVENTS 宏定义，则这个函数为空函数，  
    如果定义了这个宏，这执行初始化  tick 控制功能，注册 clockevents 的框架。  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
    boot\_cpu\_init();  
    对于  CPU 核的系统来说，设置第一个  CPU 核为活跃  CPU 核。  
    对于单  CPU 核系统来说，设置 CPU 核为活跃  CPU 核。  
    参考《linux-mips启动分析（2-1）》。  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
    page\_address\_init();  
    当定义了 CONFIG\_HIGHMEM  宏，并且没有定义   WANT\_PAGE\_VIRTUAL  宏时，非空函数。  
    其他情况为空函数。  
    参考《linux-mips启动分析（2-1）》。  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
    printk(KERN\_NOTICE);  
    printk(linux\_banner);  
    输出打印版本信息。  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
    setup\_arch(&command\_line);  
    每种体系结构都有自己的  setup\_arch() 函数，这些是体系结构相关的。  
    如何确定编译那个体系结构的    setup\_arch() 函数呢？  
    主要由  linux 源码树顶层  Makefile 中  ARCH 变量来决定的。  
    例如：  MIPS 体系结构的。  
    SUBARCH := mips  
    ARCH        ?= $(SUBARCH)  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
    setup\_command\_line(command\_line);  
    保存未改变的  comand\_line  到字符数组  static\_command\_line［］ 中。  
    保存  boot\_command\_line 到字符数组     saved\_command\_line［］ 中。  
    参考《linux-mips启动分析（2-1）》。  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
    unwind\_setup();  
    空函数。  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
    setup\_per\_cpu\_areas();  
    如果没有定义  CONFIG\_SMP 宏，则这个函数为空函数。  
    如果定义了    CONFIG\_SMP 宏，  
    则这个     setup\_per\_cpu\_areas() 函数给每个CPU分配内存，并拷贝   .data.percpu 段的数据。  
    参考《linux-mips启动分析（2-1）》。  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
    如果没有定义  CONFIG\_SMP 宏，则这个函数为空函数。  
    如果定义了    CONFIG\_SMP 宏，这个函数  
    smp\_prepare\_boot\_cpu();   
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
    sched\_init();  
    核心进程调度器初始化，调度器的初始化优先于任何中断的建立（包括  timer 中断）。  
    并且初始化进程 0 ，即  idle 进程，但是并没有设置 idle 进程的  NEED\_RESCHED 标志，  
    以完成内核剩余的启动部分。  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
    preempt\_disable();  
    进制内核的抢占。使当前进程的   struct thread\_info 结构  preempt\_count 成员的值增加 1。  
    参考《linux-mips启动分析（2-1）》。  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
    建立各个节点的管理区的  zonelist，便于分配内存的   fallback 使用。  
    这个链表的作用：  这个链表是为了在一个分配不能够满足时可以考察下一个管理区来设置了。  
    在考察结束时，分配将从  ZONE\_HIGHMEM 回退到  ZONE\_NORMAL，  
    在分配时从  ZONE\_NORMAL 退回到  ZONE\_DMA 就不会回退了。  
    build\_all\_zonelists();  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
    参考《linux-mips启动分析（2-1）》。  
    page\_alloc\_init();  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
    在  MIPS 体系结构下，这个函数已经在   arch\_mem\_init(） 函数中调用了一次。  
    这个函数的具体分析详细分析，请看《linux-mips启动分析（4）》。  
    所以这个函数直接返回。  
    parse\_early\_param();  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
    打印  linux 启动命令行参数。  
    printk(KERN\_NOTICE "Kernel command line: %s\n", boot\_command\_line);  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
    这个函数在  《linux-mips启动分析（4）》文件有分析，可以参考。  
    这个函数的意思对  linux 启动命令行参数进行再分析和处理。  
    这两个变量   \_\_start\_\_\_param 和   \_\_stop\_\_\_param 在  
    链接脚本   arch/mips/kernel/vmlinux.lds 中定义。  
    最后一个参数为，当不能够识别  linux 启动命令行参数时，调用的函数。  
    parse\_args("Booting kernel", static\_command\_line, \_\_start\_\_\_param,  
           \_\_stop\_\_\_param - \_\_start\_\_\_param, &unknown\_bootoption);  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
     检查中断是否已经打开了，如果已将打开了，关闭中断。  
    if (!irqs\_disabled()) {  
        local\_irq\_disable();  
         }  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
    sort\_main\_extable();  
    这个函数对内核建立的异常处理调用函数表（ exception table ）  
    根据异常的向量号进行堆排序。  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
    设置  CPU 的异常处理函数，TLB 重填，cache 出错，还有通用异常处理表的初始化。  
    trap\_init();  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
        初始化  RCU 机制，这个步骤必须比本地  timer 的初始化早。  
    rcu\_init();  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
        用来初始化中断处理硬件相关的寄存器和 中断描述符数组     irq\_desc［］ 数组，  
    每个中断号都有一个对应的中断描述符。  
    参考《linux-mips启动分析（11）》。  
    init\_IRQ();  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
    系统在初始化阶段动态的分配了  4   个  hashtable，并把它们的地址存入  pid\_hash［］ 数组。  
    便于从 PID 查找 进程描述符地址。  
    参考《linux-mips启动分析（12）》。  
    pidhash\_init();  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
     1）初始化本 CPU 上的定时器（timer）相关的数据结构  
    2）向   cpu\_chain 通知链注册元素 timers\_nb，该元素的回调函数用于初始化指定 CPU 上的定时器相关的数据结构。  
    3） 初始化时钟的软中断处理函数   
    参考《linux-mips启动分析（13）》。  
  
    init\_timers();  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
    这个       hrtimers\_init（） 函数对高精度时钟进行初始化。  
    参考《linux-mips启动分析（14）》。  
    hrtimers\_init();  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
    softirq\_init();  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
    调用  timekeeping\_init（） 函数来初始化系统计时器。  
    参考《linux-mips启动分析（16）》。  
    timekeeping\_init();  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
    调用  time\_init() 函数，初始化系统时钟源，  
    对  MIPS 体系结构的    clocksource\_mips  时钟源进行配置，和注册。  
    参考《linux-mips启动分析（17》。  
    time\_init();  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
    这个函数对内核的  profile 功能进行初始化，这是一个内核调式工具，  
    通过这个可以发现内核在内核态的什么地方花费时间最多，  
    即发现内核的“hot spot”——执行最频繁的内核代码。  
    参考《linux-mips启动分析（18）。  
    profile\_init();  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
    检测是否已经打开了中断。  
    如果已经打开了中断，则打印提示信息，报告一个  bug 。  
    if (!irqs\_disabled())  
        printk(打印提示信息，报告一个  bug );  
    通过一个静态全局变量    early\_boot\_irqs\_enabled 来帮助我们调试代码，  
    通过这个标记可以帮助我们知道是否在“early bootup code”，  
    也可以通过这个标志警告是否有无效的中断打开。  
    和     early\_boot\_irqs\_off () 函数配置使用，参考上面。  
    early\_boot\_irqs\_on();  
  
    使能本地  CPU 的中断。  
    local\_irq\_enable();  
====================================================  
  
总结：上面是早期启动阶段，在上面的执行过程中，中断是关闭的。  
  
# 输出Linux版本信息（printk(linux\_banner)）  
# 设置与体系结构相关的环境（setup\_arch()）  
# 页表结构初始化（paging\_init()）  
# 设置系统自陷入口（trap\_init()）  
#初始化系统IRQ（init\_IRQ()）  
# 核心进程调度器初始化（sched\_init()）  
# 时间、定时器初始化（包括读取CMOS时钟、估测主频、初始化定时器中断等，time\_init()）  
# 提取并分析核心启动参数（从环境变量中读取参数，设置相应标志位等待处理，（parse\_options()）   
======================================================  
  
    console\_init();  
    这个    console\_init() 函数初始化内核显示终端，在这里仅仅进行一些初级的初始化。  
    参考《linux-mips启动分析（19）》。  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
    如果这个  panic\_later 字符串已经设置了，则停止系统的启动。  
    if (panic\_later)  
        panic(panic\_later, panic\_param);  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
    如果没有定义   LOCKDEP 这个宏，则这个     lockdep\_info() 函数就为空函数。  
    lockdep\_info();  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
    如果没有定义   CONFIG\_DEBUG\_LOCKING\_API\_SELFTESTS 这个宏，  
    则这个  locking\_selftest() 函数为空函数。  
    locking\_selftest();  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
#ifdef CONFIG\_BLK\_DEV\_INITRD  
    if (initrd\_start && !initrd\_below\_start\_ok   
                            && initrd\_start < min\_low\_pfn << PAGE\_SHIFT) {  
        initrd\_start = 0;  
           }  
#endif

－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
    vfs\_caches\_init\_early();  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
    cpuset\_init\_early();  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
    mem\_init();  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
    kmem\_cache\_init();  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
    setup\_per\_cpu\_pageset();  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
    numa\_policy\_init();  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
    if (late\_time\_init)  
        late\_time\_init();  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
    calibrate\_delay();  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
    pidmap\_init();  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
    pgtable\_cache\_init();  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
    prio\_tree\_init();  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
    anon\_vma\_init();  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
    fork\_init(num\_physpages);  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
｝  
  
＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊  
问题：  
    1）核依赖关系哈希表是如何使用的？什么功能？  
    2）在变量   \_\_per\_cpu\_end  和  \_\_per\_cpu\_start 之间保存了什么数据？

**linux-mips启动分析 （2-1）**

这个文件讲解了  start\_kernel（） 函数中调用的一些小函数，可以作为参考。  
  
有下列函数：  
early\_init\_irq\_lock\_class（）  
boot\_cpu\_init（）  
page\_address\_init（）  
setup\_command\_line（）  
setup\_per\_cpu\_areas（）  
preempt\_disable（）  
  
＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊  
  
如果没有定义宏   CONFIG\_TRACE\_IRQFLAGS  和     CONFIG\_GENERIC\_HARDIRQS，  
函数      early\_init\_irq\_lock\_class() 就是空的；  
  
#if defined(CONFIG\_TRACE\_IRQFLAGS) && defined(CONFIG\_GENERIC\_HARDIRQS)  
extern void early\_init\_irq\_lock\_class(void);  
#else  
static inline void early\_init\_irq\_lock\_class(void)  
{  
}  
#endif  
  
如果没有定义宏   CONFIG\_TRACE\_IRQFLAGS  和     CONFIG\_GENERIC\_HARDIRQS，  
则      early\_init\_irq\_lock\_class() 函数在   kernel/irq/handle.c  文件中，  
进行了定义；  
  
static struct lock\_class\_key irq\_desc\_lock\_class;  
  
void early\_init\_irq\_lock\_class(void)  
{  
    int i;  
      
    for (i = 0; i < NR\_IRQS; i++)  
        lockdep\_set\_class(&irq\_desc[i].lock, &irq\_desc\_lock\_class);  
}     
每一个中断都有一个    IRQ 描述符 （ struct irq\_desc ）来进行描述。  
这个函数的主要作用是设置所有的   IRQ 描述符 （ struct irq\_desc ）的锁是统一的锁，  
还是每一个    IRQ 描述符 （ struct irq\_desc ）都有一个小锁。  
＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊  
  
这个  boot\_cpu\_init(） 函数设置  CPU 位图信息。  
参考《linux的CPU信息管理》。    
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
static void \_\_init boot\_cpu\_init(void)  
{     
    int cpu = smp\_processor\_id();  
    获取当前 CPU 的 ID 号。  
    cpu\_set(cpu, cpu\_online\_map);  
    cpu\_set(cpu, cpu\_present\_map);  
    cpu\_set(cpu, cpu\_possible\_map);  
}     
＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊  
  
    page\_address\_init();  
    当定义了 CONFIG\_HIGHMEM  宏，并且没有定义   WANT\_PAGE\_VIRTUAL  宏时，非空函数。  
    其他情况       page\_address\_init()  函数为空函数。  
  
static struct page\_address\_map page\_address\_maps[LAST\_PKMAP];  
  
void \_\_init page\_address\_init(void)  
{     
    int i;  
      
    INIT\_LIST\_HEAD(&page\_address\_pool);  
    for (i = 0; i < ARRAY\_SIZE(page\_address\_maps); i++)  
        list\_add(&page\_address\_maps[i].list, &page\_address\_pool);  
    for (i = 0; i < ARRAY\_SIZE(page\_address\_htable); i++) {  
        INIT\_LIST\_HEAD(&page\_address\_htable[i].lh);  
        spin\_lock\_init(&page\_address\_htable[i].lock);  
           }  
    spin\_lock\_init(&pool\_lock);  
}  
  
把结构体  struct page\_address\_map 类型的数组  page\_address\_maps 加入链表  page\_address\_pool 上。  
＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊  
  
字符数组   command\_line［］ 和  boot\_command\_line［］在   arch\_mem\_init(） 函数中赋值，  
为命令行启动参数字符串组合。参考《linux-mips启动分析（4）》。  
这两个字符指针   saved\_command\_line 和   static\_command\_line 在  main.c 文件中定义。  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
  
static void \_\_init setup\_command\_line(char \*command\_line)  
{  
    使用引导内存分配器分配内存   command\_line 的保存空间，并拷贝到这个空间中。  
    saved\_command\_line = alloc\_bootmem(strlen (boot\_command\_line)+1);  
    static\_command\_line = alloc\_bootmem(strlen (command\_line)+1);  
    strcpy (saved\_command\_line, boot\_command\_line);  
    strcpy (static\_command\_line, command\_line);  
}  
＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊  
  
如果没有定义  CONFIG\_SMP 宏，则这个函数为空函数。  
如果定义了    CONFIG\_SMP 宏，  
则这个     setup\_per\_cpu\_areas() 函数给每个CPU分配  pre-cpu 结构内存，  
并拷贝   .data.percpu 段的数据。  
  
参考《每CPU变量的数据组织和访问》。  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
  
static void \_\_init setup\_per\_cpu\_areas(void)  
{  
    unsigned long size, i;  
    char \*ptr;  
  
    取得  CPU 核的数目。  
    unsigned long nr\_possible\_cpus = num\_possible\_cpus();  
  
    为每个  CPU 分配  size 大小的空间。  
    size = ALIGN(PERCPU\_ENOUGH\_ROOM, PAGE\_SIZE);  
    ptr = alloc\_bootmem\_pages(size \* nr\_possible\_cpus);  
  
    for\_each\_possible\_cpu(i) {  
        \_\_per\_cpu\_offset[i] = ptr - \_\_per\_cpu\_start;  
        memcpy(ptr, \_\_per\_cpu\_start, \_\_per\_cpu\_end - \_\_per\_cpu\_start);  
        ptr += size;  
          }  
}  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
变量   \_\_per\_cpu\_end  和  \_\_per\_cpu\_start 在链接  
文件  arch/mips/kernel/vmlinux.lds 中定义。  
  
在该函数中，为每个CPU分配一段专有数据区，并将.data.percpu中的数据拷贝到其中，  
每个CPU各有一份。由于数据从\_\_per\_cpu\_start处转移到各CPU自己的专有数据区中了，  
因此存取其中的变量就不能再用原先的值了，比如存取per\_cpu\_\_runqueues  
就不能再用per\_cpu\_\_runqueues了，需要做一个偏移量的调整，  
即需要加上各CPU自己的专有数据区首地址相对于\_\_per\_cpu\_start的偏移量。  
在这里也就是\_\_per\_cpu\_offset[i]，其中CPU i的专有数据区相对于  
\_\_per\_cpu\_start的偏移量为\_\_per\_cpu\_offset[i]。  
这样，就可以方便地计算专有数据区中各变量的新地址，比如对于per\_cpu\_runqueues，  
其新地址即变成per\_cpu\_runqueues  + \_\_per\_cpu\_offset[i]。   
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
  
定义  per\_cpu 数据时使用这个   DEFINE\_PER\_CPU(type, name)  宏定义  
它的定义如下：  
  
如果定义了  CONFIG\_SMP：  
#define DEFINE\_PER\_CPU(type, name) \  
    \_\_attribute\_\_((\_\_section\_\_(".data.percpu"))) \_\_typeof\_\_(type) per\_cpu\_\_##name  
  
访问  per\_cpu 数据的实现：  
#define per\_cpu(var, cpu) (\*({              \  
    extern int simple\_identifier\_##var(void);   \  
    RELOC\_HIDE(&per\_cpu\_\_##var, \_\_per\_cpu\_offset[cpu]); }))  
#define \_\_get\_cpu\_var(var) per\_cpu(var, smp\_processor\_id())  
#define \_\_raw\_get\_cpu\_var(var) per\_cpu(var, raw\_smp\_processor\_id())  
  
如果没有定义  CONFIG\_SMP：  
#define DEFINE\_PER\_CPU(type, name) \  
    \_\_typeof\_\_(type) per\_cpu\_\_##name  
  
访问  per\_cpu 数据的实现：  
#define per\_cpu(var, cpu)           (\*((void)(cpu), &per\_cpu\_\_##var))  
#define \_\_get\_cpu\_var(var)          per\_cpu\_\_##var  
#define \_\_raw\_get\_cpu\_var(var)          per\_cpu\_\_##var  
＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊  
  
如果没有定义  CONFIG\_PREEMPT 这个宏，这个   preempt\_disable（）函数为空函数。  
如果定义了这个   CONFIG\_PREEMPT 这个宏。  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
这个 preempt\_disable（）函数在  include/linux/preempt.h 文件中实现。  
  
#define preempt\_disable() \   
do { \  
    inc\_preempt\_count(); \  
    barrier(); \  
} while (0)  
  
主要由宏    inc\_preempt\_count() 来完成工作。  
  
#define inc\_preempt\_count() add\_preempt\_count(1)  
  
# define add\_preempt\_count(val) do { preempt\_count() += (val); } while (0)  
  
#define preempt\_count() (current\_thread\_info()->preempt\_count)  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
由以上可以看出这个   preempt\_disable（）函数主要工作是使当前进程  
的  struct thread\_info 结构  preempt\_count 成员的值增加 1。  
＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊  
void \_\_init page\_alloc\_init(void)  
{     
    hotcpu\_notifier(page\_alloc\_cpu\_notify, 0);  
}     
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
这个  hotcpu\_notifier(）宏函数在  include/linux/cpu.h 文件中实现的，  
这个函数有两个版本，通过  CONFIG\_HOTPLUG\_CPU 宏定义来进行控制。  
  
我们的系统不支持  CPU 的热插拔，所以没有定义这个  CONFIG\_HOTPLUG\_CPU 宏。  
所以它的实现如下所示：  
  
#define hotcpu\_notifier(fn, pri)    do { (void)(fn); } while (0)  
  
它的第一个参数是一个函数指针，这个函数的意思是直接调用这个函数指针就行了。  
不过经过测试，好像也没有调用  page\_alloc\_cpu\_notify（） 函数。  
＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊  
  
问题：  
    1）这个    page\_alloc\_init(） 函数好像没有执行任何的操作，不知道对不对？

**linux-mips启动分析（3）**

在  start\_kernel（） 函数中调用了 setup\_arch() 函数。  
每种体系结构都有自己的  setup\_arch() 函数，这些是体系结构相关的。  
如何确定编译那个体系结构的    setup\_arch() 函数呢？  
主要由  linux 源码树顶层  Makefile 中  ARCH 变量来决定的。  
    例如：  MIPS 体系结构的。  
    SUBARCH := mips  
    ARCH        ?= $(SUBARCH)  
  
下面我们分析一下   MIPS 体系结构的   setup\_arch() 函数。  
  
从《linux-mips启动分析（3）》到《linux-mips启动分析（5）》文件中 一直在讲述 setup\_arch() 函数。  
＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊  
  
位于   /arch/mips/kernel/setup.c 文件中。  
参数  cmdline\_p 为字符的指针的指针，没有赋值。  
可能为了以后把内核启动参数保存到这个指针指向的字符串中。  
  
void \_\_init setup\_arch(char \*\*cmdline\_p)  
{  
    cpu\_probe();  
    调用函数cpu\_probe(),该函数通过MIPS CPU的PRID寄存器来确定CPU类型，  
    从而确定使用的指令集和其他一些CPU参数，如TLB等  
  
    prom\_init();  
    prom\_init() 函数是和硬件相关的，做一些低层的初始化，接受引导装载程序传给内核的参数，  
    确定 mips\_machgroup，mips\_machtype 这两个变量，这两个变量分别对应着相应的芯片组合开发板；  
  
    打印  cpu\_probe() 函数检测到的  CPU 的  Processor ID。  
    如果有浮点处理器，也打印浮点处理器的   Processor ID。  
    cpu\_report();  
  
    应用程序通过终端接口设备使用特定的接口规程与终端进行交互，与操作系统内核本身交互的终端称为控制台，  
    它可以是内核本身的内部显示终端，也可以是通过串口连接的外部哑终端。   
    由于大多数情况下控制台都是内核显示终端，因此内核显示终端也常常直接称为控制台。  
    内核终端对用户来说具有若干个虚拟终端子设备，它们共享同一物理终端，  
    但同一时刻只能有一个虚拟终端操作硬件屏幕。  
    宏  CONFIG\_VT 的意思是否支持虚拟终端。  
    当配置了宏  CONFIG\_VGA\_CONSOLE 时为内核本身的内部显示终端。  
    当配置了宏  CONFIG\_DUMMY\_CONSOLE 时为通过串口连接的外部哑终端。   
    用变量  conswitchp 来进行指定。  
#if defined(CONFIG\_VT)  
#if defined(CONFIG\_VGA\_CONSOLE)  
    conswitchp = &vga\_con;  
#elif defined(CONFIG\_DUMMY\_CONSOLE)  
    conswitchp = &dummy\_con;  
#endif  
#endif  
  
    对内存进行初始化。  
    arch\_mem\_init(cmdline\_p);  
  
    这个函数遍历每一个内存空间范围（物理地址），在资源管理器中进行资源申请，  
    并对内核代码和数据段进行资源申请。  
    resource\_init();  
  
#ifdef CONFIG\_SMP  
    plat\_smp\_setup();  
#endif  
}  
＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊  
  
第一函数：  
cpu\_probe ( )函数的部分源码如下：  
==========================================================  
这个   cpu\_data［］ 数组定义在  arch/mips/kernel/setup.c 文件中。  
定义如下所示：  
struct cpuinfo\_mips cpu\_data[NR\_CPUS] \_\_read\_mostly;  
  
它的类型为 struct cpuinfo\_mips 结构体：  
  
struct cpuinfo\_mips {  
    unsigned long       udelay\_val;  
    unsigned long       asid\_cache;  
    unsigned long       options;  
    unsigned long       ases;  
    unsigned int        processor\_id;  
    unsigned int        fpu\_id;  
    unsigned int        cputype;  
    int         isa\_level;  
    int         tlbsize;  
    struct cache\_desc   icache; /\* Primary I-cache \*/  
    struct cache\_desc   dcache; /\* Primary D or combined I/D cache \*/  
    struct cache\_desc   scache; /\* Secondary cache \*/  
    struct cache\_desc   tcache; /\* Tertiary/split secondary cache \*/  
    void            \*data;  /\* Additional data \*/  
} \_\_attribute\_\_((aligned(SMP\_CACHE\_BYTES)));  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
static inline void cpu\_probe(void) {  
  
        struct cpuinfo\_mips \*c = &current\_cpu\_data;  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
#define current\_cpu\_data cpu\_data[smp\_processor\_id()]  
    smp\_processor\_id() 函数获得当前  CPU 的  ID 号。  
    所以   cpuinfo\_mips 变量为当前  CPU 的数据结构指针。  
    下面对这个   CPU 的数据结构进行填充。  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
    c->processor\_id = read\_c0\_prid();  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
    获取CP0\_CONFIG寄存器的值，根据CP0控制寄存器PRID来确定CPU的类型  
+------------------+----------------+----------------+----------------+  
 |Company Options| Company ID     | Processor ID | Revision       |  
+------------------+----------------+----------------+----------------+  
  31     24 23                 16 15                 8 7                  0  
  这个寄存器的  ［23：16］位表明  CPU 的公司  ID 。  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
    switch (c->processor\_id & 0xff0000){  
    根据  PRID 的 ［23：16］位来选择  CPU 的公司  ID。  
  
    case PRID\_COMP\_CLXRISC: /\* CLXRISC Implementation. \*/  
        decode\_configs(c);  
        switch (c->processor\_id & 0xff000000) {  
        case PRID\_IMP\_CLXRISC:  
    对  CPU 的功能和特性进行描述。  
            c->options |= MIPS\_CPU\_MCHECK;  
            c->cputype = CPU\_CLXRISC;  
            c->isa\_level = MIPS\_CPU\_ISA\_M32R1;  
            c->tlbsize = 32;  
            break;  
        default:  
            c->cputype = CPU\_UNKNOWN;  
            break;  
                      }  
        break;  
           }  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
    根据  CPU 的特性进行检测   CPU 是否支持浮点运算单元，取得浮点运算单元的  ID 号。  
    并检测   CPU 是否支持  3D 图像运算，如果支持表明   CPU 支持。  
    if (c->options & MIPS\_CPU\_FPU) {  
        c->fpu\_id = cpu\_get\_fpu\_id();  
  
        if (c->isa\_level == MIPS\_CPU\_ISA\_M32R1 ||  
            c->isa\_level == MIPS\_CPU\_ISA\_M32R2 ||  
            c->isa\_level == MIPS\_CPU\_ISA\_M64R1 ||  
            c->isa\_level == MIPS\_CPU\_ISA\_M64R2) {  
            if (c->fpu\_id & MIPS\_FPIR\_3D)  
                c->ases |= MIPS\_ASE\_MIPS3D;  
                        }  
            }  
}  
 从中可以看出，cpu\_probe（）通过 CPU  的CP0控制寄存器   PRID 来对  CPU 功能和特性进行描述，  
这些特性在后面用来决定调用相应的异常处理和内存管理程序。  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
这个函数的  linux 内核移植相关部分：  
    1）在  include/asm-mips/cpu.h 中添加   CPU 公司的  ID。  
    2）修改添加  cpu\_probe（）函数中关于 CPU 公司 ID 的处理。  
＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊  
  
第二函数：  
  prom\_init() 函数的部分源码如下：  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
void \_\_init prom\_init(void)  
{  
    unsigned long memsize;  
  
    prom\_argc = (int) fw\_arg0;  
    prom\_argv = (char \*\*) fw\_arg1;  
    prom\_envp = (char \*\*) fw\_arg2;  
    这三个  fw\_arg0、fw\_arg1、fw\_arg2 变量的赋值，参考  《linux-mips启动分析（1）.txt》，  
    在  /arch/mips/kernel/head.s 文件中初始化的。  
  
    mips\_machgroup = MACH\_GROUP\_CLXRISC;  
    mips\_machtype  = 0;  
    初始化  mips\_machgroup，mips\_machtype  这两个变量，这两个变量分别对应着相应的芯片组合开发板；  
  
    把内核启动参数拷贝到  arcs\_cmdline［］字符数组中。  
    prom\_init\_cmdline();  
    memsize = 128;  
    if (memsize < 0x1000)  
        memsize \*= 1024\*1024;  
  
    在  结构体变量   boot\_mem\_map 中赋值指定的内存范围（物理地址）映像图。  
    add\_memory\_region(0, memsize, BOOT\_MEM\_RAM);  
    add\_memory\_region(0x28000000, memsize, BOOT\_MEM\_RAM);  
}  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
这个函数的  linux 内核移植相关部分：  
这个函数是和具体的硬件相关的，做一些底层的操作，移植  linux 内核时，需要自己手动写整个函数。  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
  
prom\_init() 函数调用  prom\_init\_cmdline（） 函数，  
  
void  prom\_init\_cmdline(void)  
{  
    char \*cp;  
    int actr;  
  
    actr = 1; /\* Always ignore argv[0] \*/  
  
    变量  arcs\_cmdline 为定义的字符数组，默认为字符串  CONFIG\_CMDLINE。  
    cp = &(arcs\_cmdline[0]);  
    while(actr < prom\_argc) {  
        strcpy(cp, prom\_argv[actr]);  
        cp += strlen(prom\_argv[actr]);  
        \*cp++ = ' ';  
        actr++;  
    }   
    if (cp != &(arcs\_cmdline[0]))     /\* get rid of trailing space \*/  
        --cp;  
    \*cp = '\0';  
  
}  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
如果内核启动参数为  
mem=64M console=tty0 console=ttyS0,115200n8 ip=192.168.4.46:::::eth0:off root=/dev/nfs  
nfsroot=192.168.4.55:/nfsroot/root-vw/ rw  
  
prom\_argv[]数组字符串如下所示。  
prom\_argv[1] len = 7.  prom\_argv[1]=mem=64M.  
prom\_argv[2] len = 12. prom\_argv[2]=console=tty0.  
prom\_argv[3] len = 22. prom\_argv[3]=console=ttyS0,115200n8.  
prom\_argv[4] len = 28. prom\_argv[4]=ip=192.168.4.46:::::eth0:off.  
prom\_argv[5] len = 13. prom\_argv[5]=root=/dev/nfs.  
prom\_argv[6] len = 38. prom\_argv[6]=nfsroot=192.168.4.55:/nfsroot/root-vw/.  
prom\_argv[7] len = 2.  prom\_argv[7]=rw.  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
add\_memory\_region（）函数在  arch/mips/kernel/setup.c 文件中定义。  
  
void \_\_init add\_memory\_region(phys\_t start, phys\_t size, long type)  
{  
    结构体变量   boot\_mem\_map 在 在  arch/mips/kernel/setup.c 文件中定义。  
    这个变量保存了命令行或者编译时指定的内存范围（物理地址）映像图。  
  
    int x = boot\_mem\_map.nr\_map;  
    struct boot\_mem\_map\_entry \*prev = boot\_mem\_map.map + x - 1;  
  
    如果添加的内存范围和已经有的内存范围有重合，进行合并。  
    if (x && prev->addr + prev->size == start && prev->type == type) {  
        prev->size += size;  
        return;  
           }  
    如果已经添加了最大量的内存范围，报错。  
    if (x == BOOT\_MEM\_MAP\_MAX) {  
        printk("Ooops! Too many entries in the memory map!\n");  
        return;  
          }  
    对内存范围（物理地址）映像图进行描述赋值。  
    boot\_mem\_map.map[x].addr = start;  
    boot\_mem\_map.map[x].size = size;  
    boot\_mem\_map.map[x].type = type;  
    boot\_mem\_map.nr\_map++;  
}  
＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊  
第三函数：  
  arch\_mem\_init () 函数主要对内存进行的初始化，内容比较复杂，  
在《linux-mips启动分析（4） 》文件对对内存系统进行比较细致的分析。  
＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊  
第四函数：  
    resource\_init（） 函数申请系统资源，具体分析如下：  
    这个函数遍历每一个内存空间范围（物理地址），在资源管理器中进行资源申请，  
    并对内核代码和数据段进行资源申请。  
    参考《linux-mips启动分析（5）》。  
＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊  
第五函数：  
    plat\_smp\_setup() 函数，如果没有定义  CONFIG\_SMP 宏定义，则不执行这个函数。  
＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊  
问题：  
    1）plat\_smp\_setup() 函数的功能？它执行了什么操作？

**linux-mips启动分析（4）**

在  setup\_arch（） 函数中调用了 arch\_mem\_init() 函数。  
  
参数  cmdline\_p 为字符的指针的指针，是从   start\_kernel（） 函数传给这个  setup\_arch() 函数的。  
在   setup\_arch() 函数没有对它进行处理。  
可能为了以后把内核启动参数保存到这个指针指向的字符串中。  
＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊  
  
位于  arch/mips/kernel/setup.c 文件中。  
  
static void \_\_init arch\_mem\_init(char \*\*cmdline\_p)  
{  
    这个函数是平台具体相关的，移植内核需要自己手动编写。  
    对于开发板的  CPU 和  board 的初始化都是在这个函数中进行的。  
    plat\_mem\_setup();  
  
    这个函数打印输出内存范围（物理地址）映像图。  
    printk("Determined physical RAM map:\n");  
    print\_memory\_map();  
  
    把字符数组   arcs\_cmdline［］  拷贝到字符数组   command\_line［］ 和  boot\_command\_line［］中。  
    字符数组   arcs\_cmdline［］  在  prom\_init\_cmdline（）  函数中进行的赋值。  
    为命令行启动参数字符串组合。参考《linux-mips启动分析（3）.txt》。  
    strlcpy(command\_line, arcs\_cmdline, sizeof(command\_line));  
    strlcpy(boot\_command\_line, command\_line, COMMAND\_LINE\_SIZE);  
  
    把字符数组   command\_line［］ 的地址赋值给函数的参数。  
    返回给上级的函数，实际上这个指针返回到了    start\_kernel（） 函数。  
    \*cmdline\_p = command\_line;  
  
    parse\_early\_param();  
  
    这个  usermem 变量在 early\_parse\_mem（）函数进行修改。  
    参考下面的分析。  
    if (usermem) {  
        printk("User-defined physical RAM map:\n");  
        print\_memory\_map();  
           }  
  
    bootmem\_init();  
    sparse\_init();  
    paging\_init();  
}  
＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊  
  
第一函数：  
    plat\_mem\_setup() 函数的分析：  
    这个函数是移植相关的，移植内核需要自己手动编写。  
    对于开发板的  CPU 和  board 的初始化都是在这个函数中进行的。  
    ========================================================  
  
在我们公司的开发板上代码如下：  
  
void \_\_init plat\_mem\_setup(void)  
{  
    主要是在这个函数中进行的  CPU 和  board 的初始化。  
    主要包括  SOC 各个功能模块的电源管理和时钟设置；  
    设置中断优先级；  
    设置  EMC 外部存储控制器的初始化；  
    设置  PCI 控制器的初始化，等等。  
    clx\_soc\_setup();  
  
    这三个变量  \_machine\_restart， \_machine\_halt，pm\_power\_off 是函数指针，  
    在 arch/mips/kernel/reset.c 文件中定义并使用的。  
    \_machine\_restart    = clx\_restart;     定义自己开发板的重启函数。  
    \_machine\_halt       = clx\_halt;           定义自己开发板的系统立即停止函数。  
    pm\_power\_off    = clx\_power\_off;       定义自己开发板的断电停止函数。  
  
    这个变量  board\_time\_init 也是一个函数指针，  
    在  arch/mips/kernel/time.c 文件中定义并使用。  
    为系统的  timer 系统初始化函数，在  time\_init（） 函数被调用。  
    board\_time\_init = cq8401\_time\_init;  
  
    设置  mips\_io\_port\_base 这个变量的值。  
    在  MIPS 体系结构下，  I/O 端口是由  memory 来映射的，并没有特殊的地址空间，  
    所以可以使用普通的  Load/Store 指令来读取。  
    这个  mips\_io\_port\_base  变量是一个虚拟地址，所有的端口都以这个地址为起始地址映射的。  
    set\_io\_port\_base(0);  
  
    对 PCI 的  IO 和   MEM 资源进行初始化。  
    资源管理器的根资源   I/O 或者   MEM 分别不同。  
    I/O 的根资源为    ioport\_resource。  
    MEM 的根资源为   iomem\_resource。  
    ioport\_resource.start   = 0x00000000;  
    ioport\_resource.end = 0xffffffff;  
    iomem\_resource.start    = 0x00000000;  
    iomem\_resource.end  = 0xffffffff;  
  
    定定义   CONFIG\_BLK\_DEV\_INITRD 决定是否是否使用了 ROMDISK 根文件系统。  
    如果定义了取得   ROMDISK 根文件系统的起始和结束地址。  
#ifdef CONFIG\_BLK\_DEV\_INITRD  
extern char \_\_initramfs\_start[], \_\_initramfs\_end[];  
    ROOT\_DEV    = MKDEV(RAMDISK\_MAJOR, 0);  
    initrd\_start    = (unsigned long)&\_\_initramfs\_start;  
    initrd\_end  = (unsigned long)&\_\_initramfs\_end;  
#endif  
  
      
#ifdef CONFIG\_PC\_KEYB  
    kbd\_ops = &std\_kbd\_ops;  
#endif  
  
    设置   NOR  flash MTD 分区和   MTD 设备的注册。  
#ifdef CONFIG\_MTD  
    physmap\_set\_partitions(cq8401\_mtd\_parts, number\_partitions);  
#endif  
  
    初始化串口。  
    clx\_serial\_setup();  
    设置开发板的板级初始化，移植相关，这个函数需要根据具体的设置进行操作。  
    clx\_board\_setup();    
}  
＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊  
第二函数：  
    print\_memory\_map() 函数的分析：  
    这个函数打印输出内存范围（物理地址）映像图，这个地址范围在  prom\_init（）函数  
    中调用  add\_memory\_region(），使用结构体变量   boot\_mem\_map 保存了命令行  
    或者编译时指定的内存范围（物理地址）映像图。  
    参考《linux-mips启动分析（3） 》。  
=====================================================  
  
打印输出信息：  
Determined physical RAM map:  
 memory: 04000000 @ 00000000 (usable)  
＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊  
第三函数：  
    parse\_early\_param() 函数分析：  
    这个函数主要内容还是在   parse\_args(） 这个函数中完成的。  
    这个     parse\_args(） 函数在  kernel/params.c 文件中实现的。  
    这个   parse\_args(） 函数是对启动命令行参数进行分析，并根据参数名称把值保存到  
    结构体    struct kernel\_param 类型的数组中。  
======================================================  
  
void \_\_init parse\_early\_param(void)  
{  
    static \_\_initdata char tmp\_cmdline[COMMAND\_LINE\_SIZE];  
  
    strlcpy(tmp\_cmdline, boot\_command\_line, COMMAND\_LINE\_SIZE);  
  
    因为没有设置  struct kernel\_param  结构体的指针，  
    所以这个和直接调用    do\_early\_param（har \*param, char \*val） 函数是一样的。  
    parse\_args("early options", tmp\_cmdline, NULL, 0, do\_early\_param);  
  
}  
    这个函数直接把内核启动命令行参数拷贝到临时的命令行参数字符数组中，  
    再对内核启动命令行参数进行语法上的分析。  
    主要内容还是在   parse\_args(） 这个函数中完成的。  
＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊  
  
    这个     parse\_args(） 函数在  kernel/params.c 文件中实现的。  
    这个     parse\_args(） 函数有五个参数：  
    第一个参数：此次操作的名称，好像没有用处，只是打印信息使用。  
    第二个参数：内核启动命令行参数字符串。  
    第三个参数：为  struct kernel\_param 结构体的变量数组，这个结构体变量用于保存内核启动命令行参数。  
    第四个参数：为 struct kernel\_param 结构体的变量数组的成员个数。  
    第五个参数：是一个函数指针，当内核启动命令行参数字符串的参数名称  
                     在 struct kernel\_param 结构体的变量数组中没有找到对应的名称时，  
                调用这个函数，当找到时，赋值对应的  struct kernel\_param 结构体的变量。  
                参数为  1） 参数名称；  2）参数值。  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
  
int parse\_args(const char \*name,  
           char \*args,  
           struct kernel\_param \*params,  
           unsigned num,  
           int (\*unknown)(char \*param, char \*val))  
{  
    char \*param, \*val;  
  
    跳过命令行的前导空白符。  
    while (\*args == ' ')  
        args++;  
  
    while (\*args) {  
        int ret;  
        int irq\_was\_disabled;  
  
        这个   next\_arg(） 函数是取得命令行中的下一个命令行参数的字符串。  
        args = next\_arg(args, &param, &val);  
  
        查看中断是否关闭，如果   irqs\_disabled 返回  0，中断没有关闭。  
        irq\_was\_disabled = irqs\_disabled();  
      
        对参数进行处理，看下面。  
        ret = parse\_one(param, val, params, num, unknown);  
        if (irq\_was\_disabled && !irqs\_disabled()) {  
            printk(打印警告信息);  
                     }  
        switch (ret) {  
        错误处理，等等。  
                     }  
           }  
  
    return 0;  
}  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
  
static int parse\_one(char \*param,  
             char \*val,  
             struct kernel\_param \*params,  
             unsigned num\_params,  
             int (\*handle\_unknown)(char \*param, char \*val))  
{  
    unsigned int i;  
  
    从    struct kernel\_param 结构体的变量数组，这个结构体变量用于保存内核启动命令行参数，  
    中找到对应的参数名称，并根据  linux 启动命令行的设置来设置这个参数的值。  
    for (i = 0; i < num\_params; i++) {  
        if (parameq(param, params[i].name)) {  
            DEBUGP("They are equal!  Calling %p\n",params[i].set);  
            return params[i].set(val, &params[i]);  
                      }  
          }  
  
    如果没有在  struct kernel\_param 结构体的变量数组中找到传入的  linux 启动命令行参数，  
    就调用默认的处理函数。  
    if (handle\_unknown) {  
        return handle\_unknown(param, val);  
          }  
    return -ENOENT;  
}  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
  
static int \_\_init do\_early\_param(char \*param, char \*val)  
{  
    struct obs\_kernel\_param \*p;  
  
      
    参考《Linux启动参数及实现》和 《启动时内核参数解析》。  
    变量    \_\_setup\_start 和  \_\_setup\_end 在连接脚本中进行定义的。  
    这两个变量决定了  init.setup 段的起始和结束地址。  
    启动参数(函数指针)被封装到    obs\_kernel\_param   结构中,  
    所有的内核启动参数形成内核映像     .init.setup  段中的一  
    个   obs\_kernel\_param 数组。  
    如何在内核启动参数中进行了这个某个参数，则对调用这个参数的支持函数。  
    for (p = \_\_setup\_start; p < \_\_setup\_end; p++) {  
      
        如果这个命令行参数有  early 处理函数，则调用这个  setup\_func（） 函数。  
        if (p->early && strcmp(param, p->str) == 0) {  
            if (p->setup\_func(val) != 0)  
                printk(KERN\_WARNING "Malformed early option '%s'\n", param);  
                        }  
            }  
  
    return 0;  
}  
＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊  
第四函数：  
    这个      early\_parse\_mem（） 函数比较特别，并不是在这里进行的调用，  
    而是在  do\_early\_param（） 函数执行时进行了调用。  
  
    因为在内核中定义了   early\_param("mem", early\_parse\_mem)。  
    当内核启动参数中定义了  mem 的大小时就会调用。  
=========================================================  
  
这个函数的意思是当  
  
static int \_\_init early\_parse\_mem(char \*p)  
{  
    unsigned long start, size;  
  
    如果用户在内核启动命令行内设置了   mem 的大小，取消以前设置的内存范围（物理地址）映像图。  
    以内核启动参数为高优先级。  
    if (usermem == 0) {  
        boot\_mem\_map.nr\_map = 0;  
        usermem = 1;  
           }  
    默认起始地址为  0。  
    start = 0;  
    size = memparse(p, &p);  
    如果命令行指定了内存的起始地址，从命令行取得内存的起始地址。  
    if (\*p == '@')  
        start = memparse(p + 1, &p);  
  
    调用  add\_memory\_region(），使用结构体变量   boot\_mem\_map 保存了命令行指定的内存范围（物理地址）映像图。  
    add\_memory\_region(start, size, BOOT\_MEM\_RAM);  
    return 0;  
}  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
代码运行到这里打印：  
User-defined physical RAM map:  
 memory: 04000000 @ 00000000 (usable)  
＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊  
  
第五函数：  
    bootmem\_init（）函数对引导内存分配器进行初始化。  
    这个函数太大了，包含的内容也比较多，所以在在《linux-mips启动分析（4-1）.txt》文件。  
＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊  
第六函数：  
    sparse\_init（）函数。  
  
    在  linux 内核中可以选择三种内部的  memory 管理模型：  FLATMEM、SPARSEMEM 和  DISCONTIGMEM。  
    一般情况下选择    FLATMEM 内存管理模型肯定是正常的，它支持所有的系统。  
    如果系统为非一致性的内存管理  NUMA 和支持内存热插拔的，可以选择其他两种。  
  
    这种  DISCONTIGMEM 内存管理模型相比于     FLATMEM 内存管理模提供了一些扩展的功能，  
    在有些系统中，内存有许多的空洞，这个中模型提供了更有效的的处理这些  hole 的方法。  
    然而，事实上尽管有许多的空洞，这些地址空间仍然是巨大的  flat 地址空间。  
    所以也可以退化为  FLATMEM 内存管理模型。  
    在许多的  NUMA 系统中，需要配置这种的内存管理模型。  
  
    这中 SPARSEMEM  内存管理模型可以支持内存的热插拔的，如果你不确定，可以不选择这个配置选项。  
  
    在我们的系统中，选择了  FLATMEM 内存管理模型。  
    没有定义这个配置选项   CONFIG\_SPARSEMEM  宏定义。  
    所以这个   sparse\_init（）函数是一个空函数。  
＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊  
第七函数：  
    paging\_init（）函数初始化内核空间的页表，建立页表项。  
    初始化内存节点和各个管理区和   mem\_map[] 数组。  
    参考《linux-mips启动分析（4-2）》。  
＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊  
问题：  
    1）内核都定义了哪些内核启动参数？都有哪些参数调用了参数的初始化？  
    2）在  IO 和  MEM 的端口操作中， 这个    mips\_io\_port\_base  变量如何使用的？

**linux-mips启动分析（4-1）**

第五函数：  
    bootmem\_init（）函数对引导内存分配器进行初始化。  
=========================================================  
  
static void \_\_init bootmem\_init(void)  
{  
    unsigned long reserved\_end;  
    unsigned long mapstart = ~0UL;  
    unsigned long bootmap\_size;  
    int i;  
  
    取得内核映像的末尾地址（物理地址），如果没有定义  CONFIG\_BLK\_DEV\_INITRD 这个宏，  
    这个  init\_initrd() 函数为空函数，如果定义了返回文件系统的映像的结束地址。  
    reserved\_end = max(init\_initrd(), PFN\_UP(\_\_pa\_symbol(&\_end)));  
  
    对变量  min\_low\_pfn 和  max\_low\_pfn 进行初始化。  
    变量  min\_low\_pfn 表示系统可用的最小的  PFN（page frame num）。  
    变量  max\_low\_pfn 表示低端内存可用的最大的  PFN（page frame num）。  
    min\_low\_pfn = ~0UL;  
    max\_low\_pfn = 0;  
  
    下面的循环对每一个内存区域（物理地址）进行检测，  
    对变量   min\_low\_pfn 和  max\_low\_pfn 根据硬件进行赋值。  
    对变量  mapstart 进行赋值，这个变量还是等于内核映像的结束地址（为什么不直接赋值呢？）。  
    for (i = 0; i < boot\_mem\_map.nr\_map; i++) {  
        unsigned long start, end;  
  
        if (boot\_mem\_map.map[i].type != BOOT\_MEM\_RAM)  
            continue;  
  
        start = PFN\_UP(boot\_mem\_map.map[i].addr);  
        end = PFN\_DOWN(boot\_mem\_map.map[i].addr  
                + boot\_mem\_map.map[i].size);  
  
        if (end > max\_low\_pfn)  
            max\_low\_pfn = end;  
        if (start < min\_low\_pfn)  
            min\_low\_pfn = start;  
        if (end <= reserved\_end)  
            continue;  
        if (start >= mapstart)  
            continue;  
        mapstart = max(reserved\_end, start);  
          }  
  
    进行检测。  
    if (min\_low\_pfn >= max\_low\_pfn)  
        panic("Incorrect memory mapping !!!");  
      
    if (min\_low\_pfn > ARCH\_PFN\_OFFSET) {  
        printk(KERN\_INFO  
               "Wasting %lu bytes for tracking %lu unused pages\n",  
               (min\_low\_pfn - ARCH\_PFN\_OFFSET) \* sizeof(struct page),  
               min\_low\_pfn - ARCH\_PFN\_OFFSET);  
    } else if (min\_low\_pfn < ARCH\_PFN\_OFFSET) {  
        printk(KERN\_INFO  
               "%lu free pages won't be used\n",  
               ARCH\_PFN\_OFFSET - min\_low\_pfn);  
          }  
    使变量    min\_low\_pfn ＝＝ 每个体系结构的最低物理地址。  
    在  MIPS 体系结构下 ARCH\_PFN\_OFFSET ＝＝ 0。  
    min\_low\_pfn = ARCH\_PFN\_OFFSET;  
  
    如果定义支持高端内存，在这里确定高端内存的范围，这个   HIGHMEM\_START 宏定义，决定了  
    高端内存地址的起始范围，为移植相关，移植  linux 内核时需要注意。  
    if (max\_low\_pfn > PFN\_DOWN(HIGHMEM\_START)) {  
#ifdef CONFIG\_HIGHMEM  
        highstart\_pfn = PFN\_DOWN(HIGHMEM\_START);  
        highend\_pfn = max\_low\_pfn;  
#endif  
        max\_low\_pfn = PFN\_DOWN(HIGHMEM\_START);  
           }  
  
    初始化低端内存引导内存分配器。分析见下面。  
    这个函数使用  mapstart 以后，第一个可用的  PFN 来初始化 min\_low\_pfn 和  max\_low\_pfn  
    之间的内存，并把节点加入   pgdat\_list  节点链表。  
    bootmap\_size = init\_bootmem\_node(NODE\_DATA(0), mapstart,  
                     min\_low\_pfn, max\_low\_pfn);  
  
    把所有低端内存中的有效  RAM 页面在引导内存分配器中进行分配。  
    for (i = 0; i < boot\_mem\_map.nr\_map; i++)  
      {  
        unsigned long start, end, size;  
  
        if (boot\_mem\_map.map[i].type != BOOT\_MEM\_RAM)  
            continue;  
  
        start = PFN\_UP(boot\_mem\_map.map[i].addr);  
        end   = PFN\_DOWN(boot\_mem\_map.map[i].addr  
                    + boot\_mem\_map.map[i].size);  
  
        跳过高端内存（如果包含高端内存  max\_low\_pfn ＝ = PFN\_DOWN(HIGHMEM\_START)）。  
        if (start >= max\_low\_pfn)  
            continue;  
        对内核映像所占据的  RAM 空间进行保留。  
        if (start < reserved\_end)  
            start = reserved\_end;  
        跳过高端内存（如果包含高端内存  max\_low\_pfn ＝ = PFN\_DOWN(HIGHMEM\_START)）。  
        if (end > max\_low\_pfn)  
            end = max\_low\_pfn;  
        进行错误检测。  
        if (end <= start)  
            continue;  
        计算得出这段 RAM 空间的 物理页面总数。  
        size = end - start;  
  
        在节点的位图中表目  size 个页面为空闲的。  
        free\_bootmem(PFN\_PHYS(start), size << PAGE\_SHIFT);  
  
        如果没有定义这个   CONFIG\_HAVE\_MEMORY\_PRESENT 宏定义，这个函数为空函数。  
        这个函数主要？？？？？？？？？？？？？？？？？？？？？？？？？？？？？？？  
        ？？？？？？？？？？？？？？？？？？？？？？？？？？？？？？？？？？？？？  
        memory\_present(0, start, end);  
           }  
  
    把节点的位图所占用的空间设置为保留（已占用），防止释放掉。  
    reserve\_bootmem(PFN\_PHYS(mapstart), bootmap\_size);  
  
    如果没有定义  CONFIG\_BLK\_DEV\_INITRD 这个宏，这个  init\_initrd(已占用) 函数为空函数，  
    如果定义了设置  INITRD 根文件系统所占用的内存为为保留，防止释放掉。  
    finalize\_initrd();  
}  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
这个函数注意点：  
    1）这个   HIGHMEM\_START 宏定义了高端内存的起始地址，为移植相关，移植  linux 内核时需要注意。  
＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊  
  
第一个函数：  
    init\_bootmem\_node（） 函数初始化引导内存分配器。  
  
    第一个参数：为  NODE\_DATA(0)，为第  0  个节点的指针，在  UMA 的机器上，  
                 只有一个静态的节点为   contig\_page\_data 表示。  
    第二个参数：为  mapstart， 即第一个可以使用的  PFN。  
    第三个参数：为   min\_low\_pfn  表示系统可用的最小的  PFN（page frame num）。  
    第四个参数：为   max\_low\_pfn  表示低端内存可用的最大的  PFN（page frame num）。  
========================================================  
  
第一个参数为  NODE\_DATA(0)，在  include/linux/mmzone.h 文件中定义。  
如果定义了这个  CONFIG\_NEED\_MULTIPLE\_NODES 宏，必须使用自己定义的。  
默认为系统定义静态内存节点  contig\_page\_data。  
 #define NODE\_DATA(nid)      (&contig\_page\_data)  
  
这个静态内存节点  contig\_page\_data，在  mm/page\_alloc.c 中进行的定义。  
static bootmem\_data\_t contig\_bootmem\_data;  
struct pglist\_data contig\_page\_data = { .bdata = &contig\_bootmem\_data };  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
  
typedef struct bootmem\_data {  
    unsigned long node\_boot\_start;  
    unsigned long node\_low\_pfn;  
    void \*node\_bootmem\_map;   
    unsigned long last\_offset;  
    unsigned long last\_pos;  
    unsigned long last\_success;   
    struct list\_head list;  
} bootmem\_data\_t;  
  
这个   bootmem\_data\_t 类型用于引导内存分配器分配内存时使用。  
这个   node\_bootmem\_map 成员是物理内存的映射位图，表明了这个节点的内存分配情况，  
包含内存节点的   holes 的情况。  
这个位图为 1 表示这个页面未分配，为 0 ，表示这个页面已经分配或者不能够使用。  
  
这个  node\_boot\_start 成员变量表示这个节点的起始物理地址。  
这个   node\_low\_pfn 成员变量表示这个节点的结束   PFN 。  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
  
unsigned long \_\_init init\_bootmem\_node(pg\_data\_t \*pgdat, unsigned long freepfn,  
                unsigned long startpfn, unsigned long endpfn)  
{     
    return init\_bootmem\_core(pgdat, freepfn, startpfn, endpfn);  
}     
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
  
static unsigned long \_\_init init\_bootmem\_core(pg\_data\_t \*pgdat,  
    unsigned long mapstart, unsigned long start, unsigned long end)  
{     
    取得内存映射结构体。这个结构体用于引导内存分配器分配内存时使用。  
    bootmem\_data\_t \*bdata = pgdat->bdata;  
    unsigned long mapsize;  
      
    这个  node\_bootmem\_map  成员变量包含已分配和空闲页面的位图的地址，  
    在这里设置为   mapstart 地址，即内核结束后的第一个  页面的起始地址。  
    bdata->node\_bootmem\_map = phys\_to\_virt(PFN\_PHYS(mapstart));  
  
    这个  node\_boot\_start 成员变量表示这个节点的起始物理地址。  
    bdata->node\_boot\_start = PFN\_PHYS(start);  
    这个   node\_low\_pfn 成员变量表示这个节点的结束   PFN 。  
    bdata->node\_low\_pfn = end;  
    把这个节点加入    pgdat\_list  节点链表。  
    link\_bootmem(bdata);  
  
    计算得出位图的大小，以一位代表一个页面。  
    mapsize = get\_mapsize(bdata);  
    设置保留所有的页面，为 1 ，在  setup\_arch（）函数中再详细的进行表明有效的  RAM 。  
    memset(bdata->node\_bootmem\_map, 0xff, mapsize);  
  
    return mapsize;  
}  
＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊  
  
第六函数：  
    sparse\_init（）函数。  
  
    在  linux 内核中可以选择三种内部的  memory 管理模型：  FLATMEM、SPARSEMEM 和  DISCONTIGMEM。  
    一般情况下选择    FLATMEM 内存管理模型肯定是正常的，它支持所有的系统。  
    如果系统为非一致性的内存管理  NUMA 和支持内存热插拔的，可以选择其他两种。  
  
    这种  DISCONTIGMEM 内存管理模型相比于     FLATMEM 内存管理模提供了一些扩展的功能，  
    在有些系统中，内存有许多的空洞，这个中模型提供了更有效的的处理这些  hole 的方法。  
    然而，事实上尽管有许多的空洞，这些地址空间仍然是巨大的  flat 地址空间。  
    所以也可以退化为  FLATMEM 内存管理模型。  
    在许多的  NUMA 系统中，需要配置这种的内存管理模型。  
  
    这中 SPARSEMEM  内存管理模型可以支持内存的热插拔的，如果你不确定，可以不选择这个配置选项。  
  
    在我们的系统中，选择了  FLATMEM 内存管理模型。  
    没有定义这个配置选项   CONFIG\_SPARSEMEM  宏定义。  
    所以这个   sparse\_init（）函数是一个空函数。  
＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊  
  
问题：  
    1）这个  memory\_present（）函数的作用？  
    2）三种内存管理模型  FLATMEM、SPARSEMEM 和  DISCONTIGMEM 的具体区别？

**linux-mips启动分析（4-2）**

第七函数：  
    paging\_init（）函数进行内核页表初始化。  
    这个函数在  arch/mips/mm/init.c 文件中定义实现的。  
=====================================================  
  
void \_\_init paging\_init(void)  
{  
      
    unsigned long zones\_size[MAX\_NR\_ZONES] = { 0, };  
  
    因为定义这个  CONFIG\_FLATMEM 宏，所以不定义下面的变量。  
#ifndef CONFIG\_FLATMEM  
    unsigned long zholes\_size[MAX\_NR\_ZONES] = { 0, };  
    unsigned long i, j, pfn;  
#endif  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－      
    这个函数利用  swapper\_pg\_dir 设立一个静态页表作为  PGD。  
    pagetable\_init();  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
#ifdef CONFIG\_HIGHMEM  
    这个函数取得高端内存首地址对应的  PTE 页表的起始页表项，并进行保护。  
    kmap\_init();  
#endif  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
    如果编译时没有设置   CONFIG\_MIPS\_MT\_SMTC 宏定义，下面这个函数为空函数。  
    kmap\_coherent\_init();  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
在内存中每个节点分为 三个管理区，    
ZONE\_DMA      表示低端范围的内存，  ISA 设备要用到它。  
ZONE\_NORMAL   由内核直接映射的线性地址空间。  
ZONE\_HIGHMEM  内核不能够直接映射的地址空间。  
  
现在  ISA 设备很少，一般不定义    CONFIG\_ZONE\_DMA 这个宏。  
    下面的代码设置内存节点的各个管理区的大小。  
  
#ifdef CONFIG\_ZONE\_DMA  
    if (min\_low\_pfn < MAX\_DMA\_PFN && MAX\_DMA\_PFN <= max\_low\_pfn) {  
        zones\_size[ZONE\_DMA] = MAX\_DMA\_PFN - min\_low\_pfn;  
        zones\_size[ZONE\_NORMAL] = max\_low\_pfn - MAX\_DMA\_PFN;  
    } else if (max\_low\_pfn < MAX\_DMA\_PFN)  
        zones\_size[ZONE\_DMA] = max\_low\_pfn - min\_low\_pfn;  
    else  
#endif  
    zones\_size[ZONE\_NORMAL] = max\_low\_pfn - min\_low\_pfn;  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
  
#ifdef CONFIG\_HIGHMEM  
    设置高端内存管理区的大小。  
    zones\_size[ZONE\_HIGHMEM] = highend\_pfn - highstart\_pfn;  
  
    这个  cpu\_has\_dc\_aliases 变量是一个宏定义，这个宏定义了数据高速缓存  
    数据重影的问题，参考《MIPS-cache重影问题的产生》。  
    如果  CPU 有数据高速缓存的数据重影问题，则不支持高端缓存。  
    if (cpu\_has\_dc\_aliases && zones\_size[ZONE\_HIGHMEM]) {  
        printk("This processor doesn't support highmem." );  
        zones\_size[ZONE\_HIGHMEM] = 0;  
          }  
#endif  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
  
    下面代码初始化各个管理区，建立各个管理区的内核页表。  
    对于  UMA 系统，调用    free\_area\_init（） 函数，  
    对于  NUMA 系统，调用     free\_area\_init\_node（） 函数。  
  
#ifdef CONFIG\_FLATMEM  
    初始化各个管理区和  mem\_map[]  数组。  
    free\_area\_init(zones\_size);  
#else  
    当为非  FLATMEM 类型内存管理模型时，执行。  
    pfn = min\_low\_pfn;  
    统计内存地址空间中的漏洞。  
    for (i = 0; i < MAX\_NR\_ZONES; i++)  
        for (j = 0; j < zones\_size[i]; j++, pfn++)  
            if (!page\_is\_ram(pfn))  
                zholes\_size[i]++;  
    free\_area\_init\_node(0, NODE\_DATA(0), zones\_size, 0, zholes\_size);  
#endif  
}  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
这个 free\_area\_init（） 函数和   free\_area\_init\_node（） 函数其实基本相同，  
都是通过调用   free\_area\_init\_core（） 函数来完成的。  
＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊  
  
    这个函数   pagetable\_init(）  利用   swapper\_pg\_dir 设立一个静态页表作为  PGD。  
    这个函数建立内核空间的页表。  
  
void \_\_init pagetable\_init(void)  
{  
    unsigned long vaddr;  
    pgd\_t \*pgd\_base;  
#ifdef CONFIG\_HIGHMEM  
    pgd\_t \*pgd;  
    pud\_t \*pud;  
    pmd\_t \*pmd;  
    pte\_t \*pte;  
#endif  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
    初始化 pgd\_t 类型的数组   swapper\_pg\_dir，这个数组中每一项指向  
    一个页面，这个页面包含一个   pte\_t 类型的书作。  
    这个函数初始化，这个 pgd\_t 类型的数组   swapper\_pg\_dir 数组中  
    每一项指向一个无效的   pte\_t 类型的数组。  
    pgd\_init((unsigned long)swapper\_pg\_dir);  
  
    再在刚才的基础上初始化  2  G 空间的   pgd\_t 项。  
    pgd\_init((unsigned long)swapper\_pg\_dir + sizeof(pgd\_t) \* USER\_PTRS\_PER\_PGD);  
  
    上面对内核空间的  4G 页表初始化为无效的。  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
    pgd\_base = swapper\_pg\_dir;  
  
    固定虚拟地址映射区域，固定映射的线性地址可以映射任何物理地址，比指针更有效。  
    固定虚拟地址空间被认为从  FIXADDR\_TOP 开始，并在虚拟地址空间前面结束。  
    这个  \_\_fix\_to\_virt（） 将一个下标作为参数。   
    \_\_end\_of\_fixed\_addresses 是上一个固定虚拟地址空间用到的下标。  
    这一行返回固定虚拟地址空间起始地址的   PMD 虚拟地址。  
    vaddr = \_\_fix\_to\_virt(\_\_end\_of\_fixed\_addresses - 1) & PMD\_MASK;  
      
    建立从  vaddr 地址对应的  PGD 表项开始的，到  PTRS\_PER\_PGD 结束的页目录项。  
    如果  PGD 表项无效，建立有效的 PGD 表项，即建立二级页表。  
  
    fixrange\_init(vaddr, 0, pgd\_base);  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
#ifdef CONFIG\_HIGHMEM  
  
    这个  PKMAP\_BASE 宏定义了  kmap 地址空间的大小。  
    vaddr = PKMAP\_BASE;  
    建立从  vaddr 地址对应的  PGD 表项开始的，到  kmap 地址空间结束的页目录项。  
    如果  PGD 表项无效，建立有效的 PGD 表项，即建立二级页表。  
    fixrange\_init(vaddr, vaddr + PAGE\_SIZE\*LAST\_PKMAP, pgd\_base);  
  
    pgd = swapper\_pg\_dir + \_\_pgd\_offset(vaddr);  
    pud = pud\_offset(pgd, vaddr);  
    pmd = pmd\_offset(pud, vaddr);  
    pte = pte\_offset\_kernel(pmd, vaddr);  
  
    最后取得   kmap 空间的二级页表基地址。  
    以后可能要用到。  
    pkmap\_page\_table = pte;  
#endif  
  
｝  
CONFIG\_MIPS\_MT\_SMTC  
＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊  
  
void pgd\_init(unsigned long page)  
{     
    unsigned long \*p = (unsigned long \*) page;  
    int i;  
      
  
    这个   USER\_PTRS\_PER\_PGD  定义在  include/asm-mips/pgtable-32.h 文件中，  
    #define USER\_PTRS\_PER\_PGD   (0x80000000UL/PGDIR\_SIZE)  
    为 2G 空间需要占用多少  pgd\_t 数组项。  
    对  2G 空间占用的  pgd\_t 数组项进行配置为无效的  pte\_t 数组地址。  
    for (i = 0; i < USER\_PTRS\_PER\_PGD; i+=8) {  
        p[i + 0] = (unsigneswapper\_pg\_dird long) invalid\_pte\_table;  
        p[i + 1] = (unsigned long) invalid\_pte\_table;  
        p[i + 2] = (unsigned long) invalid\_pte\_table;  
        p[i + 3] = (unsigned long) invalid\_pte\_table;  
        p[i + 4] = (unsigned long) invalid\_pte\_table;  
        p[i + 5] = (unsigned long) invalid\_pte\_table;  
        p[i + 6] = (unsigned long) invalid\_pte\_table;  
        p[i + 7] = (unsigned long) invalid\_pte\_table;  
    }  
}  
＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊  
    这个函数只有定义了这个   CONFIG\_HIGHMEM 宏或者这个   CONFIG\_MIPS\_MT\_SMTC 宏  
    才有效，否则为空函数。  
  
void \_\_init fixrange\_init(unsigned long start, unsigned long end,  
    pgd\_t \*pgd\_base)  
{  
#if defined(CONFIG\_HIGHMEM) || defined(CONFIG\_MIPS\_MT\_SMTC)  
    pgd\_t \*pgd;  
    pud\_t \*pud;  
    pmd\_t \*pmd;  
    pte\_t \*pte;  
    int i, j, k;  
    unsigned long vaddr;  
  
    vaddr = start;  
    取得对应于   start 的  PGD 内部索引。  
    i = \_\_pgd\_offset(vaddr);  
    取得对应于   start 的  PUD 内部索引。  
    j = \_\_pud\_offset(vaddr);  
    取得对应于   start 的  PMD 内部索引。  
    k = \_\_pmd\_offset(vaddr);  
    pgd = pgd\_base + i;  
  
    在这种模式下，  
    PTRS\_PER\_PGD = 1024  
    PTRS\_PER\_PUD = 1  
    PTRS\_PER\_PMD = 1，  
    即只有两层页表，没有中间层。  
    所以可以直接使  pud， pmd 指向  PGD 页表，而不用检测   PUD 和  PMD 页表项是否  
    有具体的页面。而是通过指向  pgd 表项的指针，检测  PGD 表项是否有效。  
    这段代码的意思是检测 从  start 地址开始的  PGD 表项是否有效有效，如果无效，  
    而给这个表项分配页面。建立第二级页表。  
  
    一直循环到达  end ，如果  end＝＝ 0，将继续循环直到  PGD 的末端。  
    for ( ; (i < PTRS\_PER\_PGD) && (vaddr != end); pgd++, i++) {  
        强制把   pgd 的地址付给  pud，即使  pud 这个指针指向 pgd 数组中的某一项。  
        pud = (pud\_t \*)pgd;  
        for ( ; (j < PTRS\_PER\_PUD) && (vaddr != end); pud++, j++) {  
        强制把   pud 的地址付给  pmd，即使  pmd 这个指针指向   pud 数组中的某一项。  
            实际还是指向 pgd 数组中的某一项。  
            pmd = (pmd\_t \*)pud;  
            for (; (k < PTRS\_PER\_PMD) && (vaddr != end); pmd++, k++) {  
                    如果  pmd 指针指向的页表项，为无效项。  
                if (pmd\_none(\*pmd)) {  
                        为 PGD 分配一个页面。  
                    pte = (pte\_t \*) alloc\_bootmem\_low\_pages(PAGE\_SIZE);  
                        设置  PGD 指针指向这个页面。  
                    set\_pmd(pmd, \_\_pmd((unsigned long)pte));  
                    if (pte != pte\_offset\_kernel(pmd, 0))  
                        BUG();  
                                           }  
                vaddr += PMD\_SIZE;  
                               }  
            k = 0;  
                     }  
        j = 0;  
           }  
#endif  
}  
＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊  
    在  pagetable\_init（） 函数中对  kmap 地址空间建立的二级页表，  
    并使变量   pkmap\_page\_table 等于   kmap 地址空间二级页表的基地址。  
    这个  kmap\_init（）  函数，仅在编译时配置了  CONFIG\_HIGHMEM 的情况下使用，  
    它负责获取   kmap 地址空间的首地址，引用  kmap 地址空间的  PTE 页表（二级页表），  
    并对它进行保护。  
    这意味着使用  kmap（） 函数映射高端内存时可以，不需要检查  PGD 页目录表。  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
  
pte\_t \*kmap\_pte;  
pgprot\_t kmap\_prot;  
  
static void \_\_init kmap\_init(void)  
{  
    unsigned long kmap\_vstart;  
          
    取得   kmap 地址空间的首地址。  
    kmap\_vstart = \_\_fix\_to\_virt(FIX\_KMAP\_BEGIN);  
    遍历页表取得   kmap 地址空间的首地址对应的  PTE 页表的首个页表项。  
    kmap\_pte = kmap\_get\_fixmap\_pte(kmap\_vstart);  
    设置  页表项的保护项。  
    kmap\_prot = PAGE\_KERNEL;  
}         
＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊  
问题：  
    1）MIPS-cache重影问题的产生和解决办法？  
    2）这个  cpu\_has\_dc\_aliases 变量（实际为宏定义）的使用，它的大小的确定？

**linux-mips启动分析（4-3）**

这个文件主要讲述管理区的初始化。  
  
    初始化各个管理区，建立各个管理区的内核页表。  
    对于  UMA 系统，调用    free\_area\_init（） 函数，  
    对于  NUMA 系统，调用     free\_area\_init\_node（） 函数。  
 这个 free\_area\_init（） 函数和   free\_area\_init\_node（） 函数其实基本相同，  
 也是通过调用   free\_area\_init\_node（） 函数来完成的。就是传入的参数不同。   
  
 free\_area\_init\_node(）函数各个参数的意思：  
    第一个参数：将要初始化的节点标识。  
    第二个参数：节点  struct pglist\_data 结构的指针。  
    第三个参数：各个管理区大小的数组指针。  
    第四个参数：将要初始化的节点的可是  PFN（物理页面震号）。  
    第五个参数：将要初始化的节点内存空间的漏洞。  
＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊  
  
｜--------------------｜  
｜                              ｜  
｜                              ｜      KSEG2 段，这段地址通过 MMU 来转换成物理地址。为1G空间。  
｜                              ｜                         
｜--------------------｜  
｜                              ｜       KSEG1 段，这段地址映射到物理地址空间的  512MB 的低内存段，  
｜--------------------｜                                 为不经过  cache 存取的。  
｜                              ｜       KSEG1 段，这段地址映射到物理地址空间的  512MB 的低内存段，  
｜--------------------｜                                  经过  cache 存取的。  
｜                              ｜  
｜                              ｜  
｜                              ｜       KUSEG 段，这段地址空间为用户地址空间。  
｜                              ｜       它的操作是有特定的  CPU 来定义的。  
｜                              ｜  
｜                              ｜  
｜                              ｜  
｜--------------------｜  
  
void \_\_init free\_area\_init(unsigned long \*zones\_size)  
{  
    对于  MIPS 的  UMA 系统来说，只有一个静态的节点   contig\_page\_data ，  
    这个宏   PAGE\_OFFSET 定义了，内核空间和用户空间的分界，  
    在  MIPS32 的体系结构中，我们可以通过基本的地址空间看出，  
    PAGE\_OFFSET 等于  0x80000000。  
    内核用户空间可以从这里看出为  从   0x80000000 到  0xFFFFFFFF 的  2G 空间。  
  
    free\_area\_init\_node(0, NODE\_DATA(0), zones\_size,  
            \_\_pa(PAGE\_OFFSET) >> PAGE\_SHIFT, NULL);  
}         
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
void \_\_meminit free\_area\_init\_node(int nid, struct pglist\_data \*pgdat,  
        unsigned long \*zones\_size, unsigned long node\_start\_pfn,  
        unsigned long \*zholes\_size)  
{         
    pgdat->node\_id = nid;  
    记录节点的起始物理祯号（PFN）。  
    pgdat->node\_start\_pfn = node\_start\_pfn;  
      
    这个函数，取得这个节点各个管理区所包含的实际页面数，和内存空间漏洞的页面数。  
    calculate\_node\_totalpages(pgdat, zones\_size, zholes\_size);  
  
    为全局的  mem\_map[] 数组分配空间。  
    alloc\_node\_mem\_map(pgdat);  
          
    free\_area\_init\_core(pgdat, zones\_size, zholes\_size);  
}  
＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊  
    这个  calculate\_node\_totalpages（） 函数主要工作是通过两个函数完成的，  
    zone\_spanned\_pages\_in\_node（） 函数和    zone\_absent\_pages\_in\_node（）函数  
    这个       zone\_spanned\_pages\_in\_node（） 函数返回节点一个管理区的跨度所包含的页面数，  
    包含内存空间的漏洞。而要取得这个节点管理区所包含的实际页面数，需要减去这个管理区内存空间的漏洞  
    所包含的页面数。  
    这个    zone\_absent\_pages\_in\_node（）函数返回节点一个管理区所包含的内存空间漏洞页面数。  
  
    在这个  calculate\_node\_totalpages（） 函数内取得了这个节点各个管理区的跨度所包含的页面数，  
    并赋值给这个节点结构体   struct pglist\_data 的   node\_spanned\_pages 成员。  
    把这个管理区所包含的实际页面数赋值给这个节点结构体   struct pglist\_data 的   node\_present\_pages 成员。  
  
static void \_\_meminit calculate\_node\_totalpages(struct pglist\_data \*pgdat,  
        unsigned long \*zones\_size, unsigned long \*zholes\_size)  
{     
    unsigned long realtotalpages, totalpages = 0;  
    enum zone\_type i;  
  
    for (i = 0; i < MAX\_NR\_ZONES; i++)  
        totalpages += zone\_spanned\_pages\_in\_node(pgdat->node\_id, i, zones\_size);  
    pgdat->node\_spanned\_pages = totalpages;  
  
    realtotalpages = totalpages;  
    for (i = 0; i < MAX\_NR\_ZONES; i++)  
        realtotalpages -= zone\_absent\_pages\_in\_node(pgdat->node\_id, i, zholes\_size);  
    pgdat->node\_present\_pages = realtotalpages;  
  
｝  
＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊  
static void \_\_init\_refok alloc\_node\_mem\_map(struct pglist\_data \*pgdat)  
{  
    跳过空的内存节点。  
    if (!pgdat->node\_spanned\_pages)  
        return;  
  
#ifdef CONFIG\_FLAT\_NODE\_MEM\_MAP  
    如果这个节点的   mem\_map 数组不存在，  
    if (!pgdat->node\_mem\_map) {  
        unsigned long size, start, end;  
        struct page \*map;  
          
    节点结构体  struct pglist\_data 的成员  node\_start\_pfn 表示节点的物理地址起始页面祯号（PFN）。  
    这表示  start 的地址必须对齐    (PAGE\_SIZE \* MAX\_ORDER\_NR\_PAGES) ，为什么？  
  
        取得开始这个节点的物理地址起始页面祯号（PFN）。  
        start = pgdat->node\_start\_pfn & ~(MAX\_ORDER\_NR\_PAGES - 1);  
  
        取得开始这个节点的物理地址结束页面祯号（PFN）。  
        这个  pgdat->node\_spanned\_pages 成员在  calculate\_node\_totalpages（） 函数赋值。  
        为包含内存漏洞的总的页面数目。  
        end = pgdat->node\_start\_pfn + pgdat->node\_spanned\_pages;  
        end = ALIGN(end, MAX\_ORDER\_NR\_PAGES);  
  
        取得这个节点的  mem\_map［］ 数组的大小，即内存页面的数目乘以页面结构体的大小。  
        size =  (end - start) \* sizeof(struct page);  
  
        这个  alloc\_remap() 函数，为空函数，只返回   NULL 指针。  
        map = alloc\_remap(pgdat->node\_id, size);  
        if (!map)  
        从这个节点上为  mem\_map［］ 数组分配  size 数量的字节，这个分配将对齐  CPU 的一级高速缓存。  
        为从 静态的节点   contig\_page\_data 节点的第一物理页面开始分配。  
            map = alloc\_bootmem\_node(pgdat, size);  
        pgdat->node\_mem\_map = map + (pgdat->node\_start\_pfn - start);  
            }  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
#ifndef CONFIG\_NEED\_MULTIPLE\_NODES  
    if (pgdat == NODE\_DATA(0)) {  
        mem\_map = NODE\_DATA(0)->node\_mem\_map;  
#ifdef CONFIG\_ARCH\_POPULATES\_NODE\_MAP  
        if (page\_to\_pfn(mem\_map) != pgdat->node\_start\_pfn)  
            mem\_map -= pgdat->node\_start\_pfn;  
#endif /\* CONFIG\_ARCH\_POPULATES\_NODE\_MAP \*/  
           }  
#endif  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
这段代码主要是为全局变量    struct page \*mem\_map   指针赋值。  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
#endif /\* CONFIG\_FLAT\_NODE\_MEM\_MAP \*/  
}  
＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊  
这个   free\_area\_init\_core（） 函数为节点的核心初始化函数。  
这个函数初始化全局的  mem\_map［］ 数组。  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
在  mm/page\_alloc.c 文件头部定义了三个全局变量。  
  
unsigned long \_\_meminitdata nr\_kernel\_pages;  
unsigned long \_\_meminitdata nr\_all\_pages;  
static unsigned long \_\_meminitdata dma\_reserve;  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
static void \_\_meminit free\_area\_init\_core(struct pglist\_data \*pgdat,  
        unsigned long \*zones\_size, unsigned long \*zholes\_size)  
{  
    enum zone\_type j;  
    int nid = pgdat->node\_id;  
    unsigned long zone\_start\_pfn = pgdat->node\_start\_pfn;  
    int ret;  
  
    如果没有定义这个   CONFIG\_MEMORY\_HOTPLUG 宏定义，即不支持内存的热插拔。  
    这个   pgdat\_resize\_init(）   函数为空函数。  
    pgdat\_resize\_init(pgdat);  
  
    默认该节点的管理区数据为  0 ，即没有管理区。  
    pgdat->nr\_zones = 0;  
      
    初始化该节点的页面  swap 等待列表，在内核中，，每个节点都有一个  kswapdN 的交换线程，  
    每个节点都有自己的等待队列。  
    init\_waitqueue\_head(&pgdat->kswapd\_wait);  
    pgdat->kswapd\_max\_order = 0;  
  
    for (j = 0; j < MAX\_NR\_ZONES; j++) {  
        struct zone \*zone = pgdat->node\_zones + j;  
        unsigned long size, realsize, memmap\_pages;  
        这个       zone\_spanned\_pages\_in\_node（） 函数返回节点一个管理区的跨度所包含的页面数，  
        包含内存空间的漏洞。而要取得这个节点管理区所包含的实际页面数，需要减去这个管理区内存空间的漏洞  
        所包含的页面数。  
        size = zone\_spanned\_pages\_in\_node(nid, j, zones\_size);  
  
        这个    zone\_absent\_pages\_in\_node（）函数返回节点一个管理区所包含的内存空间漏洞页面数。  
        realsize = size - zone\_absent\_pages\_in\_node(nid, j, zholes\_size);  
  
        减去这个管理区的   memmap[] 数组，占用的空间，取得实际可用的内存空间大小。  
        memmap\_pages = (size \* sizeof(struct page)) >> PAGE\_SHIFT;  
        if (realsize >= memmap\_pages) {  
            realsize -= memmap\_pages;  
        } else  
            printk(KERN\_WARNING “打印错误提示。”);  
  
        当需要保留  DMA 空间时，保留  DMA 空间。  
        if (j == 0 && realsize > dma\_reserve) {  
            realsize -= dma\_reserve;  
            printk(KERN\_DEBUG " 打印提示信息");  
                     }  
  
        对于非高端内存累加计算到    nr\_kernel\_pages 变量。  
        对于所有内存累加计算到    nr\_all\_pages 变量。  
        if (!is\_highmem\_idx(j))  
            nr\_kernel\_pages += realsize;  
        nr\_all\_pages += realsize;  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
    初始化各个管理区的结构体。  
        spanned\_pages    ：这个管理区包含内存漏洞的，总得页面数。  
        present\_pages    ：这个管理区不包含内存漏洞的，页面数目。  
        node             ：管理区所属的节点号。  
        zone\_pgdat       : 指向节点的指针。  
        lock             ： 并行访问该管理区的自旋锁。  
        lru\_lock         ：管理区的  LRU 链表的自旋锁。  
        active\_list      ： 管理区的活动链表。每个管理区一个。  
        inactive\_list    ： 管理区的非活动链表。每个管理区一个。  
        nr\_scan\_active   ：   active\_list  链表上上的数目。  
        nr\_scan\_inactive ：  inactive\_list  链表上上的数目。  
        vm\_stat          ：   
        reclaim\_in\_progress ：  
wait\_table\_hash\_nr\_entries： 等待队列的个数。  
        wait\_table\_bits  : 等待队列哈希表的大小。  
        wait\_table       ： 等待队列哈希表。  
        zone\_start\_pfn   ： 管理区的起始物理内存页面祯号。  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
        zone->spanned\_pages = size;  
        zone->present\_pages = realsize;  
  
#ifdef CONFIG\_NUMA  
        zone->node = nid;  
        zone->min\_unmapped\_pages = (realsize\*sysctl\_min\_unmapped\_ratio)  
                        / 100;  
        zone->min\_slab\_pages = (realsize \* sysctl\_min\_slab\_ratio) / 100;  
#endif  
        zone->name = zone\_names[j];  
        spin\_lock\_init(&zone->lock);  
        spin\_lock\_init(&zone->lru\_lock);  
        zone\_seqlock\_init(zone);  
        zone->zone\_pgdat = pgdat;  
        zone->prev\_priority = DEF\_PRIORITY;  
        zone\_pcp\_init(zone);  
        INIT\_LIST\_HEAD(&zone->active\_list);  
        INIT\_LIST\_HEAD(&zone->inactive\_list);  
        zone->nr\_scan\_active = 0;  
        zone->nr\_scan\_inactive = 0;  
        对  vm\_stat 成员清  0。  
        zap\_zone\_vm\_stats(zone);  
        atomic\_set(&zone->reclaim\_in\_progress, 0);  
        if (!size)  
            continue;  
  
        ret = init\_currently\_empty\_zone(zone, zone\_start\_pfn, size, MEMMAP\_EARLY);  
        BUG\_ON(ret);  
  
        zone\_start\_pfn += size;  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
        }    这个是   for 循环的结束。  
}  
＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊  
  
\_\_meminit int init\_currently\_empty\_zone(struct zone \*zone,  
                    unsigned long zone\_start\_pfn,  
                    unsigned long size,  
                    enum memmap\_context context)  
{     
    struct pglist\_data \*pgdat = zone->zone\_pgdat;  
    int ret;  
  
    初始化管理区的等待队列哈希表和等待队列头。  
    ret = zone\_wait\_table\_init(zone, size);  
    if (ret)  
        return ret;  
    pgdat->nr\_zones = zone\_idx(zone) + 1;  
      
    赋值该管理区的起始物理内存页面祯号。  
    zone->zone\_start\_pfn = zone\_start\_pfn;  
  
    这个  memmap\_init(） 函数初始化  mem\_map[] 数值。  
    memmap\_init(size, pgdat->node\_id, zone\_idx(zone), zone\_start\_pfn);  
  
    初始化管理区的空闲块链表。  
    zone\_init\_free\_lists(pgdat, zone, zone->spanned\_pages);  
  
    return 0;  
}  
＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊  
这个   zone\_wait\_table\_init(） 函数初始化等待队列哈希表和等待队列头。  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
static noinline \_\_init\_refok  
int zone\_wait\_table\_init(struct zone \*zone, unsigned long zone\_size\_pages)  
{  
    int i;  
    struct pglist\_data \*pgdat = zone->zone\_pgdat;  
    size\_t alloc\_size;  
  
    计算有几个等待队列，最多有  4096 个等待队列。  
    zone->wait\_table\_hash\_nr\_entries = wait\_table\_hash\_nr\_entries(zone\_size\_pages);  
    计算等待队列哈希表的大小。  
    zone->wait\_table\_bits = wait\_table\_bits(zone->wait\_table\_hash\_nr\_entries);  
    alloc\_size = zone->wait\_table\_hash\_nr\_entries  \* sizeof(wait\_queue\_head\_t);  
  
    分配哈希表空间。  
    if (system\_state == SYSTEM\_BOOTING) {  
        zone->wait\_table = (wait\_queue\_head\_t \*) alloc\_bootmem\_node(pgdat, alloc\_size);  
    } else {  
        zone->wait\_table = (wait\_queue\_head\_t \*)vmalloc(alloc\_size);  
          }  
    if (!zone->wait\_table)  
        return -ENOMEM;  
      
    初始化各个等待队列。  
    for(i = 0; i < zone->wait\_table\_hash\_nr\_entries; ++i)  
        init\_waitqueue\_head(zone->wait\_table + i);  
  
    return 0;  
}  
＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊  
这个  memmap\_init(） 函数初始化  mem\_map[] 数值。  
设置这个管理区页面为空闲状态，引用计数为 1。初始化  LRU 链表头。  
并设置    page->flags 标志，标记页面属于那个节点，那个管理区，那个  section。  
  
#ifndef \_\_HAVE\_ARCH\_MEMMAP\_INIT  
#define memmap\_init(size, nid, zone, start\_pfn) \  
    memmap\_init\_zone((size), (nid), (zone), (start\_pfn), MEMMAP\_EARLY)  
#endif  
  
移植相关：如果没有定义这个   \_\_HAVE\_ARCH\_MEMMAP\_INIT 宏，则使用公共的    memmap\_init（） 函数。  
        一般是否定义这个宏，由特定的体系结构决定。  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
void \_\_meminit memmap\_init\_zone(unsigned long size, int nid, unsigned long zone,  
        unsigned long start\_pfn, enum memmap\_context context)  
{  
    struct page \*page;  
    unsigned long end\_pfn = start\_pfn + size;  
    unsigned long pfn;  
      
    for (pfn = start\_pfn; pfn < end\_pfn; pfn++) {  
        if (context == MEMMAP\_EARLY) {  
  
    如果没有定义   CONFIG\_SPARSEMEM 这个宏，这个  early\_pfn\_valid(） 、  
        和 early\_pfn\_in\_nid(） 函数一直返回  TRUE。  
            if (!early\_pfn\_valid(pfn))  
                continue;  
            if (!early\_pfn\_in\_nid(pfn, nid))  
                continue;  
                     }  
  
    取得  mem\_map[] 数组中的   struct page 结构体指针。  
        page = pfn\_to\_page(pfn);  
  
    设置  page->flags 标志，标记页面属于那个节点，那个管理区，那个  section。  
        set\_page\_links(page, zone, nid, pfn);  
  
    设置页面被应用的数目，初始化为  1，当为 0 时，页面被释放。  
        init\_page\_count(page);  
  
    设置  \_mapcount 成员为  －1。  
        reset\_page\_mapcount(page);  
    设置这个页面为空闲状态。  
        SetPageReserved(page);  
    初始化   LRU 链表的头。  
        INIT\_LIST\_HEAD(&page->lru);  
  
#ifdef WANT\_PAGE\_VIRTUAL  
        这个  set\_page\_address(） 和     page\_address\_init() 函数一样，参考《linux-mips启动分析（2）》。  
        当定义了 CONFIG\_HIGHMEM  宏，并且没有定义   WANT\_PAGE\_VIRTUAL  宏时，非空函数。  
           其他情况为空函数。  
        if (!is\_highmem\_idx(zone))  
            set\_page\_address(page, \_\_va(pfn << PAGE\_SHIFT));  
#endif  
        }  
}  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
static inline void set\_page\_links(struct page \*page, enum zone\_type zone,  
    unsigned long node, unsigned long pfn)  
{  
    set\_page\_zone(page, zone);  
    set\_page\_node(page, node);  
    set\_page\_section(page, pfn\_to\_section\_nr(pfn));  
}    
＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊  
这个函数是初始化管理区的空闲块链表。  
每个管理区有一个  struct free\_area  free\_area[MAX\_ORDER]，保存着  zone 中的空闲块。  
struct free\_area {  
    struct list\_head    free\_list;  
    unsigned long       nr\_free;  
};  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
void zone\_init\_free\_lists(struct pglist\_data \*pgdat, struct zone \*zone,  
                unsigned long size)  
{  
    int order;  
    for (order = 0; order < MAX\_ORDER ; order++) {  
        INIT\_LIST\_HEAD(&zone->free\_area[order].free\_list);  
        zone->free\_area[order].nr\_free = 0;  
          }  
}  
＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊  
问题：  
    1）在这个  alloc\_node\_mem\_map（） 函数中，分配   mem\_map 数组时，为什么  start 要  
          和      (PAGE\_SIZE \* MAX\_ORDER\_NR\_PAGES)  地址必须对齐 ？  
    2）  MAX\_ORDER\_NR\_PAGES 宏定义的意义和大小的确定？  
    3）有  alloc\_bootmem\_node（）函数返回的地址为虚拟地址，为什么直接赋值给  pgdat->node\_mem\_map ？  
    4）为什么要加上   (pgdat->node\_start\_pfn - start)，在赋值给   pgdat->node\_mem\_map ？  
＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊  
  
参考：  
mem\_map:一个Struct page数组，对应系统中所有的物理内存页。  
而每一个zone结构里都有一个zone\_mem\_map  域指向这个zone的第一个page 在mem\_map的位置，  
还有一个域size代表这个区的大小，即总共有多少页。  
  
每一个zone都有自己的buddy system,由下面的zone结构就可以看出。  
空闲块是根据其大小做的保存，特别强调的是struct free\_area  free\_area[MAX\_ORDER];  
保存着zone中的空闲块。数组中的每一个元素都有个双链表结构。比如说     free\_area中第  K  个元素  
保存着大小为  2  的k次方大小的块的链表结构。数组中保存的是表头结构，即指向第一个2的k次方大小块的  
第一个页面。那块的剩余的页面怎么办？不用管，因为都是按块来操作的，只需要知道块的第一个页面即可，  
最后一个页面就是第一个页面加上2的k次方。同属于一个链表的块与块之间由每一个块的第一个  
页面  的    struct page 中的    list\_head lru  来相互链接。  
  
#ifndef CONFIG\_FORCE\_MAX\_ZONEORDER  
#define MAX\_ORDER 11  
#else  
#define MAX\_ORDER CONFIG\_FORCE\_MAX\_ZONEORDER  
#endif

#define MAX\_ORDER\_NR\_PAGES (1 << (MAX\_ORDER - 1))  
  
struct free\_area {  
       struct list\_head  free\_list;  
       unsigned long            nr\_free;  
};  
＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊

**linux-mips启动分析（4-4）**

    这个文件主要讲述引导内存分配器的内存分配函数。  
    主要  API 函数：  
        alloc\_bootmem\_node(）  
        alloc\_bootmem\_pages\_node（）  
        alloc\_bootmem\_low\_pages\_node（）  
＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊  
总结：  
    在第一次分配  mem\_map[] 数组时， 参数为静态节点   contig\_page\_data ，  
分配空间大小为  size ，对齐方式为 SMP\_CACHE\_BYTES。  
从     ZONE\_NORMAL 处的第一页开始分配。  
＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊  
  
    这个   alloc\_bootmem\_node(） 函数为引导内存分配器的内存分配  API。  
    这个  \_\_alloc\_bootmem\_core（） 函数为引导内存分配器的核心内存分配函数。  
  
    这个  alloc\_bootmem\_node(） 函数在指定的节点上开始分配内存，从  ZONE\_NORMAL 处的  
    第一页开始分配  size 数量的字节，这个分配将对齐  CPU 的一级高速缓存，充分利用硬件的高速缓存。  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
#define alloc\_bootmem\_node(pgdat, x) \  
    \_\_alloc\_bootmem\_node(pgdat, x, SMP\_CACHE\_BYTES, \_\_pa(MAX\_DMA\_ADDRESS))  
  
这个宏  SMP\_CACHE\_BYTES 定义决定了分配内存的对齐字节，在这里对齐  CPU 的一级高速缓存。  
这个  MAX\_DMA\_ADDRESS 宏定义决定了从什么地方开始分配内存，在这里  
从   ZONE\_NORMAL 处的第一页开始分配。  
＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊  
  
void \* \_\_init \_\_alloc\_bootmem\_node(pg\_data\_t \*pgdat, unsigned long size,  
                   unsigned long align, unsigned long goal)  
{  
    void \*ptr;  
  
    ptr = \_\_alloc\_bootmem\_core(pgdat->bdata, size, align, goal, 0);  
    if (ptr)  
        return ptr;  
  
    return \_\_alloc\_bootmem(size, align, goal);  
}  
＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊  
这个  \_\_alloc\_bootmem\_core（） 函数为引导内存分配器的核心内存分配函数分析，这个函数是不可重载的。  
  
参数如下：  
    1）要分配结构体的启动内存。  
    2）请求分配的大小。  
    3）分配内存的对齐方式，它必须是  2  的冥数。  
    4）分配内存的开始地址。  
    5）分配内存的最大的结束地址，如果 ＝＝ 0，可以任意，如果不为 0，则不能超过这个地址。  
  
这个函数分为四部分：  
    1）参数检测，保证参数的正确性。  
    2）以  goal 参数为基础计算并扫描开始地址。  
    3）检测本次分配是否可以使用上次分配使用的页面，以节省内存。  
    4）在引导内存分配器的“位图”中，标记页面已分配，并将页面内存清  0。  
  
void \* \_\_init  
\_\_alloc\_bootmem\_core(struct bootmem\_data \*bdata, unsigned long size,  
          unsigned long align, unsigned long goal, unsigned long limit)  
{  
    unsigned long offset, remaining\_size, areasize, preferred;  
    unsigned long i, start = 0, incr, eidx, end\_pfn;  
    void \*ret;  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
    检查   size 参数，不能为  0。  
    if (!size) {  
        BUG();  
          }  
    检查  align 参数，如果对齐不是 2  的幂数，则 BUG（）。  
    BUG\_ON(align & (align-1));  
  
    如果这个节点的起始地址，大于  limit 并且 limit 不为  0，则返回报错。  
    if (limit && bdata->node\_boot\_start >= limit)  
        return NULL;  
  
    如果节点没有页面位图，则报错，返回  NULL。  
    if (!bdata->node\_bootmem\_map)  
        return NULL;  
  
    把这个节点（低端内存）的结束  PFN。  
    end\_pfn = bdata->node\_low\_pfn;  
    取得  limit 地址所对应的物理页面祯号。  
    limit = PFN\_DOWN(limit);  
      
    if (limit && end\_pfn > limit)  
        end\_pfn = limit;  
  
    eidx = end\_pfn - PFN\_DOWN(bdata->node\_boot\_start);  
  
    对齐方式默认为  0 。  
    offset = 0;  
      
    如果指定了分配内存的对齐方好，计算对齐方式。  
    if (align && (bdata->node\_boot\_start & (align - 1UL)) != 0)  
        offset = align - (bdata->node\_boot\_start & (align - 1UL));  
  
    offset = PFN\_DOWN(offset);  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
  
    如果设置了分配空间的目标地址，目标地址在这个节点的空间内。  
    if (goal && goal >= bdata->node\_boot\_start && PFN\_DOWN(goal) < end\_pfn) {  
    开始的适当偏移是  goal 减去该节点可寻址的内存起点。  
        preferred = goal - bdata->node\_boot\_start;  
        考虑到上次分配的地址。  
        if (bdata->last\_success >= preferred)  
            if (!limit || (limit && limit > bdata->last\_success))  
                preferred = bdata->last\_success;  
    } else  
        preferred = 0;  
  
    上面这段代码确定了 preferred 这个变量的值，这个值确定了内存分配的地址，  
    下面设置了这个地址对齐设置。  
  
    preferred = PFN\_DOWN(ALIGN(preferred, align)) + offset;  
    对要分配的内存空间大小进行调整，进行整页分配。  
    areasize = (size + PAGE\_SIZE-1) / PAGE\_SIZE;  
    这个 incr 变量是要跳过的页面数，如果要求对齐方式多于一页，则他们满足对齐请求。  
    incr = align >> PAGE\_SHIFT ? : 1;  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
  
restart\_scan:  
    for (i = preferred; i < eidx; i += incr) {  
        unsigned long j;  
        i = find\_next\_zero\_bit(bdata->node\_bootmem\_map, eidx, i);  
        i = ALIGN(i, incr);  
        if (i >= eidx)  
            break;  
        if (test\_bit(i, bdata->node\_bootmem\_map))  
            continue;  
        for (j = i + 1; j < i + areasize; ++j) {  
            if (j >= eidx)  
                goto fail\_block;  
            if (test\_bit(j, bdata->node\_bootmem\_map))  
                goto fail\_block;  
                     }  
        start = i;  
        goto found;  
    fail\_block:  
        i = ALIGN(j, incr);  
          }  
  
    如果不能够从  goal 以开始分配足够的空间，直接从最低端的对齐地址，重新进行扫描。  
    if (preferred > offset) {  
        preferred = offset;  
        goto restart\_scan;  
          }  
    return NULL;  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
这段代码在页面   preferred 和这个节点的结束（或者 limit）之间扫描地址空间，  
当需要对齐大于 1  页时，步长以  incr 为准。  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
  
found:  
    记录这次分配的起始地址。  
    bdata->last\_success = PFN\_PHYS(start);  
    BUG\_ON(start >= eidx);  
  
    bdata->last\_offset 表示最后一次分配所在页面的偏移，如果为  0， 则表示该页全部使用。  
    bdata->last\_pos 表示最后一次分配时的物理页面祯数。  
    如果这次空间分配可以和上面的分配进行合并。  
    if (align < PAGE\_SIZE && bdata->last\_offset && bdata->last\_pos+1 == start) {  
        offset = ALIGN(bdata->last\_offset, align);  
        BUG\_ON(offset > PAGE\_SIZE);  
  
        上次分配之后，这个页面还有多少内存空间（经过对齐的）。  
        remaining\_size = PAGE\_SIZE - offset;  
  
        如果要分配的空间大小，小于这个页面还没有分配的空间大小。  
        if (size < remaining\_size) {  
            areasize = 0;  
            /\* last\_pos unchanged \*/  
            bdata->last\_offset = offset + size;  
            ret = phys\_to\_virt(bdata->last\_pos \* PAGE\_SIZE +  
                       offset +  
                       bdata->node\_boot\_start);  
        } else {  
        这个页面的空间不满足这次的空间分配，分配下一页。  
            remaining\_size = size - remaining\_size;  
            areasize = (remaining\_size + PAGE\_SIZE-1) / PAGE\_SIZE;  
            ret = phys\_to\_virt(bdata->last\_pos \* PAGE\_SIZE +  
                       offset +  
                       bdata->node\_boot\_start);  
            bdata->last\_pos = start + areasize - 1;  
            bdata->last\_offset = remaining\_size;  
                    }  
        bdata->last\_offset &= ~PAGE\_MASK;  
    } else {  
    如果这个内存空间分配，不可以和上次的空间分配进行合并。  
        bdata->last\_pos = start + areasize - 1;  
        bdata->last\_offset = size & ~PAGE\_MASK;  
        ret = phys\_to\_virt(start \* PAGE\_SIZE + bdata->node\_boot\_start);  
          }  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
  
这段代码主要测试，这次的内存空间地址分配是否能够和上次的地址空间分配进行合并。  
并对      bdata->last\_offset  和       bdata->last\_pos 进行更新。  
  
    bdata->last\_offset 表示最后一次分配所在页面的偏移，如果为  0， 则表示该页全部使用。  
    bdata->last\_pos 表示最后一次分配时的物理页面祯数。  
  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
    for (i = start; i < start + areasize; i++)  
        if (unlikely(test\_and\_set\_bit(i, bdata->node\_bootmem\_map)))  
            BUG();  
    memset(ret, 0, size);  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
把这次分配的页面在引导内存分配器的位图上表为  1  ，并将其内容清 0。  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
    return ret;  
}  
＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊  
问题：  
    1）在这个  alloc\_node\_mem\_map（） 函数中，分配   mem\_map 数组时，为什么  start 要  
          和      (PAGE\_SIZE \* MAX\_ORDER\_NR\_PAGES)  地址必须对齐 ？  
    2）  MAX\_ORDER\_NR\_PAGES 宏定义的意义和大小的确定？  
    3）在  \_\_alloc\_bootmem\_core（） 函数中   BUG\_ON(align & (align-1)) 的意思？  
＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊  
参考：  
mem\_map:一个Struct page数组，对应系统中所有的物理内存页。  
  
而每一个zone结构里都有一个zone\_mem\_map域指向这个zone的第一个page 在mem\_map的位置，  
还有一个域size代表这个区的大小，即总共有多少页。  
  
每一个zone都有自己的buddy system,由下面的zone结构就可以看出。  
  
空闲块是根据其大小做的保存，特别强调的是struct free\_area  free\_area[MAX\_ORDER];  
  
保存着zone中的空闲块。数组中的每一个元素都有个双链表结构。比如说     free\_area中第  K  个元素  
保存着大小为2的k次方大小的块的链表结构。数组中保存的是表头结构，即指向第一个2的k次方大小块的  
第一个页面。那块的剩余的页面怎么办？不用管，因为都是按块来操作的，只需要知道块的第一个页面即可，  
最后一个页面就是第一个页面加上2的k次方。同属于一个链表的块与块之间由每一个块的第一个  
页面  的    struct page 中的    list\_head lru  来相互链接。  
  
#ifndef CONFIG\_FORCE\_MAX\_ZONEORDER  
#define MAX\_ORDER 11  
#else  
#define MAX\_ORDER CONFIG\_FORCE\_MAX\_ZONEORDER  
#endif  
  
#define MAX\_ORDER\_NR\_PAGES (1 << (MAX\_ORDER - 1))  
  
struct free\_area {  
       struct list\_head  free\_list;  
       unsigned long            nr\_free;  
};

**linux-mips启动分析（4-5）**

在   mem\_init()  函数中调用   free\_all\_bootmem() 函数，取得内存启动分配器的空闲页面数目。  
在我们的系统中，为  UMA 内存模型，只有一个内存节点，为静态内存节点  contig\_page\_data 。  
  
这个     free\_all\_bootmem() 函数销毁引导内存分配器。  
==============================================================  
unsigned long \_\_init free\_all\_bootmem(void)  
{  
    return free\_all\_bootmem\_core(NODE\_DATA(0));  
}  
＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊  
这个  free\_all\_bootmem\_core() 函数为销毁引导内存分配器的核心函数。  
它分为如下的步骤：  
1）   对该节未分配的页面  
    清除  struct page 结构的  PG\_reserved 标志。  
    设置  struct page 结构引用技术为 1。  
    调用  \_\_free\_pages() 函数构建伙伴分配的空闲页面链表。  
  
2）  释放用于位图的页面，并将其释放给伙伴分配器。  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
static unsigned long \_\_init free\_all\_bootmem\_core(pg\_data\_t \*pgdat)  
{  
    struct page \*page;  
    unsigned long pfn;  
  
    取得内存映射结构体。这个结构体用于引导内存分配器分配内存时使用。  
    这个    bootmem\_data\_t 类型参考《linux-mips启动分析（4-1）》。  
    bootmem\_data\_t \*bdata = pgdat->bdata;  
    unsigned long i, count, total = 0;  
    unsigned long idx;  
    unsigned long \*map;  
    int gofast = 0;  
  
    BUG\_ON(!bdata->node\_bootmem\_map);  
    count = 0;  
  
    这个  node\_boot\_start 成员变量表示这个节点的起始物理地址。  
    取得这个内存节点的起始物理页面祯号（PFN）。  
    pfn = PFN\_DOWN(bdata->node\_boot\_start);  
    这个   node\_low\_pfn 成员变量表示这个节点的结束   PFN 。  
    idx = bdata->node\_low\_pfn - pfn;  
    这个  node\_bootmem\_map 是这个内存节点页面映射位图。  
    map = bdata->node\_bootmem\_map;  
  
    由于    node\_bootmem\_map 是  long 类型的，当第二个条件成立时，  
    也就是指  node\_boot\_start ，这个内存节点的起始地址，是  LOG2(BITS\_PER\_LONG) 个页面对齐的。  
    也就是当这个节点内存的起始地址为  0 ，或者 LOG2(BITS\_PER\_LONG) 个页面对齐时， gofast = 1。  
    if (bdata->node\_boot\_start == 0 ||    
                    ffs(bdata->node\_boot\_start)-PAGE\_SHIFT>ffs(BITS\_PER\_LONG))  
        gofast = 1;  
    for (i = 0; i < idx; ) {  
  
        取得内存页面  i 所在的页面映射位图占用的  long 字。  
        unsigned long v = ~map[i / BITS\_PER\_LONG];  
  
        如果  gofast 等于  1 ，并且这个    long 字的位全部为  1  ，  
        表明这个    long 字表示未分配的页面，可以快速的设置。  
        if (gofast && v == ~0UL) {  
            int order;  
  
            这个  page 变量为这个  long 字的起始的第一个  struct page 的指针。  
            这个   struct page mem\_map[] 数组是在   free\_area\_init\_core()函数中初始化的。  
            参考《linux-mips启动分析（4-3）》。  
            page = pfn\_to\_page(pfn);  
  
            使用  count 变量进行统计所有的未分配页面的数量。  
            count += BITS\_PER\_LONG;  
  
            这个  order 变量表示内存空间的为  2  的  order 次页面数目。  
            order = ffs(BITS\_PER\_LONG) - 1;  
            \_\_free\_pages\_bootmem(page, order);  
            i += BITS\_PER\_LONG;  
            page += BITS\_PER\_LONG;  
  
        当这个    long 字的位并不全部为  1  时，也不全部为  0 时。  
        } else if (v) {  
            unsigned long m;  
  
            这个  page 变量为这个  long 字的起始的第一个  struct page 的指针。  
            page = pfn\_to\_page(pfn);  
  
            循环测试，当  v 的某一位为 1 时，表明这个页面存在并且没有分配。  
            for (m = 1; m && i < idx; m<<=1, page++, i++) {  
                if (v & m) {  
                    count++;  
                    \_\_free\_pages\_bootmem(page, 0);  
                                          }  
                                }  
  
        当这个    long 字的位不全部为  0 时。  
        } else {  
            i += BITS\_PER\_LONG;  
                     }  
  
        这个  pfn 变量，用于  pfn\_to\_page(pfn) 取得每一个  long 字的起始的第一个  struct page 的指针。  
        pfn += BITS\_PER\_LONG;  
           }  
  
    使用  total 变量计数所有可以分配的页面总数。  
    total += count;  
  
  
    page = virt\_to\_page(bdata->node\_bootmem\_map);  
    count = 0;  
  
    取得位图所占用的内存页面数量。  
    idx = (get\_mapsize(bdata) + PAGE\_SIZE-1) >> PAGE\_SHIFT;  
  
    循环遍历位图所占用的页面，进行释放。  
    for (i = 0; i < idx; i++, page++) {  
        \_\_free\_pages\_bootmem(page, 0);  
        count++;  
           }  
    使用  total 变量计数所有可以分配的页面总数。  
    total += count;  
      
    设置引导内存分配器的位图等于  NULL。  
    bdata->node\_bootmem\_map = NULL;  
  
    返回空闲内存页面的总数。  
    return total;  
}  
＊＊＊\*＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊  
这个  \_\_free\_pages\_bootmem() 函数建立伙伴分配器的空闲页面的空闲页面链表。  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
void fastcall \_\_init \_\_free\_pages\_bootmem(struct page \*page, unsigned int order)  
{  
  
    这个  order 表示  
    如果等于 0 ，则清除  struct  page 结构体的  PG\_reserved 标志。引用计数设为 1。  
    if (order == 0) {  
        \_\_ClearPageReserved(page);  
        set\_page\_count(page, 0);  
        set\_page\_refcounted(page);  
  
        这个  \_\_free\_page() 函数建立伙伴分配器的空闲链表。  
        \_\_free\_page(page);  
    } else {  
  
        int loop;  
        prefetchw(page);  
          
        循环清除  page 结构体的  PG\_reserved 标志。引用计数设为   0  。  
        for (loop = 0; loop < BITS\_PER\_LONG; loop++) {  
            struct page \*p = &page[loop];  
  
            if (loop + 1 < BITS\_PER\_LONG)  
                prefetchw(p + 1);  
            \_\_ClearPageReserved(p);  
            set\_page\_count(p, 0);  
                     }  
          
        设置  struct page 结构体的引用计数为  1 。  
        set\_page\_refcounted(page);  
  
        这个  \_\_free\_pages() 函数建立伙伴分配器的空闲链表。  
        \_\_free\_pages(page, order);  
          }  
}  
＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊  
这个     \_\_free\_page() 或者  \_\_free\_pages() 函数的  
实现参考《linux-mips启动分析（21-1）》。  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
#define \_\_free\_page(page) \_\_free\_pages((page), 0)  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
fastcall void \_\_free\_pages(struct page \*page, unsigned int order)   
{  
  
    这个  put\_page\_testzero() 函数，把  page 的计数原子性的减 1 ，  
    并测试是否为 0 ，如果为 0 ，返回  true，否则返回  false。  
    if (put\_page\_testzero(page)) {  
        if (order == 0)  
            free\_hot\_page(page);  
        else  
            \_\_free\_pages\_ok(page, order);  
           }  
}  
＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊  
问题：  
    1）   在    \_\_free\_pages\_bootmem() 函数中，当  order 不等于  0  时，  
           为什么只设置一个   page 的引用计数为 1 ，其他的为  0 ？

**linux-mips启动分析（5）**

在  setup\_arch（） 函数中调用了 resource\_init() 函数。  
  
这个函数遍历每一个内存空间范围（物理地址），在资源管理器中进行资源申请，  
并对内核代码和数据段进行资源申请。  
==============================================================  
  
static void \_\_init resource\_init(void)  
{  
    int i;  
      
    这个宏    UNCAC\_BASE 和   IO\_BASE 是在  include/asm-mips/page.h  文件中定义的。  
    表示不经过  cache 的内存的起始地址和   IO mem 的起始地址。  
    if (UNCAC\_BASE != IO\_BASE)  
        return;  
      
    对将要申请的资源进行赋值。（\_text，\_etext 等由链接文件中得出。）  
    变量  code\_resource 和  data\_resource 是静态定义的。  
    code\_resource.start = \_\_pa\_symbol(&\_text);  
    code\_resource.end = \_\_pa\_symbol(&\_etext) - 1;  
    data\_resource.start = \_\_pa\_symbol(&\_etext);  
    data\_resource.end = \_\_pa\_symbol(&\_edata) - 1;  
  
    for (i = 0; i < boot\_mem\_map.nr\_map; i++) {  
        struct resource \*res;  
        unsigned long start, end;  
  
        start = boot\_mem\_map.map[i].addr;  
        end = boot\_mem\_map.map[i].addr + boot\_mem\_map.map[i].size - 1;  
        if (start >= HIGHMEM\_START)  
            continue;  
        if (end >= HIGHMEM\_START)  
            end = HIGHMEM\_START - 1;  
      
    动态分配   struct resource 结构，每个内存空间范围（物理地址）对应一个  struct resource 结构。  
        res = alloc\_bootmem(sizeof(struct resource));  
        switch (boot\_mem\_map.map[i].type) {  
        case BOOT\_MEM\_RAM:  
        case BOOT\_MEM\_ROM\_DATA:  
            res->name = "System RAM";  
            break;  
        case BOOT\_MEM\_RESERVED:  
        default:  
            res->name = "reserved";  
       }    switch 选择结束。  
  
        res->start = start;  
        res->end = end;  
        res->flags = IORESOURCE\_MEM | IORESOURCE\_BUSY;  
  
    这个   request\_resource(）  在资源管理器上申请资源。  
    把每一个内存空间范围（物理地址）都在根资源   iomem\_resource  中申请。  
    根资源    iomem\_resource 是在   plat\_mem\_setup（） 函数中进行的赋值。  
        request\_resource(&iomem\_resource, res);  
  
    因为不知道内核代码和数据方在哪个内存空间范围（物理地址）中，所以都进行申请，由资源管理器来测试。  
        request\_resource(res, &code\_resource);  
        request\_resource(res, &data\_resource);  
       }   循环结束。  
}  
＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊  
进行  I/O 或者   MEM 申请都是由  request\_resource(） 来完成的。  
资源管理器的根资源   I/O 或者   MEM 分别不同。  
    I/O 的根资源为    ioport\_resource。  
    MEM 的根资源为   iomem\_resource。  
这两种资源在   plat\_mem\_setup（） 函数中进行的赋值。  
  
这个    request\_resource(） 函数的参数为：  
root 表示上一级的资源结构。  new 表示新添加的资源。  
这个函数主要功能还是有   \_\_request\_resource（） 函数来实现的。  
==============================================================  
  
这个   resource\_lock 锁定所有的资源链表，在申请或者移出资源时，需要锁定这个锁。  
在  kernel/resource.c 文件进行定义。  
static DEFINE\_RWLOCK(resource\_lock);  
  
int request\_resource(struct resource \*root, struct resource \*new)  
{  
    struct resource \*conflict;  
  
    write\_lock(&resource\_lock);  
    conflict = \_\_request\_resource(root, new);  
    write\_unlock(&resource\_lock);  
    return conflict ? -EBUSY : 0;  
}  
＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊  
  
资源的组织方式，在  struct resource 结构体中有三个指针，  
->parent    指向父资源的指针。  
->child     指向子资源的指针。  
->sibling   指向兄弟资源的指针。  
  
  --------------------- /     parent指针  
  | iomem\_resource  | -------------------  
  --------------------- \          ｜  
             /\         | ->child 指向子资源的指针   ｜  
        parent指针 ｜           \/                                                                                          ｜  
  ----------------------  sibling 指针       -------------------   sibling 指针  
  | new\_resource\_1  | ---------------------------|  new\_resource\_2  | －－－－－－－－  
  -------------------                                           -------------------  
                   /\        | ->child 指向子资源的指针                        
        parent指针｜           \/          
    -------------------   sibling 指针       -------------------   sibling 指针  
   | new\_resource\_3  | ---------------------------| new\_resource\_4  | －－－－－－－－  
     -------------------                                     -------------------  
   | ->child 指向子资源的指针  
     \/  
  
各个兄弟资源组成一个链表，从  root->child 到依次的  sibling 资源，  
它们的各个资源的  I/O 或者  MEM 空间也是依次的从低到高的。  
例如    new\_resource\_1 的  start ＝  0x00000000， end ＝ 0x000000FF。  
      new\_resource\_2 的  start 必须大于   0x000000FF。  
    new\_resource\_3 和   new\_resource\_4 的空间必须在从   
   start ＝  0x00000000， end ＝ 0x000000FF 之间。  
===============================================================  
  
static struct resource \* \_\_request\_resource(struct resource \*root, struct resource \*new)  
{  
    resource\_size\_t start = new->start;  
    resource\_size\_t end = new->end;  
    struct resource \*tmp, \*\*p;  
  
    首先进行参数检测，如果错误则返回   root 指针。  
    if (end < start)  
        return root;  
    if (start < root->start)  
        return root;  
    if (end > root->end)  
        return root;  
  
    取得指向  root->child 指针的指针。  
    p = &root->child;  
    for (;;) {  
        取得     root->child 指针 ＝＝ tmp 。  
        tmp = \*p;  
        如果   root->child 指针是  NULL 指针。或者新资源空间块在    root->child 资源块之下。  
        if (!tmp || tmp->start > end) {  
            new->sibling = tmp;  
            \*p = new;  
            new->parent = root;  
            return NULL;  
                     }  
        取得   root->child 的兄弟资源指针。  
        p = &tmp->sibling;  
  
        如果新资源和    root->child  资源没有冲突，则继续循环。  
        if (tmp->end < start)  
            continue;  
        返回有冲突的资源指针。  
        return tmp;  
        }  
}

**linux-mips启动分析（6）**

在  start\_kernel（） 函数中调用了  sched\_init() 函数。  
这个函数进行核心进程调度器初始化，调度器的初始化优先于任何中断  
的建立（包括  timer 中断）。  
这个函数函数的主要工作是初始化各个  CPU 的运行队列。  
并且初始化进程 0 ，即  idle 进程，但是并没有设置 idle 进程的  NEED\_RESCHED 标志，  
以完成内核剩余的启动部分。  
  
下面对进程调度器进行初始化函数   sched\_init() 函数进行分析。  
================================================================  
  
void \_\_init sched\_init(void)  
{  
    int i, j, k;  
    int highest\_cpu = 0;  
  
    for\_each\_possible\_cpu(i) {  
        struct prio\_array \*array;  
        struct rq \*rq;  
  
    这个 cpu\_rq(i) 函数取得索引为  n 的  CPU 的运行队列的地址。   
        rq = cpu\_rq(i);  
  
    锁定这个运行队列保护进程链表的自选锁。  
        spin\_lock\_init(&rq->lock);  
  
    设置锁的依赖关系。  
        lockdep\_set\_class(&rq->lock, &rq->rq\_lock\_key);  
        rq->nr\_running = 0;  
  
    这个  active 指针表示指向活动进程链表的指针。  
    这个  expired 指针表示指向已过时进程链表的指针。  
        rq->active = rq->arrays;  
        rq->expired = rq->arrays + 1;  
  
    这个   best\_expired\_prio 成员表示过期进程中静态优先级最高的进程。  
        rq->best\_expired\_prio = MAX\_PRIO;  
  
#ifdef CONFIG\_SMP  
    如果定义了   CONFIG\_SMP 这个宏定义，则执行下面的。  
    这些成员的主要意义是保持各个 CPU 运行链表的负载平衡使用。  
        rq->sd = NULL;  
        for (j = 1; j < 3; j++)  
            rq->cpu\_load[j] = 0;  
        rq->active\_balance = 0;  
        rq->push\_cpu = 0;  
        rq->cpu = i;  
        rq->migration\_thread = NULL;  
        INIT\_LIST\_HEAD(&rq->migration\_queue);  
#endif  
        atomic\_set(&rq->nr\_iowait, 0);  
  
    这个   array 成员是一个包含两个  prio\_array\_t 结构的数组。  
    每个数据结构表示一个进程的集合，包含  140 个双向链表头，  
    每个链表表示一个可能的进程优先级，一个优先级位图和一个计数器。  
    这个位图中置  1， 表示这个优先级的链表中有进程。  
    这个计数器表示这个进程组合中共有多少进程。  
        for (j = 0; j < 2; j++) {  
            array = rq->arrays + j;  
            for (k = 0; k < MAX\_PRIO; k++) {  
                INIT\_LIST\_HEAD(array->queue + k);  
                \_\_clear\_bit(k, array->bitmap);  
                                }  
        把   MAX\_PRIO ＝ 140 为置  1， 定义查找    bitmap 的终止符。  
            \_\_set\_bit(MAX\_PRIO, array->bitmap);  
                     }  
        highest\_cpu = i;  
          }  
  
    设置进程  0 的  load\_weight 成员。  
    set\_load\_weight(&init\_task);  
  
#ifdef CONFIG\_SMP  
    设置  CPU 的总数目。  
    nr\_cpu\_ids = highest\_cpu + 1;  
    调用  open\_softirq()函数开启使用软中断向量  SCHED\_SOFTIRQ 。  
    设置它的软中断处理函数指针为  run\_rebalance\_domains 函数。  
    open\_softirq(SCHED\_SOFTIRQ, run\_rebalance\_domains, NULL);  
#endif  
  
#ifdef CONFIG\_RT\_MUTEXES  
    初始化每个 mutex 的最高优先级的任务的队列头和队列锁。  
    plist\_head\_init(&init\_task.pi\_waiters, &init\_task.pi\_lock);  
#endif  
    原子性增加  init\_mm 的引用计数，初始化为 1， 当这个成员为  0   时，则释放对这段空间。  
    atomic\_inc(&init\_mm.mm\_count);  
  
    如果  进程  0  （idle进程）也进行缓慢的  MMU 转换，则这个函数不为空函数，  
    否则是空函数，在  MIPS 体系结构下，为空函数。  
    enter\_lazy\_tlb(&init\_mm, current);  
  
    初始化指定  CPU （当前 CPU）的  idle 进程。  
    这个 idle 进程只有当这个  CPU 的可运行进程队列为空时，才会被调度。  
    init\_idle(current, smp\_processor\_id());  
}  
＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊  
这个  for\_each\_possible\_cpu(i)  循环的意思可以在  include/linux/cpumask.h 文件中  
看到，实际是查找这个   cpu\_possible\_map 的信息。  
参考《linux的CPU信息管理》。  
  
这个函数是取的位图      cpu\_possible\_map 的置位的位置，返回给变量  i。  
＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊  
第一个函数：    set\_load\_weight(&init\_task);  
      
参数：init\_task 是进程  0 的进程描述符。  
    它是在 arch/mips/kernel/init\_task.c 文件中定义的，通过  INIT\_TASK 宏静态初始化。  
  
这个函数根据进程的实时优先级或者静态优先级设置进程的  load\_weight 成员，  
这个成员是为了各个  CPU 的运行队列均衡。  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
static void set\_load\_weight(struct task\_struct \*p)  
{  
      
    这个  has\_rt\_policy（） 判断进程是否是实时进程，  
    即它的调度策略不是  SCHED\_NORMAL 和  SCHED\_BATCH 策略。  
    if (has\_rt\_policy(p)) {  
#ifdef CONFIG\_SMP  
        这个   task\_rq(） 宏定义是冲进程  p 取得进程  p 所属的运行队列指针。  
        if (p == task\_rq(p)->migration\_thread)  
        运行队列  struct runquere 的成员  migration\_thread 表示迁移线程的进程描述符指针。  
        这个进程赋值各个  CPU 的运载平衡。  
            p->load\_weight = 0;  
        else   
#endif  
            p->load\_weight = RTPRIO\_TO\_LOAD\_WEIGHT(p->rt\_priority);  
    } else  
        p->load\_weight = PRIO\_TO\_LOAD\_WEIGHT(p->static\_prio);  
}  
＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊  
  
第二个函数：open\_softirq(SCHED\_SOFTIRQ, run\_rebalance\_domains, NULL);  
  
这个函数open\_softirq()用于开启一个指定的软中断向量   nr，  
也即适当地初始化软中断向量nr所对应的软中断描述符softirq\_vec［nr］。  
  
在这里开启使用软中断向量  SCHED\_SOFTIRQ 。  
设置它的软中断处理函数指针为  run\_rebalance\_domains 函数。  
在这里这个软中断处理函数   run\_rebalance\_domains（） 函数的作用以后在讲解。  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
void open\_softirq(int nr, void (\*action)(struct softirq\_action\*), void \*data)  
{  
    softirq\_vec[nr].data = data;  
    softirq\_vec[nr].action = action;  
}  
＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊  
第三个函数： plist\_head\_init(&init\_task.pi\_waiters, &init\_task.pi\_lock);  
  
这个函数初始化  struct task\_struct 结构体的   struct plist\_head 结构成员。  
  
这个  struct task\_struct 结构体的  struct plist\_head  pi\_waiters  成员，  
是按优先级排序的，它其实是两个双链表的组合，目的是为了在遍历优先级时减少处理重复优  
先级所浪费的时间；最后，优先级是按从高至低排序的。  
  
这个成员表示：进程拥有的每个 mutex 的最高优先级的任务的队列，  
                  这个任务的  PI 优先级应等于  TOP。  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
static inline void  
plist\_head\_init(struct plist\_head \*head, spinlock\_t \*lock)  
{  
     初始化两个链表的头。  
    INIT\_LIST\_HEAD(&head->prio\_list);  
    INIT\_LIST\_HEAD(&head->node\_list);  
#ifdef CONFIG\_DEBUG\_PI\_LIST  
    head->lock = lock;  
#endif  
}     
＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊  
第四个函数：init\_idle(current, smp\_processor\_id());  
  
这个函数在这里初始化当前   CPU  的  idle 进程。  
这个函数没有设置这个  idle 进程的  NEED\_RESCHED 标志，以完成内核的启动。  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
void \_\_cpuinit init\_idle(struct task\_struct \*idle, int cpu)  
{  
    struct rq \*rq = cpu\_rq(cpu);  
    unsigned long flags;  
  
    idle->timestamp = sched\_clock();  
    idle->sleep\_avg = 0;  
    idle->array = NULL;   
    idle->prio = idle->normal\_prio = MAX\_PRIO;  
    idle->state = TASK\_RUNNING;  
    idle->cpus\_allowed = cpumask\_of\_cpu(cpu);  
    set\_task\_cpu(idle, cpu);  
  
    spin\_lock\_irqsave(&rq->lock, flags);  
    rq->curr = rq->idle = idle;  
#if defined(CONFIG\_SMP) && defined(\_\_ARCH\_WANT\_UNLOCKED\_CTXSW)  
    idle->oncpu = 1;  
#endif  
    spin\_unlock\_irqrestore(&rq->lock, flags);  
  
#if defined(CONFIG\_PREEMPT) && !defined(CONFIG\_PREEMPT\_BKL)  
    task\_thread\_info(idle)->preempt\_count = (idle->lock\_depth >= 0);  
#else  
    task\_thread\_info(idle)->preempt\_count = 0;  
#endif  
  
}  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
task\_struct 结构成员：  
    timestamp：  
    sleep\_avg：                 平均睡眠时间  
    array ：                    struct prio\_array 指针。  
    prio ：                      
    normal\_prio：              
    state ：                    进程的状态，运行，睡眠、或停止。  
    cpus\_allowed ：            设置进程可以使用的 CPU 。  
    oncpu :       
  
＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊  
问题：  
    1）锁的依赖关系如何设置？  
    2）task\_struct 结构成员的含义？  
＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊  
题外话：  
    文件   arch/mips/kernel/init\_task.c 中对进程 0 需要的一些数据结构进行了定义，并通过  
    一些宏静态初始化。  
      
static struct fs\_struct init\_fs = INIT\_FS;  
static struct files\_struct init\_files = INIT\_FILES;  
static struct signal\_struct init\_signals = INIT\_SIGNALS(init\_signals);  
static struct sighand\_struct init\_sighand = INIT\_SIGHAND(init\_sighand);  
struct mm\_struct init\_mm = INIT\_MM(init\_mm);  
  
初始化进程  0  的线程信息和堆栈。  
union thread\_union init\_thread\_union  
    \_\_attribute\_\_((\_\_section\_\_(".data.init\_task"),  
                   \_\_aligned\_\_(THREAD\_SIZE))) =  
        { INIT\_THREAD\_INFO(init\_task) };  
  
初始化进程  0   进程描述符。  
struct task\_struct init\_task = INIT\_TASK(init\_task);

**linux-mips启动分析（7）**

在 start\_kernel（） 函数中调用了 build\_all\_zonelists（） 函数。  
这个函数主要是构建系统所有节点的管理区链表。  
  
这个链表的作用：  这个链表是为了在一个分配不能够满足时可以考察下一个管理区来设置了。  
    在考察结束时，分配将从  ZONE\_HIGHMEM 回退到  ZONE\_NORMAL，  
    在分配时从  ZONE\_NORMAL 退回到  ZONE\_DMA 就不会回退了。  
==============================================================  
  
void \_\_meminit build\_all\_zonelists(void)  
{  
  
    在  start\_kernel 函数中调用   system\_state ＝ SYSTEM\_BOOTING。  
    if (system\_state == SYSTEM\_BOOTING) {  
        \_\_build\_all\_zonelists(NULL);  
  
        这个函数如果没有定义  CONFIG\_CPUSETS 宏，就是空函数。  
        这个特性是主要是用于NUMA架构和拥有大量逻辑CPU的SMP架构，开  
        启这一特性会浪费一些内存，对一般人的桌面环境没有任何帮助。   
        cpuset\_init\_current\_mems\_allowed();  
    } else {  
          
        这个函数我们暂时不关心。  
        stop\_machine\_run(\_\_build\_all\_zonelists, NULL, NR\_CPUS);  
         }  
  
    统计所有的管理区可以用于分配的内存空间的大小。  
    vm\_total\_pages = nr\_free\_pagecache\_pages();  
  
    printk( 打印节点数目和总共页面数目   );  
}  
＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊  
这个   \_\_build\_all\_zonelists（）  函数遍历每个内存节点，  
为每个内存节点的每个管理区建立  zonelist 链表（其实是数组）。  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
static int \_\_meminit \_\_build\_all\_zonelists(void \*dummy)  
{  
    int nid;  
  
    循环遍历每个内存节点。  
    for\_each\_online\_node(nid) {  
  
    为节点的每个管理区建立  zonelist 链表（其实是数组）。  
        build\_zonelists(NODE\_DATA(nid));  
  
    把节点的每个管理区的  zonelist 的  zlcache\_ptr 指针置为  NULL。  
        build\_zonelist\_cache(NODE\_DATA(nid));  
          }  
    return 0;  
}  
＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊  
遍历每个节点是通过  for\_each\_online\_node() 宏来实现的。  
这个宏定义在   include/linux/nodemask.h 文件。  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
#define for\_each\_online\_node(node) for\_each\_node\_mask((node), node\_online\_map)  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
#if MAX\_NUMNODES > 1  
#define for\_each\_node\_mask(node, mask)          \  
    for ((node) = first\_node(mask);         \  
        (node) < MAX\_NUMNODES;          \  
        (node) = next\_node((node), (mask)))  
#else /\* MAX\_NUMNODES == 1 \*/  
#define for\_each\_node\_mask(node, mask)          \  
    if (!nodes\_empty(mask))             \  
        for ((node) = 0; (node) < 1; (node)++)  
#endif /\* MAX\_NUMNODES \*/  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
可以使用的内存节点位图   node\_online\_map 来记录节点的存在。  
这个内存节点位图是在  mm/page\_alloc.c  文件中定义的。  
  
nodemask\_t node\_online\_map \_\_read\_mostly = { { [0] = 1UL } };  
EXPORT\_SYMBOL(node\_online\_map);  
nodemask\_t node\_possible\_map \_\_read\_mostly = NODE\_MASK\_ALL;  
EXPORT\_SYMBOL(node\_possible\_map);  
＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊  
这个  build\_zonelists（） 函数根据配置的不同也不同。  
  
我们没有配置   CONFIG\_NUMA 宏定义，为一致性内存模型。  
  
如果有大于  1  个的内存节点时，也需要把其他的内存节点的管理区也加入这个   zonelist 链表。  
不过为了降低某个特定的节点的压力，所以加入这个   zonelist 链表的管理区必须属于这个节点  N 的  
下一个节点 （N ＋ 1）的，如此形成一个环形。  
  
例如：要为节点  2  的  ZONE\_NORMAL 管理区建立它的  zonelist 链表。  
首先加入这个  zonelist 链表的管理区是节点 2  的管理区，  
其次加入这个  zonelist 链表的管理区是节点 3  的管理区，一直到  MAX\_NUMNODES －1 的节点。  
最后又从  0  节点开始加入这个  zonelist 链表，知道小于  2  的 节点结束。  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
  
static void \_\_meminit build\_zonelists(pg\_data\_t \*pgdat)  
{  
    int node, local\_node;  
    enum zone\_type i,j;  
  
    取得内存节点号。  
    local\_node = pgdat->node\_id;  
  
    for (i = 0; i < MAX\_NR\_ZONES; i++) {  
        struct zonelist \*zonelist;  
  
        每一个节点都有一个  zonelist 数组。  
        这个节点的每个管理区分配对应了一个  zonelist 。  
        zonelist = pgdat->node\_zonelists + i;  
  
        构建这个节点的   zone[i] 的管理区链表。  
        j = build\_zonelists\_node(pgdat, zonelist, 0, i);  
  
          
        下面这两个循环的意思是如果有其他的内存节点，就把其他的内存节点的管理区也加入这个链表。  
        不过为了降低某个特定的节点的压力，所以加入这个链表的管理区必须属于这个节点  N 的  
        下一个节点 （N ＋ 1）的，如此形成一个环形。  
        for (node = local\_node + 1; node < MAX\_NUMNODES; node++) {  
            if (!node\_online(node))  
                continue;  
            j = build\_zonelists\_node(NODE\_DATA(node), zonelist, j, i);  
                     }  
        for (node = 0; node < local\_node; node++) {  
            if (!node\_online(node))  
                continue;  
            j = build\_zonelists\_node(NODE\_DATA(node), zonelist, j, i);  
                     }  
  
        这个管理区链表的最后一个为  NULL 指针。  
        zonelist->zones[j] = NULL;  
          }  
}  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
struct zonelist {  
    struct zonelist\_cache \*zlcache\_ptr;          // NULL or &zlcache  
    struct zone \*zones[MAX\_ZONES\_PER\_ZONELIST + 1];      // NULL delimited  
#ifdef CONFIG\_NUMA  
    struct zonelist\_cache zlcache;               // optional ...  
#endif  
};        
  
所有的内存分配操作都是基于  zonelist 的，一个  zonelist 就是一系列的  zone 的列表，  
链表的第一个  zone 是分配的首要目标  zone ，其余的  zone 都是 ballback 使用的，  
他们的优先级依次降低。  
  
如果 zlcache\_ptr 指针不是 NULL，它所指向的就是 zlcache 的地址。  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－

这个函数被  build\_zonelists（） 函数调用。  
  
参数  zone\_type 的类型定义如下所示。  
enum zone\_type {  
#ifdef CONFIG\_ZONE\_DMA  
    ZONE\_DMA,  
#endif  
#ifdef CONFIG\_ZONE\_DMA32  
    ZONE\_DMA32,  
#endif  
    ZONE\_NORMAL,  
#ifdef CONFIG\_HIGHMEM  
    ZONE\_HIGHMEM,  
#endif  
    MAX\_NR\_ZONES  
};  
  
因此当我们传入的参数  zone\_type ＝ ZONE\_HIGHMEM 时，  
那么参数  zonelist 的成员  zone［］ 数组所形成的链表就是  
这个节点各个低一级的管理区的指针，  
  
zonelist->zones[ZONE\_HIGHMEM ］＝ pgdat->node\_zones ＋ ZONE\_HIGHMEM；  
zonelist->zones[ZONE\_NORMA   ］＝ pgdat->node\_zones ＋  ZONE\_NORMA；  
zonelist->zones[ZONE\_DMA32   ］＝ pgdat->node\_zones ＋  ZONE\_DMA32；  
zonelist->zones[ZONE\_DMA     ］＝ pgdat->node\_zones ＋  ZONE\_DMA；  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
static int \_\_meminit build\_zonelists\_node(pg\_data\_t \*pgdat,  
            struct zonelist \*zonelist, int nr\_zones, enum zone\_type zone\_type)  
{         
    struct zone \*zone;  
          
    检测参数。  
    BUG\_ON(zone\_type >= MAX\_NR\_ZONES);  
    zone\_type++;  
  
    do {  
        zone\_type--;  
        取得管理区的指针。  
        zone = pgdat->node\_zones + zone\_type;  
  
        这个  populated\_zone(） 函数是检测  zone 指针所指向的管理区所实际存在的页面数量是否不为  0 。  
        if (populated\_zone(zone)) {  
            zonelist->zones[nr\_zones++] = zone;  
            check\_highest\_zone(zone\_type);  
                     }  
  
    } while (zone\_type);  
  
    返回  zonelist 的  成员  zone[] 数组共有几个成员。  
    return nr\_zones;  
}  
＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊  
这个   build\_zonelist\_cache(） 函数把特定节点的  
每个管理区的  zonelist 的  zlcache\_ptr 指针置为  NULL。  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
static void \_\_meminit build\_zonelist\_cache(pg\_data\_t \*pgdat)  
{  
    int i;  
  
    for (i = 0; i < MAX\_NR\_ZONES; i++)  
        pgdat->node\_zonelists[i].zlcache\_ptr = NULL;  
}  
＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊  
这个   nr\_free\_pagecache\_pages（） 函数主要统计所有的管理区可以用于分配的内存空间的大小。  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
unsigned int nr\_free\_pagecache\_pages(void)  
{     
    return nr\_free\_zone\_pages(gfp\_zone(GFP\_HIGHUSER));  
}  
  
这个函数的参数   gfp\_zone(GFP\_HIGHUSER) 等价于   ZONE\_HIGHMEM 。  
＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊  
这个  present\_pages 成员表示这个管理区实际的页面数量。  
这个  pages\_high 是和  pages\_low、pages\_min 配合使用的，  
这三个极值用于跟踪一个管理区的分配内存的压力，  
当这个管理区的空闲页面的数量小于 pages\_low 值时，就会唤醒 kswapd 内核线程，来释放页面。  
当 kswapd 内核线程释放的内存页面大于   pages\_high 时，这个  kswapd 内核线程就会停止运行。  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
  
static unsigned int nr\_free\_zone\_pages(int offset)  
{  
    /\* Just pick one node, since fallback list is circular \*/  
    pg\_data\_t \*pgdat = NODE\_DATA(numa\_node\_id());  
    unsigned int sum = 0;  
  
    struct zonelist \*zonelist = pgdat->node\_zonelists + offset;  
    struct zone \*\*zonep = zonelist->zones;  
    struct zone \*zone;  
  
    for (zone = \*zonep++; zone; zone = \*zonep++) {  
        unsigned long size = zone->present\_pages;  
        unsigned long high = zone->pages\_high;  
        if (size > high)  
            sum += size - high;  
          }  
  
    return sum;  
}  
＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊  
问题：  
    1）遍历每个内存节点函数是通过位图  node\_online\_map 实现的，但是这个位图在什么地方初始化的？  
    2） 这个 stop\_machine\_run() 函数的意思？

**linux-mips启动分析（8）**

在  start\_kernel 函数中调用  sort\_main\_extable() 函数。  
这个函数对内核建立的异常表（ exception table ）进行重新排序。  
这个函数对  exception\_table\_entry［］ 数组根据异常的向量号进行堆排序。  
=============================================================  
  
void \_\_init sort\_main\_extable(void)  
{     
    sort\_extable(\_\_start\_\_\_ex\_table, \_\_stop\_\_\_ex\_table);  
}     
  
这个函数直接调用   sort\_extable(） 函数，这个  sort\_extable(） 函数的两个参数，  
这个两个变量    \_\_start\_\_\_ex\_table 和   \_\_stop\_\_\_ex\_table 在  
链接脚本文件    arch/mips/kernel/vmlinux.lds 中定义。  
＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊  
由于  MIPS 体系结构没有定义自己的  sort\_extable(） 函数，即没有定义  
这个   ARCH\_HAS\_SORT\_EXTABLE 宏，所以  MIPS 体系结构使用  linux 共有  
的   sort\_extable(） 函数和   cmp\_ex (） 函数。  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
这个    exception\_table\_entry 结构体如下。  
struct exception\_table\_entry  
{  
    unsigned long insn;  
    unsigned long nextinsn;  
} ;   
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
void sort\_extable(struct exception\_table\_entry \*start,  
          struct exception\_table\_entry \*finish)  
{  
    sort(start, finish - start, sizeof(struct exception\_table\_entry),  
         cmp\_ex, NULL);  
}     
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
static int cmp\_ex(const void \*a, const void \*b)  
{  
    const struct exception\_table\_entry \*x = a, \*y = b;  
  
    if (x->insn > y->insn)  
        return 1;  
    if (x->insn < y->insn)  
        return -1;  
    return 0;  
}  
＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊  
这个   sort(）函数定义在  lib/sort.c 文件中。  
这个 sort 函数对数组进行堆排序。  
参数：  
    base ：  是数组的基地址  
    num：       是数组所占有的空间大小，以字节为单位  
    size：    是数组成员的大小，以字节为单位  
    cmd：      是一个函数指针，对两个成员进行比较  
    swap：   是成员交换函数。  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
  
void sort(void \*base, size\_t num, size\_t size,  
      int (\*cmp)(const void \*, const void \*),  
      void (\*swap)(void \*, void \*, int size))  
{  
  
    int i = (num/2 - 1) \* size, n = num \* size, c, r;  
  
    if (!swap)  
        swap = (size == 4 ? u32\_swap : generic\_swap);  
  
    构建堆。  
    for ( ; i >= 0; i -= size) {  
        for (r = i; r \* 2 + size < n; r  = c) {  
            c = r \* 2 + size;  
            if (c < n - size && cmp(base + c, base + c + size) < 0)  
                c += size;  
            if (cmp(base + r, base + c) >= 0)  
                break;  
            swap(base + r, base + c, size);  
                     }  
           }  
  
  
    /\* sort \*/  
    for (i = n - size; i >= 0; i -= size) {  
        swap(base, base + i, size);  
        for (r = 0; r \* 2 + size < i; r = c) {  
            c = r \* 2 + size;  
            if (c < i - size && cmp(base + c, base + c + size) < 0)  
                c += size;  
            if (cmp(base + r, base + c) >= 0)  
                break;  
            swap(base + r, base + c, size);  
                       }  
          }  
}  
＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊  
参考：  
堆排序的设计思路是：  
定义堆为一个键值序列  {  k1 ,k2 ,…,kn  }，  
它具有如下特性：ki < = k2i，ki < = k2i + 1 (i = 1 ,2 ,…, [ n/ 2 ]) 。  
（这个思想是利用二叉树的特性，节点   i 的子节点为    2 ＊ i 和   2  ＊  i ＋  1 ）。  
对一组待排序的键值，首先把它们按堆的定义排列成一个序列 ( 称为初建堆)，这就找到了最小键值。  
然后将最小的键值取出，用剩下的键值再重新建堆，便得到次小键值。  
如此反复进行，直到把全部键值排好序为止。  
＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊  
问题：  
    1）这段代码没有仔细阅读，为什么要重新排序呢，而不是本身已经排好序列了？

**linux-mips启动分析（9）**

在   start\_kernle（） 函数中调用了  trap\_init（） 函数。  
这个函数  
==========================================================  
void \_\_init trap\_init(void)  
{  
    extern char except\_vec3\_generic, except\_vec3\_r4000;  
    extern char except\_vec4;  
    unsigned long i;  
      
    在我们的系统中没有定义这两个变量  ＝＝  0。  
    if (cpu\_has\_veic || cpu\_has\_vint)  
        ebase = (unsigned long) alloc\_bootmem\_low\_pages (0x200 + VECTORSPACING\*64);  
    else  
        ebase = CAC\_BASE;  
    这个   CAC\_BASE 表明支持  CACHE 的  base 地址，由于是  32 的 CPU ，所以等于  0x80000000。  
    这个   ebase 变量表示异常入口点的基地址。  
  
    一般情况下不配置  CONFIG\_CPU\_MIPSR2\_SRSS 这个宏，所以这个函数为空函数。  
    mips\_srs\_init();  
  
    初始化  CUP 的  TLB 和  cache ，以及它们的异常处理程序。  
    per\_cpu\_trap\_init();  
  
    设置通用异常入口（ebase＋0x180）的处理程序为  except\_vec3\_generic 。  
    set\_handler(0x180, &except\_vec3\_generic, 0x80);  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
    在通用异常处理程序中，读取  CAUSE 寄存器的  ExcCode 域的值，这个预在发生异常时，  
    由  CPU 自动设置，使软件也可以知道异常的原因。  
    通用异常处理程序使用  ExcCode 域的值，来索引异常处理表   exception\_handlers［］。  
    这个通用异常处理表其实是一个  unsigned long exception\_handlers[32] 数组，表示  
    相应异常的处理程序地址，下面的代码进行初始化为保留的异常处理。  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
    for (i = 0; i <= 31; i++)  
        set\_except\_vector(i, handle\_reserved);  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
    如果  CPU 有  EJTAG，设置  EJTAG 的异常处理程序。  
    这个可以在  probe\_cpu() 函数中自动检测，并设置  cpu\_data 的  option 成员。  
    也可以在  include/asm-mips/mach-XXXX/cpu-feature-overrides.h  
    头文件中设置  。  
    移植相关，移植时需要配置。  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
    if (cpu\_has\_ejtag && board\_ejtag\_handler\_setup)  
        board\_ejtag\_handler\_setup ();  
  
    设置  CPU 是否有  watch （内存的访问检测点）异常，如果有进行设置。  
    移植相关，移植时需要配置。  
    if (cpu\_has\_watch)  
        set\_except\_vector(23, handle\_watch);  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
    初始化中断处理器，如果有外部中断控制器或者支持中断向量模式，  
    或者除法异常时 (cpu\_has\_divec) ，需要进行特殊的操作。  
    下面的我们的  CPU 不执行，不支持。  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
    if (cpu\_has\_veic || cpu\_has\_vint) {  
        int nvec = cpu\_has\_veic ? 64 : 8;  
        for (i = 0; i < nvec; i++)  
            set\_vi\_handler(i, NULL);  
    }else if (cpu\_has\_divec)  
        set\_handler(0x200, &except\_vec4, 0x8);  
  
    设置  CPU 的  cache 的奇偶检测位。  
    parity\_protection\_init();  
  
    由于  Data Bus Errors / Instruction Bus Errors 异常是由外部硬件通知的，  
    所以这两种异常需要有板级的特殊处理程序。  
    if (board\_be\_init)  
        board\_be\_init();  
  
    根据  ExcCode 域的值对通用异常处理表进行初始化，注册各个通用异常处理程序。  
    可以查看  CACUS 寄存器    ExcCode 域的值所对应的异常种类。  
    set\_except\_vector(0, handle\_int);      中断的处理程序  
    set\_except\_vector(1, handle\_tlbm);       
    set\_except\_vector(2, handle\_tlbl);  
    set\_except\_vector(3, handle\_tlbs);  
    set\_except\_vector(4, handle\_adel);  
    set\_except\_vector(5, handle\_ades);  
    set\_except\_vector(6, handle\_ibe);  
    set\_except\_vector(7, handle\_dbe);  
    set\_except\_vector(8, handle\_sys);  
    set\_except\_vector(9, handle\_bp);  
    set\_except\_vector(10, rdhwr\_noopt ? handle\_ri :  
              (cpu\_has\_vtag\_icache ?  
               handle\_ri\_rdhwr\_vivt : handle\_ri\_rdhwr));  
    set\_except\_vector(11, handle\_cpu);  
    set\_except\_vector(12, handle\_ov);  
    set\_except\_vector(13, handle\_tr);  
  
    if (current\_cpu\_data.cputype == CPU\_R6000 ||  
        current\_cpu\_data.cputype == CPU\_R6000A) {  
        如果  CPU 的类型是   CPU\_R6000 或者   CPU\_R6000A，就  
        注册一些专用的通用异常处理程序。  
      }  
      
    如果  CPU 支持不可屏蔽中断，设置不可屏蔽中断的处理函数。  
    移植相关，移植时需要配置。  
    if (board\_nmi\_handler\_setup)  
        board\_nmi\_handler\_setup();  
  
    if (cpu\_has\_fpu && !cpu\_has\_nofpuex)  
        set\_except\_vector(15, handle\_fpe);  
  
    set\_except\_vector(22, handle\_mdmx);  
  
    if (cpu\_has\_mcheck)  
        set\_except\_vector(24, handle\_mcheck);  
  
    if (cpu\_has\_mipsmt)  
        set\_except\_vector(25, handle\_mt);  
  
    set\_except\_vector(26, handle\_dsp);  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
    选择特殊的通用异常处理程序，并拷贝到通用异常入口（ebase＋0x180）。  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
    if (cpu\_has\_vce)  
        memcpy((void \*)(CAC\_BASE + 0x180), &except\_vec3\_r4000, 0x100);  
    else if (cpu\_has\_4kex)  
        memcpy((void \*)(CAC\_BASE + 0x180), &except\_vec3\_generic, 0x80);  
    else  
        memcpy((void \*)(CAC\_BASE + 0x080), &except\_vec3\_generic, 0x80);  
  
    设置浮点处理器的保存和加载程序指针。  
    signal\_init();  
  
    这个  flush\_icache\_range(）是一个函数指针，在  r4k\_cache\_init（） 函数中赋值。  
    参考《linux-mips启动分析（9-1）》。  
    把  ebase 地址到   ebase + 0x400 地址的指令装入  指令  cache 中去。  
    flush\_icache\_range(ebase, ebase + 0x400);  
  
    把  TLB 修改、加载和读取异常的处理代码加载到指令缓存中。  
    flush\_tlb\_handlers();  
}  
＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊  
 这个  cpu\_has\_veic 变量是在  include/asm-mips/cpu-features.h 文件中定义的，  
  
这个    CONFIG\_CPU\_MIPSR2\_IRQ\_EI 宏定义，表明是是使用了一个外部中断控制器来快速的  
分发中断，如果不知道是否是使用了外部中断控制器，就不应该设置这个宏定义。  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
#if defined(CONFIG\_CPU\_MIPSR2\_IRQ\_EI) && !defined(cpu\_has\_veic)  
# define cpu\_has\_veic       (cpu\_data[0].options & MIPS\_CPU\_VEIC)  
#elif !defined(cpu\_has\_veic)  
# define cpu\_has\_veic           0  
#endif  
＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊  
这个  cpu\_has\_vint 变量是在  include/asm-mips/cpu-features.h 文件中定义的，  
  
这个    CONFIG\_CPU\_MIPSR2\_IRQ\_VI 宏定义，表明  CPU 是否支持  Vectored interrupt mode。  
这种模式能够使中断分发反应更加快速。这种模式的代码兼容非  Vectored interrupt mode。  
所以如果一个  CPU 不支持这种模式，也可以选择上这个选项。  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
#if defined(CONFIG\_CPU\_MIPSR2\_IRQ\_VI) && !defined(cpu\_has\_vint)  
# define cpu\_has\_vint       (cpu\_data[0].options & MIPS\_CPU\_VINT)  
#elif !defined(cpu\_has\_vint)  
# define cpu\_has\_vint           0  
#endif  
＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊  
这个  CONFIG\_CPU\_MIPSR2\_SRS 宏配置的帮助信息：  
  
       Allow the kernel to use shadow register sets for fast interrupts.  
       Interrupt handlers must be specially written to use shadow sets.  
       Say N unless you know that shadow register set upport is needed.  
＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊  
把  addr 起始地址的代码（异常处理程序）拷贝到  (ebase + offset) 地址（异常处理入口点）。  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
void \_\_init set\_handler (unsigned long offset, void \*addr, unsigned long size)  
{  
    memcpy((void \*)(ebase + offset), addr, size);  
    flush\_icache\_range(ebase + offset, ebase + offset + size);  
}  
＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊  
这个 signal\_init（） 函数被  trap\_init() 函数所调用。  
这个函数初始化  save\_fp\_context 和   restore\_fp\_context 这两个函数指针。  
  
这两个变量   save\_fp\_context 和   restore\_fp\_context 是两个函数指针。  
这个两个函数用来保存和重新加载协处理器浮点处理器的上下文。  
  
asmlinkage int (\*save\_fp\_context)(struct sigcontext \_\_user \*sc);  
asmlinkage int (\*restore\_fp\_context)(struct sigcontext \_\_user \*sc);  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
static inline void signal\_init(void)  
{  
  
    if (cpu\_has\_fpu) {  
        save\_fp\_context = \_save\_fp\_context;  
        restore\_fp\_context = \_restore\_fp\_context;  
    } else {  
        save\_fp\_context = fpu\_emulator\_save\_context;  
        restore\_fp\_context = fpu\_emulator\_restore\_context;  
           }  
}  
＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊  
    这个  flush\_icache\_range(）是一个函数指针，在  r4k\_cache\_init（） 函数中赋值。  
    参考《linux-mips启动分析（9-1）》。  
    在  r4k\_cache\_init（）赋值   flush\_icache\_range  = r4k\_flush\_icache\_range 。  
    这个函数的主要作用是刷新  指定指令  cache 的范围。  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
static void r4k\_flush\_icache\_range(unsigned long start, unsigned long end)  
{  
    struct flush\_icache\_range\_args args;  
      
    args.start = start;  
    args.end = end;  
  
    r4k\_on\_each\_cpu(local\_r4k\_flush\_icache\_range, &args, 1, 1);  
    instruction\_hazard();  
}     
＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊  
这个   flush\_tlb\_handlers(） 函数被  trap\_init() 函数所调用。  
这个函数把  TLB 修改、加载和读取异常的处理代码加载到指令缓存中。  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
void \_\_init flush\_tlb\_handlers(void)  
{  
    把  TLB 读取异常的处理代码加载到指令缓存中。  
    flush\_icache\_range((unsigned long)handle\_tlbl,  
               (unsigned long)handle\_tlbl + sizeof(handle\_tlbl));  
    把  TLB 加载异常的处理代码加载到指令缓存中。  
    flush\_icache\_range((unsigned long)handle\_tlbs,  
               (unsigned long)handle\_tlbs + sizeof(handle\_tlbs));  
    把  TLB 修改异常的处理代码加载到指令缓存中。  
    flush\_icache\_range((unsigned long)handle\_tlbm,  
               (unsigned long)handle\_tlbm + sizeof(handle\_tlbm));  
}  
＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊  
问题：  
    1）通用异常处理程序    except\_vec3\_generic（） 没有看？  
    2）这个    flush\_icache\_range（） 函数的实现？

**linux-mips启动分析（9-1）**

这个   per\_cpu\_trap\_init（） 函数被  trap\_init（） 函数调用。  
这个函数根据  cpu\_probe() 函数检测的  CPU 特性或者移植时的配置，对寄存器进行设置。  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
在这个函数中多次使用到了    CONFIG\_MIPS\_MT\_SMTC  这个宏定义。  
  
宏定义   CONFIG\_MIPS\_MT\_SMTC 是使用多核的  SMTC Linux 时定义的。  
一般情况下不考虑。MIPS已经开发出  SMP Linux的改进版，  
叫做  SMTC (线程上下文对称多处理) Linux。  
SMTC Linux能理解轻量级  TC 的概念，并能因此减少某些与SMP Linux相关的开销。  
  
所以把这个     CONFIG\_MIPS\_MT\_SMTC 宏关闭的代码去掉了。  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
由于  CPU 是  32 位的，没有定义   CONFIG\_64BIT 这个宏，把这个宏关闭的代码也去掉了。  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
在移植时需要选择  MIPS 体系结构的版本， Release 1 or Release 2。（移植相关）  
config CPU\_MIPSR1  
    bool  
    default y if CPU\_MIPS32\_R1 || CPU\_MIPS64\_R1  
  
config CPU\_MIPSR2  
    bool  
    default y if CPU\_MIPS32\_R2 || CPU\_MIPS64\_R2  
  
我们的  CPU 为 Release 1，所以把   CPU\_MIPSR2 宏关闭的代码去掉了。  
=============================================================  
void \_\_init per\_cpu\_trap\_init(void)  
{  
    取得当前  CPU 的逻辑号。  
    unsigned int cpu = smp\_processor\_id();  
    unsigned int status\_set = ST0\_CU0;  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
    这个  current\_cpu\_data 变量是由宏定义的：  
    如下所示：  
    #define current\_cpu\_data cpu\_data[smp\_processor\_id()]  
    这个  cpu\_data［］的数据是在  cpu\_probe（） 函数中填充的。  
    参考《linux-mips启动分析（3）》文件中的     cpu\_probe（） 函数讲解。  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
    检测当前  CPU 的  ISA 等级，如果等于  MIPS\_CPU\_ISA\_IV，则修改状态寄存器进行配置。  
    if (current\_cpu\_data.isa\_level == MIPS\_CPU\_ISA\_IV)  
        status\_set |= ST0\_XX;  
    change\_c0\_status(ST0\_CU|ST0\_MX|ST0\_RE|ST0\_FR|ST0\_BEV|ST0\_TS|ST0\_KX|ST0\_SX|ST0\_UX, status\_set);  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
这个  cpu\_has\_dsp 变量是在     cpu\_probe（） 函数中检测到的，  
如：  #define cpu\_has\_dsp     (cpu\_data[0].ases & MIPS\_ASE\_DSP)     
  
也可以在移植内核时，定义这个宏，为  0  或者  1  ，移植相关。  
    if (cpu\_has\_dsp)  
        set\_c0\_status(ST0\_MX);  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
    从上面的代码分析，得知这两个变量  cpu\_has\_veic， cpu\_has\_vint 都等于  0。  
    if (cpu\_has\_veic || cpu\_has\_vint) {  
        write\_c0\_ebase (ebase);  
        change\_c0\_intctl (0x3e0, VECTORSPACING);  
           }  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
这个  cpu\_has\_divec 变量是在     cpu\_probe（） 函数中检测到的，  
如： #define cpu\_has\_divec       (cpu\_data[0].options & MIPS\_CPU\_DIVEC)  
  
也可以在移植内核时，定义这个宏，为  0  或者  1  ，移植相关。  
  
    if (cpu\_has\_divec) {  
        if (cpu\_has\_mipsmt) {  
            unsigned int vpflags = dvpe();  
            set\_c0\_cause(CAUSEF\_IV);  
            evpe(vpflags);  
        } else  
            set\_c0\_cause(CAUSEF\_IV);  
          }  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
这个  cpu\_has\_mips\_r2 宏变量的定义与否，参考上面的解释。  
在我们的  CPU 上没有定义  MIPSR2 这个宏，而是  MIPSR1 版本的。  
  
这两个变量在  arch/mips/kernel/time.c 文件中定义。  
  
    if (cpu\_has\_mips\_r2) {  
        cp0\_compare\_irq = (read\_c0\_intctl () >> 29) & 7;  
        cp0\_perfcount\_irq = (read\_c0\_intctl () >> 26) & 7;  
        if (cp0\_perfcount\_irq == cp0\_compare\_irq)  
            cp0\_perfcount\_irq = -1;  
    } else {  
        cp0\_compare\_irq = CP0\_LEGACY\_COMPARE\_IRQ;  
        cp0\_perfcount\_irq = -1;  
           }  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
    SAID 是地址空间标识的意思，这个值用来让操作系统识别当前进程的地址空间。  
  
    cpu\_data[cpu].asid\_cache = ASID\_FIRST\_VERSION;  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
    设置 TLB 异常需要提前设置的寄存器和变量。参考下面。  
  
    TLBMISS\_HANDLER\_SETUP();  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
    初始化当前进程 ，swapper 进程的，进程地址空间指针。  
    atomic\_inc(&init\_mm.mm\_count);  
    current->active\_mm = &init\_mm;  
    BUG\_ON(current->mm);  
    enter\_lazy\_tlb(&init\_mm, current);  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
    这个函数没有看完，在前面是检测  cache 的 cache linesize，cache 的路数和组数。  
    cpu\_cache\_init();  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
    初始化  TLB  的各个条目，并自动构建  TLB （重添、修改、读取、写入）的异常处理函数。  
    tlb\_init();  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
}  
＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊  
#define TLBMISS\_HANDLER\_SETUP\_PGD(pgd) \  
    pgd\_current[smp\_processor\_id()] = (unsigned long)(pgd)  
  
#define TLBMISS\_HANDLER\_SETUP()                     \  
    write\_c0\_context((unsigned long) smp\_processor\_id() << 25); \  
    TLBMISS\_HANDLER\_SETUP\_PGD(swapper\_pg\_dir)  
这个    TLBMISS\_HANDLER\_SETUP()  宏函数，设置   CP0 的 CONTEXT 寄存器。  
这个寄存器主要为为了在  TLB 异常发生后，加速异常的处理过程，把常驻内存的页表和  VPN 以某种格式  
打包一下，形成一个可以立即使用的指向内存页表的指针值。  
第一步设置  CP0 的 CONTEXT 寄存器为（  CPU 的逻辑号左移 25 位）。  
第二步调用   TLBMISS\_HANDLER\_SETUP\_PGD(） 宏函数，  
设置每 CPU 变量  pgd\_current［］数组， 当前 CPU 的页目录基地址。  
＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊  
这个      cpu\_cache\_init() 函数被   per\_cpu\_trap\_init（） 函数所调用。  
这个函数根据  cpu\_probe() 函数检测的  CPU 信息，或者配置的 CPU 信息，来选择 cache 的初始化函数。  
我们的 CPU 选择的是     r4k\_cache\_init() 函数。  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
void \_\_init cpu\_cache\_init(void)  
{  
    if (cpu\_has\_4k\_cache) {  
        extern void \_\_weak r4k\_cache\_init(void);  
  
        r4k\_cache\_init();  
        return;  
          }  
}  
＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊  
这个   r4k\_cache\_init() 函数定义在   arch/mips/mm/c-r4k.c 文件。  
  
void \_\_init r4k\_cache\_init(void)  
{  
    extern void build\_clear\_page(void);  
    extern void build\_copy\_page(void);  
    extern char except\_vec2\_generic;  
    struct cpuinfo\_mips \*c = &current\_cpu\_data;  
  
    设置  cache 错误的处理入口点  （ KSEG1ADDR（ebase＋0x100）），  
        处理程序为   except\_vec2\_generic。  
    set\_uncached\_handler (0x100, &except\_vec2\_generic, 0x80);  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
    检测一级和二级  cache 的    cache\_size, ways 和 line\_size。  
    由于我们的  CPU 没有二级  cache ，所以    setup\_scache() 函数只是返回了。  
    probe\_pcache();   
    setup\_scache();  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
    根据数据  cache linesize，设置  r4k\_blast\_dcache\_page 函数指针。  
    r4k\_blast\_dcache\_page\_setup();  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
    根据数据  cache linesize，设置 r4k\_blast\_dcache\_page\_indexed 函数指针。  
    r4k\_blast\_dcache\_page\_indexed\_setup();  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
    根据数据  cache linesize，设置 r4k\_blast\_dcache 函数指针。  
    r4k\_blast\_dcache\_setup();  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
    根据指令  cache linesize，设置  r4k\_blast\_icache\_page 函数指针。  
    r4k\_blast\_icache\_page\_setup();  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
    根据指令  cache linesize，设置  r4k\_blast\_icache\_page\_indexed 函数指针。  
    r4k\_blast\_icache\_page\_indexed\_setup();  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
    根据指令  cache linesize，设置  r4k\_blast\_icache 函数指针。  
    r4k\_blast\_icache\_setup();  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
    r4k\_blast\_scache\_page\_setup();  
    r4k\_blast\_scache\_page\_indexed\_setup();  
    r4k\_blast\_scache\_setup();  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
  
    设置数据 cache 的对齐掩码。  
    if (c->dcache.linesz)  
        shm\_align\_mask = max\_t( unsigned long,  
                    c->dcache.sets \* c->dcache.linesz - 1,  
                    PAGE\_SIZE - 1);  
    else  
        shm\_align\_mask = PAGE\_SIZE-1;  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
对  cache 的一些操作函数指针，进行赋值。  
  
    flush\_cache\_all     = r4k\_flush\_cache\_all;  
    \_\_flush\_cache\_all   = r4k\_\_\_flush\_cache\_all;  
    flush\_cache\_mm      = r4k\_flush\_cache\_mm;  
    flush\_cache\_page    = r4k\_flush\_cache\_page;  
    flush\_cache\_range   = r4k\_flush\_cache\_range;  
  
    flush\_cache\_sigtramp    = r4k\_flush\_cache\_sigtramp;  
    flush\_icache\_all    = r4k\_flush\_icache\_all;  
    flush\_data\_cache\_page   = r4k\_flush\_data\_cache\_page;  
    flush\_icache\_range  = r4k\_flush\_icache\_range;  
  
#ifdef CONFIG\_DMA\_NONCOHERENT  
    \_dma\_cache\_wback\_inv    = r4k\_dma\_cache\_wback\_inv;  
    \_dma\_cache\_wback    = r4k\_dma\_cache\_wback\_inv;  
    \_dma\_cache\_inv      = r4k\_dma\_cache\_inv;  
#endif  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
    这个些函数不知道有什么作用，看着好象是通过软件构建模拟指令？  
  
    build\_clear\_page();  
    build\_copy\_page();  
    local\_r4k\_\_\_flush\_cache\_all(NULL);  
    coherency\_setup();  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
}  
＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊  
static void \_\_init probe\_pcache(void)  
{  
    struct cpuinfo\_mips \*c = &current\_cpu\_data;  
    unsigned int config = read\_c0\_config();  
  
    取得  CPU 标识寄存器。  
    unsigned int prid = read\_c0\_prid();  
    unsigned long config1;  
    unsigned int lsize;  
  
    根据自己  CPU 的类型来选择处理的过程，cputype 字段在   cpu\_probe（） 函数中赋值。  
    switch (c->cputype) {  
  
    case CPU\_CLXRISC:  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
        初始  CP0 的  config1 寄存器的值。  
        config1 = read\_c0\_config1();  
        取得  config1 寄存器的  ［22：24］位的值，为指令cache的每一路有几组。  
        config1 = (config1 >> 22) & 0x07;  
        if (config1 == 0x07)  
            config1 = 10;  
        else  
            config1 = config1 + 11;  
        config1 += 2;  
        icache\_size = (1 << config1);  
        c->icache.linesz = 32;  
        c->icache.ways = 4;  
        c->icache.waybit = \_\_ffs(icache\_size / c->icache.ways);  
    对指令 cache 的  way 、  linesize 进行设置。  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
        config1 = read\_c0\_config1();  
        config1 = (config1 >> 13) & 0x07;  
        if (config1 == 0x07)  
            config1 = 10;  
        else  
            config1 = config1 + 11;  
        config1 += 2;  
        dcache\_size = (1 << config1);  
        c->dcache.linesz = 32;  
        c->dcache.ways = 4;  
        c->dcache.waybit = \_\_ffs(dcache\_size / c->dcache.ways);  
    对数据 cache 的  way 、  linesize 进行设置。  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
        c->dcache.flags = 0;  
        c->options |= MIPS\_CPU\_PREFETCH;  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
        break;  
    ｝  
  
    c->icache.waysize = icache\_size / c->icache.ways;  
    c->dcache.waysize = dcache\_size / c->dcache.ways;  
  
    c->icache.sets = c->icache.linesz ?  
        icache\_size / (c->icache.linesz \* c->icache.ways) : 0;  
    c->dcache.sets = c->dcache.linesz ?  
        dcache\_size / (c->dcache.linesz \* c->dcache.ways) : 0;  
｝  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
移植相关函数，如果这个函数不支持自己  CPU 的类型，需要添加，自己 CPU 类型的处理过程。  
  
如我们的处理过程为       case CPU\_CLXRISC，就是自己添加的。  
＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊  
这个   tlb\_init（） 函数被    per\_cpu\_trap\_init（） 函数所调用。  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
void \_\_init tlb\_init(void)  
{  
    取得  CP0 的  config0 寄存器的值。  
    unsigned int config = read\_c0\_config();  
  
    取得  TLB 的条目数。  
    probe\_tlb(config);  
  
    设置  CP0 的  PageMask 寄存器的 MASK 标志位，用来确定 1 条  TLB 可以映射的地址空间的大小。  
    write\_c0\_pagemask(PM\_DEFAULT\_MASK);  
  
    设置  CP0 的  Wired 寄存器，用来确定固定的 TLB 条目的数目＝＝0 。  
    write\_c0\_wired(0);  
    我们的  CPU 上没有这个寄存器。  
    write\_c0\_framemask(0);  
    temp\_tlb\_entry = current\_cpu\_data.tlbsize - 1;  
  
    初始化  TLB 的每个条目。为进程 0  的虚拟地址对应物理地址 0。  
    local\_flush\_tlb\_all();  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
    这个变量  ntlb 一个内核启动参数设置的，它的定义如下：  
    \_\_setup("ntlb=", set\_ntlb);   
    这个参数的处理函数为   static int \_\_init set\_ntlb(char \*str)；  
    这个  set\_ntlb(）主要是根据内核的启动命令行参数设置变量   ntlb 的值。  
    如果没有设置这个参数，则这个变量    ntlb 的值 默认 ＝＝ 0。  
    参考《Linux启动参数及实现》和 《启动时内核参数解析》。      
    下面我们看看设置了这个命令行启动参数的时候，发生什么？  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
    if (ntlb) {  
        检测设置的  TLB 参数的值是个有效值，大于 1，并且小于  TLB 总的条目数目。  
        if (ntlb > 1 && ntlb <= current\_cpu\_data.tlbsize) {  
            int wired = current\_cpu\_data.tlbsize - ntlb;  
            主要是设置  ntlb 个固定的 TLB 条目数目。  
            write\_c0\_wired(wired);  
            write\_c0\_index(wired-1);  
            printk ("Restricting TLB to %d entries\n", ntlb);  
        } else  
            printk("Ignoring invalid argument ntlb=%d\n", ntlb);  
         }  
  
    build\_tlb\_refill\_handler();  
}  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
static void \_\_init probe\_tlb(unsigned long config)  
{  
    struct cpuinfo\_mips \*c = &current\_cpu\_data;  
    unsigned int reg;  
  
    如果这个  CPU 不是 MIPS32 或者   MIPS64 的兼容  CPU，就不支持 config1 寄存器，  
    所以我们直接返回。  
    if ((c->processor\_id & 0xff0000) == PRID\_COMP\_LEGACY)  
        return;  
    读   CP0 的  config1 寄存器。  
    reg = read\_c0\_config1();  
    如果  CP0 的 config0 寄存器的 ［7：9］为 ＝＝ 0，则表示这个  CPU 没有 TLB。  
    if (!((config >> 7) & 3))  
        panic("No TLB present");  
  
    协处理器  CP0 的 config1 寄存器的 ［25：30］表示  TLB 的条目数。  
    c->tlbsize = ((reg >> 25) & 0x3f) + 1;  
}  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
void local\_flush\_tlb\_all(void)  
{  
    unsigned long flags;  
    unsigned long old\_ctx;  
    int entry;  
  
    进入临界区，关闭中断，保存中断标志。  
    ENTER\_CRITICAL(flags);  
  
    取得  EntryHi 寄存器的值。  
    在  TLB 异常中，   EntryHi 寄存器会装入引发 TLB 异常的程序地址的  VPN。  
    old\_ctx = read\_c0\_entryhi();  
  
    把  EntryLo0, EntryLo1 寄存器清  0。  
    write\_c0\_entrylo0(0);  
    write\_c0\_entrylo1(0);  
  
    读取   Wired 寄存器，取得固定的 TLB 条目的数目。  
    entry = read\_c0\_wired();  
  
    对每一个  TLB 条目，进行初始化。  
    必须保证  TLB 条目的  VFN 不同，如果相同，就会出现错误。  
    while (entry < current\_cpu\_data.tlbsize) {  
        设置   EntryHi 寄存器，即程序地址的  VPN。 ASID 进程号为 0。  
        write\_c0\_entryhi(UNIQUE\_ENTRYHI(entry));  
        设置  TLB 的  index 寄存器。  
        write\_c0\_index(entry);  
  
        如果是  MIPSR2 的  CPU 需要这个步骤，来保证  TLB 的填写正确，  
        为了  兼容  MIPSR2 的  CPU，所以  MIPSR1 的内核也需要做这个步骤。  
        mtc0\_tlbw\_hazard();  
        把  index 寄存器选择的  TLB 条目，写入   EntryHi、 EntryLo0, EntryLo1 寄存器的值。  
        tlb\_write\_indexed();  
        entry++;  
          }  
  
    如果是  MIPSR2 的  CPU 需要这个步骤，来保证  TLB 的填写正确，  
    为了  兼容  MIPSR2 的  CPU，所以  MIPSR1 的内核也需要做这个步骤。  
    tlbw\_use\_hazard();  
  
    把    EntryHi 寄存器原来的值写入     EntryHi 寄存器。  
    write\_c0\_entryhi(old\_ctx);  
      
    出临界区，打开中断，恢复中断标志位。  
    EXIT\_CRITICAL(flags);  
}  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
void \_\_init build\_tlb\_refill\_handler(void)  
{  
  
    设置标志变量，检测是否是第一次执行。  
    static int run\_once = 0;  
  
    根据  CPU 的类型，选择执行。  
    switch (current\_cpu\_data.cputype) {  
  
    default:  
        动态构建  TLB refill 异常处理程序，并拷贝到  ebase 变量指定的地址。  
        这个    ebase 变量的赋值，参考《linux-mips启动分析（9）》，在   trap\_init(）函数中赋值的。  
        当发生    TLB refill 异常时从  ebase 取指令（实际是  0x80000000 地址）。  
  
        build\_r4000\_tlb\_refill\_handler();  
        如果不是第一次执行则直接退出。  
        if (!run\_once) {  
            动态自己构建  TLB modify 异常、  TLB load 异常 和  TLB store 异常处理程序。  
            build\_r4000\_tlb\_load\_handler();  
            build\_r4000\_tlb\_store\_handler();  
            build\_r4000\_tlb\_modify\_handler();  
            run\_once++;  
                     }  
           }  
}  
＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊  
这两个函数      mtc0\_tlbw\_hazard() 和  tlbw\_use\_hazard() 的使用。  
  
        如果是  MIPSR2 的  CPU 需要这个步骤，来保证  TLB 的填写正确，  
        为了  兼容  MIPSR2 的  CPU，所以  MIPSR1 的内核也需要做这个步骤。  
＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊  
问题：  
    1）这两个  cp0\_compare\_irq 和   cp0\_perfcount\_irq 变量的意义？  
    2）struct cpuinfo\_mips 结构体中这个   asid\_cache 成员的意义？  
    3）这个      TLBMISS\_HANDLER\_SETUP() 宏函数，为什么这么设置  CONTEXT 寄存器？  
    4）这个  shm\_align\_mask 掩码的使用？  
    5）这个  cpu\_has\_4k\_cache 宏定义的实现和赋值？  
    6）这个    build\_clear\_page() 函数的作用？  
    7）这个  r4k\_cache\_init（） 函数最后有几个函数没有看完？  
    8）这四个函数  
          build\_r4000\_tlb\_load\_handler();  
          build\_r4000\_tlb\_store\_handler();  
          build\_r4000\_tlb\_modify\_handler();     
             build\_r4000\_tlb\_refill\_handler()   
        函数如何构建  TLB 异常处理指令的？  
    9）下面四个函数  
            build\_clear\_page();  
            build\_copy\_page();  
            local\_r4k\_\_\_flush\_cache\_all(NULL);  
            coherency\_setup();  
        的作用？  
    10） cache 错误的处理程序   except\_vec2\_generic（） 没有读？

**linux-mips启动分析（10）**

这个   rcu\_init（） 函数被  start\_kernel() 函数调用。  
  
这个  rcu\_init（） 函数初始化  RCU 机制，这个步骤必须比本地  timer 的初始化早。  
并在   cpu\_chain  通知链表上注册了一个通知结构体，由  CPU 管理子系统检测到   CPU\_UP\_PREPARE、  
 CPU\_UP\_PREPARE\_FROZEN、CPU\_DEAD、CPU\_DEAD\_FROZEN 事件时同时  RCU 管理子系统。  
  
参考《linux通知链表机制》和《Linux RCU 机制详解》。  
＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊  
在  rcu\_init（） 函数中使用到了  struct notifier\_block 类型的变量  rcu\_nb，  
  
这个变量定义如下：  
  
static struct notifier\_block \_\_cpuinitdata rcu\_nb = {  
    .notifier\_call  = rcu\_cpu\_notify,  
};  
  
这个通知结构体的回调函数为    rcu\_cpu\_notify（）。  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
void \_\_init rcu\_init(void)  
{  
    通知  RCU 管理子系统，当前  CPU 已经准备工作了。  
    rcu\_cpu\_notify(&rcu\_nb, CPU\_UP\_PREPARE, (void \*)(long)smp\_processor\_id());  
  
    register\_cpu\_notifier(&rcu\_nb);  
}  
＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊  
下面是  通知结构体的回调函数为    rcu\_cpu\_notify（）。  
当  CPU 管理子系统检测到   CPU\_UP\_PREPARE、 CPU\_UP\_PREPARE\_FROZEN、CPU\_DEAD、  
CPU\_DEAD\_FROZEN 事件时通知  RCU 管理子系统。  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
static int \_\_cpuinit rcu\_cpu\_notify(struct notifier\_block \*self,  
                unsigned long action, void \*hcpu)  
{             
    long cpu = (long)hcpu;  
    switch (action) {  
    case CPU\_UP\_PREPARE:  
    case CPU\_UP\_PREPARE\_FROZEN:  
        rcu\_online\_cpu(cpu);  
        break;  
    case CPU\_DEAD:  
    case CPU\_DEAD\_FROZEN:  
        rcu\_offline\_cpu(cpu);  
        break;  
    default:  
        break;  
           }  
    return NOTIFY\_OK;  
}  
＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊  
每一个 CPU 维护两个数据结构   rcu\_data,rcu\_bh\_data，它们用于保存回调函数，  
函数   call\_rcu()  和函数   call\_rcu\_bh()  用户注册回调函数，  
前者把回调函数注册到  rcu\_data，而后者则把回调函数注册到r  cu\_bh\_data，  
在每一个数据结构上，回调函数被组成一个链表，先注册的排在前头，后注册的排在末尾。  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
static void \_\_devinit rcu\_online\_cpu(int cpu)  
{  
    struct rcu\_data \*rdp = &per\_cpu(rcu\_data, cpu);  
    struct rcu\_data \*bh\_rdp = &per\_cpu(rcu\_bh\_data, cpu);  
  
    rcu\_init\_percpu\_data(cpu, &rcu\_ctrlblk, rdp);CT\_TARGET\_VENDOR  
    rcu\_init\_percpu\_data(cpu, &rcu\_bh\_ctrlblk, bh\_rdp);  
  
    初始化   RCU\_SOFTIRQ  类型的软中断，软中断处理函数为   rcu\_process\_callbacks（）。  
    tasklet\_init(&per\_cpu(rcu\_tasklet, cpu), rcu\_process\_callbacks, 0UL);  
}  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
参考《linux-mips启动分析（2-1）》文件    setup\_per\_cpu\_areas（） 函数的实现。  
参考《每CPU变量的数据组织和访问》。  
  
这个函数使用每  CPU 变量  rcu\_data 和  rcu\_bh\_data ，它们的定义如下所示：  
DEFINE\_PER\_CPU(struct rcu\_data, rcu\_data) = { 0L };  
DEFINE\_PER\_CPU(struct rcu\_data, rcu\_bh\_data) = { 0L };  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
调用这个函数的第二个参数是一个全局变量   rcu\_ctlblk 和  rcu\_bh\_ctrlblk，定义如下：  
  
static struct rcu\_ctrlblk rcu\_ctrlblk = {  
    .cur = -300,  
    .completed = -300,  
    .lock = \_\_SPIN\_LOCK\_UNLOCKED(&rcu\_ctrlblk.lock),  
    .cpumask = CPU\_MASK\_NONE,  
};  
static struct rcu\_ctrlblk rcu\_bh\_ctrlblk = {  
    .cur = -300,  
    .completed = -300,  
    .lock = \_\_SPIN\_LOCK\_UNLOCKED(&rcu\_bh\_ctrlblk.lock),  
    .cpumask = CPU\_MASK\_NONE,  
};  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
在  linux  内核中有一个总的  RCU 控制块，它的结构如下所示：  
  
struct rcu\_ctrlblk {      
    long    cur;        是指当前等待  grace period 的  RCU 的回调函数的批号。  
    long    completed;  是指已经完成的上次的  RCU 回调函数的批号。  
    int next\_pending;   指在  CPU 上是否还有   call\_rcu(）新添加的回调函数。  
      
    int signaled;   
      
    spinlock\_t  lock    \_\_\_\_cacheline\_internodealigned\_in\_smp;  
    cpumask\_t   cpumask;   
                              
} \_\_\_\_cacheline\_internodealigned\_in\_smp;  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
每个  CPU 有一个  per\_cpu 数据，它的类型如下所示：  
  
struct rcu\_data {  
  
    下面为 quiescent state 的处理成员  
    long    quiescbatch;       当前等待的  quiescent state 号  
    int     passed\_quiesc;     由  rcu\_qsctr\_inc(） 函数进行设置，  
                                          表示当前  CPU 已经经过了一次   quiescent state。  
    int     qs\_pending;        表示当前是否在等待    quiescent state 。  
      
    下面为回调函数的批号的处理成员  
    long            batch;        表示当前  CPU 等待的   grace period 的  RCU 的回调函数的批号。  
    struct rcu\_head \*nxtlist;     当调用  call\_rcu(）时在  nextlist 链表注册新的回调函数  
    struct rcu\_head \*\*nxttail;    指向  nextlist 链表最后一个节点  next 指针的地址  
  
    long            qlen;         
    struct rcu\_head \*curlist;      是指当前等待  grace period 的  RCU 的回调函数的链表。  
    struct rcu\_head \*\*curtail;  
  
    struct rcu\_head \*donelist;     是指已经等待到  grace period 的  RCU 的回调函数的链表。  
    struct rcu\_head \*\*donetail;  
  
    long        blimit;        
    int cpu;  
    struct rcu\_head barrier;  
};  
每次调用   call\_rcu(）  函数注册的回调函数，都会链入到   taillist 链表；  
当前等待   grace period  完成的函数都会链入到rdp->curlist上.  
到等待的  grace period  已经到来,就会将curlist上的链表移到donelist上.  
当一个grace period过了之后,就会将taillist上的数据移到rdp->curlist上.之后加册的回调函数又会将其加到rdp->taillist上.  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
每个  CPU 有一个   RCU\_SOFTIRQ  类型的软中断，软中断处理函数为   rcu\_process\_callbacks（）。  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
static void rcu\_init\_percpu\_data(int cpu, struct rcu\_ctrlblk \*rcp,  
                        struct rcu\_data \*rdp)  
{  
    memset(rdp, 0, sizeof(\*rdp));  
  
    初始化了三个链表，分别是    curlist ，  nextlist和   donelist。  
    rdp->curtail = &rdp->curlist;  
    rdp->nxttail = &rdp->nxtlist;  
    rdp->donetail = &rdp->donelist;  
  
    这个   quiescbatch 成员表示   rcu\_data  已经完成的   grace period  序号，  
    （在代码中也被称为了batch），  
    将   quiescbatch = completed，表示不需要等待  grace period.  
    rdp->quiescbatch = rcp->cCT\_TARGET\_VENDORompleted;  
    rdp->qs\_pending = 0;  
    rdp->cpu = cpu;  
    rdp->blimit = blimit;  
}  
＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊  
这个   rcu\_offline\_cpu（） 函数是一个可选函数，  
当  没有配置   CONFIG\_HOTPLUG\_CPU 这个选项时，这个rcu\_offline\_cpu（） 函数是空函数。  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
static void rcu\_offline\_cpu(int cpu)  
{     
    struct rcu\_data \*this\_rdp = &get\_cpu\_var(rcu\_data);  
    struct rcu\_data \*this\_bh\_rdp = &get\_cpu\_var(rcu\_bh\_data);  
      
    \_\_rcu\_offline\_cpu(this\_rdp, &rcu\_ctrlblk,  
                    &per\_cpu(rcu\_data, cpu));  
    \_\_rcu\_offline\_cpu(this\_bh\_rdp, &rcu\_bh\_ctrlblk,  
                    &per\_cpu(rcu\_bh\_data, cpu));  
  
    是否掉  per\_cpu 数据  rcu\_data 和  rcu\_bh\_data。  
    put\_cpu\_var(rcu\_data);  
    put\_cpu\_var(rcu\_bh\_data);  
  
    释放相应  CPU 的   RCU\_SOFTIRQ 软中断。  
    tasklet\_kill\_immediate(&per\_cpu(rcu\_taskleubuntu:/opt/my-crosstool/cross-ngt, cpu), cpu);  
}  
＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊  
在  CPU 管理的通知链表   cpu\_chain 注册通知结构  rcu\_nb。  
static struct notifier\_block \_\_cpuinitdata rcu\_nb = {  
    .notifier\_call  = rcu\_cpu\_notify,  
};  
  
这个通知结构的回调函数为    rcu\_cpu\_notify（） ，在上面也讲解过。  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
这个   Raw 类型的通知链表   cpu\_chain 是在  kernel/cpu.c 文件中定义的。  
static \_\_cpuinitdata RAW\_NOTIFIER\_HEAD(cpu\_chain);  
  
int \_\_cpuinit register\_cpu\_notifier(struct notifier\_block \*nb)  
{  
  
    int ret;  
    锁定   cpu\_add\_remove\_lock 互斥锁。  
    mutex\_lock(&cpu\_add\_remove\_lock);  
  
    在   Raw 类型的通知链表   cpu\_chain 上注册通知结构   nb。  
    ret = raw\_notifier\_chain\_register(&cpu\_chain, nb);  
    释放   cpu\_add\_remove\_lock 互斥锁。  
    mutex\_unlock(&cpu\_add\_remove\_lock);  
    return ret;  
}  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
int raw\_notifier\_chain\_register(struct raw\_notifier\_head \*nh,  
        struct notifier\_block \*n)  
{     
    return notifier\_chain\_register(&nh->head, n);   
}     
＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊  
最后，总结一下，RCU 机制在  linux 中的实现。  
一）     首先讲一下， RCU 机制使用的数据结构。  
1）   总控制单元  
在  linux 中有两种 RCU 锁，    一种是和软中断相关的，一种是和软中断不相关的，  
每种锁有一个总的控制单元，就有两种数据结构：  
  
static struct rcu\_ctrlblk rcu\_ctrlblk = {  
    .cur = -300,  
    .completed = -300,  
    .lock = \_\_SPIN\_LOCK\_UNLOCKED(&rcu\_ctrlblk.lock),  
    .cpumask = CPU\_MASK\_NONE,  
};  
static struct rcu\_ctrlblk rcu\_bh\_ctrlblk = {  
    .cur = -300,  
    .completed = -300,  
    .lock = \_\_SPIN\_LOCK\_UNLOCKED(&rcu\_bh\_ctrlblk.lock),  
    .cpumask = CPU\_MASK\_NONE,  
};  
  
2）     per\_cpu 数据  
这种  RCU 机制需要统计每个  CPU 是否经过了静止状态（quiescent state）。  
所以每个  CPU 也就需要一个数据结构，来统计自己是否经过了静止状态（quiescent state）。  
  
就有了  per\_cpu 数据（在  include/linux/rcupdate.h 文件定义）：  
  
DECLARE\_PER\_CPU(struct rcu\_data, rcu\_data);  
DECLARE\_PER\_CPU(struct rcu\_data, rcu\_bh\_data);  
  
二）   RCU 的操作流程  
1）  读者操作  
对于读者，RCU 仅需要使内核抢占失效，因此获得读锁和释放读锁分别定义为：  
#define rcu\_read\_lock()         preempt\_disable()  
#define rcu\_read\_unlock()       preempt\_enable()  
  
而和软中断相关的  RCU 锁，在进行临界区时，禁止了软中断饥和禁止抢占，使用下面的函数：  
#define rcu\_read\_lock\_bh()      local\_bh\_disable()  
#define rcu\_read\_unlock\_bh()    local\_bh\_enable()  
  
2）  写者的操作  
在写者访问数据时首先拷贝一个副本，然后对副本进行修改，最后调用   call\_rcu（） 函数  
把一个回调函数注册在当前   CPU 的  rcu\_data 或者   rcu\_bh\_data 的 nextlist 链表注册新的回调函数。  
这个   RCU 等待一个  grace period 周期，如果在当前  CPU 上执行这个  回调函数。  
  
3）  在进程切换时  
在每一次进程切换的时候,都会调用rcu\_qsctr\_inc().如下代码片段如示:  
asmlinkage void \_\_sched schedule(void)  
{  
    ......  
    rcu\_qsctr\_inc(cpu);  
    ......  
}  
Rcu\_qsctr\_inc()代码如下:  
static inline void rcu\_qsctr\_inc(int cpu)  
{  
    struct rcu\_data \*rdp = &per\_cpu(rcu\_data, cpu);  
    rdp->passed\_quiesc = 1;  
}  
该函数将对应   CPU  上的   rcu\_data  的  passed\_quiesc  成员设为了  1  。  
这个过程就标识该   CPU   经过了一次静止状态（ quiescent state）。  
  
4） 在  timer 中断时  
每个  CPU 都会在每次时钟中断时，周期性的检测在总控制单元  rcu\_ctrlblk 和 rcu\_bh\_ctrlblk 中是否有  
RCU 的回调函数需要处理。如果没有则跳过，如果有则执行下面的操作，  
  
本地  CPU 是否经过了一个静止状态（ quiescent state）。  
也检测本地   CPU 上的  RCU 回调函数经过了静止状态（ quiescent state），  
如果经过了则调用   RCU 回调函数的处理函数  rcu\_do\_batch（） 函数。  
  
三）     数据结构分析  
这种  RCU 机制，要求写者在访问它时首先拷贝一个副本，然后对副本进行修改，最后使用一个  
回调（callback）机制在适当的时机把指向原来数据的指针重新指向新的被修改的数据。  
这个时机就是所有引用该数据的   CPU  都退出对共享数据的操作。  
即要求写者必须要等它之前的读者全部都退出之后才能释放之前的资源。  
  
这里规定了是那个  CPU 修改的资源，必须有那个  CPU 释放资源的副本。  
  
所以每一个 CPU 都提供了一个链表   nxtlist ，供写者在这里注册回调函数，  
    struct rcu\_head \*nxtlist;     当调用  call\_rcu(）时在  nextlist 链表注册新的回调函数  
    struct rcu\_head \*\*nxttail;    指向  nextlist 链表最后一个节点  next 指针的地址  
  
每一个 CPU 都提供了一个链表   curlist ，表示当前正在等待 静止状态（quiescent state）的链表。  
    struct rcu\_head \*curlist;      是指当前等待  grace period 的  RCU 的回调函数的链表。  
    struct rcu\_head \*\*curtail;  
  
每一个 CPU 都提供了一个链表   doenlist ，表示已经等待到  grace period 的可以释放资源副本的链表。  
    struct rcu\_head \*donelist;     是指已经等待到  grace period 的  RCU 的回调函数的链表。  
    struct rcu\_head \*\*donetail;  
  
在  RCU 机制中，最困难的是判断这个  RCU 已经经过了一次   静止状态（quiescent state）。  
  
读者在调用  rcu\_read\_lock() 的时候要禁止抢占，所以如果读者仍然在这个临界区内，就不会发生上下文切换。  
因此，我们只需要判断如有的   CPU  都经过了一次上下文切换，就说明所有读者已经退出了。  
而  和软中断相关的  RCU 锁，读者调用   rcu\_read\_lock\_bh() 在进入读者临界区时，禁止了软中断饥和禁止抢占，  
基于同样的原因，如果所有   CPU 的都经过了一次上下文切换或者软中断，说明  所有读者已经退出了。  
＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊  
问题：  
    1）《Linux RCU 机制详解》没有看懂，需要再看一遍？

**linux-mips启动分析（11）**

在  start\_kernel（） 函数中调用   init\_IRQ（）  函数，用来初始化  
中断处理硬件相关的寄存器和 中断描述符数组     irq\_desc［］ 数组，  
每个中断号都有一个对应的中断描述符。  
  
移植相关，移植  linux 内核需要自己写       arch\_init\_irq() 函数的实现。  
=============================================================  
void \_\_init init\_IRQ(void)  
{  
    移植相关，移植  linux 内核需要自己写       arch\_init\_irq() 函数的实现。  
    arch\_init\_irq();  
  
    下面的内容先不看了，以后再看和   KBDG 相关的内容。  
#ifdef CONFIG\_KGDB  
    if (kgdb\_flag) {  
        printk("Wait for gdb client connection ...\n");  
        set\_debug\_traps();  
        breakpoint();  
    }  
#endif  
}  
＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊  
void \_\_init arch\_init\_irq(void)  
{  
    unsigned long cp0\_status;  
  
    设置  CP0 状态寄存器和中断相关的位。  
    清除  ERL 位，并屏蔽除了  HW0 外的所有中断，  HW0（IP2）连接到中断控制器的中断申请。  
    这个  CP0 状态寄存器  IM［8：15］的其他位，没有使用。  
    如果有另外一个中断控制器，可能使用这个  IM［8：15］的其他位。  
    cp0\_status = read\_c0\_status();  
    cp0\_status &= ~0x4;            
    cp0\_status |= 0x00000400;    
    set\_c0\_status(cp0\_status & 0xffff04ff);  
  
    设置   irq\_desc［］ 数组。  
    soc\_init\_irq();  
｝  
＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊  
/\*\*  
 \* struct irq\_chip - hardware interrupt chip descriptor  
   \*  
 \* @name:       name for /proc/interrupts  
 \* @startup:        start up the interrupt (defaults to ->enable if NULL)  
 \* @shutdown:       shut down the interrupt (defaults to ->disable if NULL)  
 \* @enable:     enable the interrupt (defaults to chip->unmask if NULL)  
 \* @disable:        disable the interrupt (defaults to chip->mask if NULL)  
 \* @ack:        start of a new interrupt  
 \* @mask:       mask an interrupt source  
 \* @mask\_ack:       ack and mask an interrupt source  
 \* @unmask:     unmask an interrupt source  
 \* @eoi:        end of interrupt - chip level  
 \* @end:        end of interrupt - flow level  
 \* @set\_affinity:   set the CPU affinity on SMP machines  
 \* @retrigger:      resend an IRQ to the CPU  
 \* @set\_type:       set the flow type (IRQ\_TYPE\_LEVEL/etc.) of an IRQ  
 \* @set\_wake:       enable/disable power-management wake-on of an IRQ  
   \*  
 \* @release:        release function solely used by UML  
 \* @typename:       obsoleted by name, kept as migration helper  
 \*/  
  
struct irq\_chip {  
    const char  \*name;  
    unsigned int    (\*startup)(unsigned int irq);  
    void        (\*shutdown)(unsigned int irq);  
    void        (\*enable)(unsigned int irq);  
    void        (\*disable)(unsigned int irq);  
  
    void        (\*ack)(unsigned int irq);  
    void        (\*mask)(unsigned int irq);  
    void        (\*mask\_ack)(unsigned int irq);  
    void        (\*unmask)(unsigned int irq);  
    void        (\*eoi)(unsigned int irq);  
  
    void        (\*end)(unsigned int irq);  
    void        (\*set\_affinity)(unsigned int irq, cpumask\_t dest);  
    int     (\*retrigger)(unsigned int irq);  
    int     (\*set\_type)(unsigned int irq, unsigned int flow\_type);  
    int     (\*set\_wake)(unsigned int irq, unsigned int on);  
  
    /\* Currently used only by UML, might disappear one day.\*/  
#ifdef CONFIG\_IRQ\_RELEASE\_METHOD  
    void        (\*release)(unsigned int irq, void \*dev\_id);  
#endif  
    /\*  
     \* For compatibility, ->typename is copied into ->name.  
     \* Will disappear.  
     \*/  
    const char  \*typename;  
};  
  
linux 使用  struct irq\_chip 结构体过定义一组接口方法，将不同的中断硬件行为统一进行了封装。  
一方面底层中断硬件驱动只要根据自身功能部分或全部实现这组方法，向上提供满足需要的中断处理功能；  
另一方面负责中断处理流程的   handle\_irq，通过使用   chip  的方法与硬件交互。  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
struct irq\_desc {  
    irq\_flow\_handler\_t     handle\_irq;         /\*高级中断事件控制\*/  
    struct irq\_chip         chip;                 /\*低级别硬件中断\*/  
    struct msi\_desc         msi\_desc;             /\*可屏蔽软中断描述符\*/  
    void             \*            handler\_data;  
    void             \*            chip\_data;  
    struct irqaction \*    action;                 /\* IRQ 服务程序链表 \*/  
    unsigned int             status;                 /\* IRQ 状态 \*/  
  
    unsigned int             depth;                 /\* 中断禁止次数,也称为禁止深度\*/  
    unsigned int             wake\_depth;         /\* nested wake enables \*/  
    unsigned int             irq\_count;             /\* 中断发生次数\*/  
    unsigned int             irqs\_unhandled;     /\*未处理中断的计数，这个的计数是在HZ/10内未处理的中断数\*/  
    unsigned long             last\_unhandled;     /\* 未处理中断计数的计时，他的值是上一次发生未处理中断时的jiffies值 \*/  
    spinlock\_t                 lock;                 /\*用于串行访问中断描述符数组的自旋锁\*/  
  
#ifdef CONFIG\_SMP  
    cpumask\_t                 affinity;  
    unsigned int             cpu;                     /\*SMP中cup的索引号，用于平衡调度＊/  
#endif  
  
#if defined(CONFIG\_GENERIC\_PENDING\_IRQ) || defined(CONFIG\_IRQBALANCE)  
    cpumask\_t                 pending\_mask;  
#endif  
  
#ifdef CONFIG\_PROC\_FS  
    struct proc\_dir\_entry \*dir;                 /\*  在/proc/irq/中对应的文件  \*/  
#endif  
    const char     \*            name;                 /\*要在interrupts中显示的名字\*/  
} \_\_\_\_cacheline\_internodealigned\_in\_smp;  
  
这是在SMP中使用的一个数据结构，当然在单cpu（更准确的说应该是单核cpu中）有一些数据项是没有使用的。  
这其中比较重要的数据项是   action  和   status。  
一个是真正要执行服务程序的一个链表，另一个是当前irq的状态。  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
这个函数初始化  irq\_desc［］ 数组，每个中断号都有一个对应的中断描述符，这些  
描述符组织在一起形成    irq\_desc［］ 数组。  
  
struct irq\_desc irq\_desc[NR\_IRQS] \_\_cacheline\_aligned\_in\_smp = {  
    [0 ... NR\_IRQS-1] = {  
        .status = IRQ\_DISABLED,  
        .chip = &no\_irq\_chip,  
        .handle\_irq = handle\_bad\_irq,  
        .depth = 1,  
        .lock = \_\_SPIN\_LOCK\_UNLOCKED(irq\_desc->lock),  
#ifdef CONFIG\_SMP  
        .affinity = CPU\_MASK\_ALL  
#endif  
  
            }  
}  ;  
  
这个   NR\_IRQS 宏是中断描述符的数量，默认是  128 个。  
#define NR\_IRQS 128  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
void \_\_init soc\_init\_irq(void)  
{  
    int i;  
  
    关闭从  0  到 31   的中断，并设置从  0  到 32 的中断的底层硬件为中断控制器。  
    for (i=0;i<32;i++) {  
        disable\_intc\_irq(i);  
        irq\_desc[i].chip= &intc\_irq\_type;  
           }  
  
    设置从  32  到   37  的中断描述符的  chip 指针为   DMA 类型的中断操作。  
    for (i=0;i<NUM\_DMA;i++)  
        irq\_desc[IRQ\_DMA\_0+i].chip = &dma\_irq\_type;  
  
    使能中断控制器中  DMA 中断控制位。  
    enable\_intc\_irq(IRQ\_DMAC);  
  
    设置从   40  到  43  的中断描述符的  chip 指针为  PCIC 类型的中断操作。  
    for (i=0;i<NUM\_PCID;i++)  
        irq\_desc[IRQ\_PCIC\_0+i].chip = &pcic\_irq\_type;  
  
    使能中断控制器中的  PCIC 的的中断控制位。  
    enable\_intc\_irq(IRQ\_PCIC);  
  
    设置从  48  到  117   的中断描述符的  chip 指针为  GPIO 类型的中断操作。  
    for (i=0;i<NUM\_GPIO;i++){  
        irq\_desc[IRQ\_GPIO\_0+i].status = IRQ\_DISABLED;  
        irq\_desc[IRQ\_GPIO\_0+i].action = NULL;  
        irq\_desc[IRQ\_GPIO\_0+i].depth = 1;  
        irq\_desc[IRQ\_GPIO\_0+i].chip = &gpio\_irq\_type;  
          }  
  
    使能中断控制器中的   GPIO 的中断控制位。  
    enable\_intc\_irq(IRQ\_GPIO0);  
    enable\_intc\_irq(IRQ\_GPIO1);  
    enable\_intc\_irq(IRQ\_GPIO2);  
}  
＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊  
在   soc\_init\_irq（） 函数中，最主要的操作是对中断描述符的  chip 指针赋值。  
其中共有四种类型的中断底层操作，intc\_irq\_type、dma\_irq\_type，  
pcic\_irq\_type  和   gpio\_irq\_type 。  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
static struct irq\_chip intc\_irq\_type = {  
    .name       = "INTC",  
    .startup    = startup\_intc\_irq,  
    .shutdown   = shutdown\_intc\_irq,  
    .enable     = enable\_intc\_irq,  
    .disable    = disable\_intc\_irq,  
    .ack        = mask\_and\_ack\_intc,  
    .mask       = disable\_intc\_irq,  
    .mask\_ack   = mask\_and\_ack\_intc,  
    .unmask     = enable\_intc\_irq,  
    .end        = end\_intc\_irq,  
};    
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
static struct irq\_chip dma\_irq\_type = {  
    .name   = "DMA",  
    .startup    = startup\_dma\_irq,  
    .shutdown   = shutdown\_dma\_irq,  
    .enable     = enable\_dma\_irq,  
    .disable    = disable\_dma\_irq,  
    .ack        = mask\_and\_ack\_dma\_irq,  
    .mask       = disable\_dma\_irq,  
    .mask\_ack   = mask\_and\_ack\_dma\_irq,  
    .unmask     = enable\_dma\_irq,  
    .end        = end\_dma\_irq,  
};  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
static struct irq\_chip pcic\_irq\_type = {  
    .name   = "PCIC",  
    .startup    = startup\_pcic\_irq,  
    .shutdown   = shutdown\_pcic\_irq,  
    .enable     = enable\_pcic\_irq,  
    .disable    = disable\_pcic\_irq,  
    .ack        = mask\_and\_ack\_pcic\_irq,  
    .mask       = disable\_pcic\_irq,  
    .mask\_ack   = mask\_and\_ack\_pcic\_irq,  
    .unmask     = enable\_pcic\_irq,  
    .end        = end\_pcic\_irq,  
};  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
static struct irq\_chip gpio\_irq\_type = {  
    .name   = "GPIO",  
    .startup    = startup\_gpio\_irq,  
    .shutdown   = shutdown\_gpio\_irq,  
    .enable     = enable\_gpio\_irq,  
    .disable    = disable\_gpio\_irq,  
    .ack        = mask\_and\_ack\_gpio\_irq,  
    .mask       = disable\_gpio\_irq,  
    .mask\_ack   = mask\_and\_ack\_gpio\_irq,  
    .unmask     = enable\_gpio\_irq,  
    .end        = end\_gpio\_irq,  
};  
＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊  
问题：  
    1） 中断描述符中的   chip 指针中的函数的调用？

**linux-mips启动分析（12）**

在  start\_kernel（） 函数中调用   pidhash\_init（）  函数。  
  
在  linux 内核中，有很多情况下需要从进程的  PID 导出对应的进程描述符指针。  
为了这个操作，系统在初始化阶段动态的分配了  4   个  hashtable，并把它们的地址  
存入  pid\_hash［］ 数组。  
  
下面是四种散列表和进程描述符中相关字段：  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
Hash table type   ｜      Field name         ｜         Description  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
PIDTYPE\_PID       ｜              pid             ｜         进程的  PID  
PIDTYPE\_TGID      ｜             tgid         ｜      线程组的领头进程的  PID  
PIDTYPE\_PGID      ｜        pgid         ｜      进程组的领头进程的  PID  
PIDTYPE\_SID       ｜       session      ｜        会话组的领头进程的  PID  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
这个  pidhash\_init() 函数为全局变量    pidhash\_shift 和  pid\_hash 进行赋值。  
这两个变量是在   kernel/pid.c  文件中定义的。  
  
static int pidhash\_shift;  
static struct hlist\_head \*pid\_hash;  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
散列表讲解：  
struct hlist\_head {  
    struct hlist\_node \*first;  
};  
  
struct hlist\_node {  
    struct hlist\_node \*next, \*\*pprev;  
};  
  
散列表的表头存放在   hlist\_head 数据结构中，这个结构仅仅是指向第一个节点的指针，  
如果链表为空，这个指针为  NULL。  
这个散列表的每一个节点的类型为   struct hlist\_node 数据结构，  
每个节点有两个指针，其中   next 指针指向下一个节点，  
另一个指针   pprev 指针指向前一个节点的  next 指针字段（它是个指针的指针，指向前一个节点  
的  next 字段，而前一个节点的  next 字段又指向这个节点）。  
因为不是循环链表，所以第一个节点的 pprev 指针和  最后一个节点的 next 指针为  NULL 指针。  
==============================================================  
void \_\_init pidhash\_init(void)  
{     
    int i, pidhash\_size;  
  
    这个  nr\_kernel\_pages 变量在  free\_area\_init\_core（） 函数中赋值，  
    参考《linux-mips启动分析（4-3）》，表示不包含高端内存的系统内存共有的内存页面数。  
    通过下面的公式计算出不包含高端内存的系统内存共有   多少  MB。  
    unsigned long megabytes = nr\_kernel\_pages >> (20 - PAGE\_SHIFT);  
      
    下面步骤计算出   pid\_hash 表的表项数目。这个  hashtable 的长度依赖于不包含高端内存  
    的系统内存的大小。  
    这个   pidhash\_shift 的范围是  从  4  到 12 。hashtable 的长度从  16 到 4096 项。  
    pidhash\_shift = max(4, fls(megabytes \* 4));  
    pidhash\_shift = min(12, pidhash\_shift);  
    pidhash\_size = 1 << pidhash\_shift;  
      
    为  PID hash 表分配空间。  
    pid\_hash = alloc\_bootmem(pidhash\_size \* sizeof(\*(pid\_hash)));  
    if (!pid\_hash)  
        panic("Could not alloc pidhash!\n");  
  
    初始化  PID 哈希表的头节点。  
    for (i = 0; i < pidhash\_size; i++)  
        INIT\_HLIST\_HEAD(&pid\_hash[i]);  
}  
＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊  
问题：  
    1）应该是四个  PID hash 表，为什么只分配了一个？

**linux-mips启动分析（13）**

在 star\_kernel() 函数中调用  init\_timers（） 函数。  
这个函数完成如下功能：  
    1）初始化本 CPU 上的定时器（timer）相关的数据结构  
    2）向   cpu\_chain 通知链注册元素 timers\_nb，该元素的回调函数用于初始化指定 CPU 上的定时器相关的数据结构。  
    3） 初始化时钟的软中断处理函数   
  
参考《Linux内部的时钟处理机制全面剖析》。  
参考《深入理解 linux 内核》。  
参考《linux通知链表机制》。  
================================================================  
void \_\_init init\_timers(void)  
{     
  
    初始当前  CPU 的定时器链表。  
    int err = timer\_cpu\_notify(&timers\_nb, (unsigned long)CPU\_UP\_PREPARE,  
                (void \*)(long)smp\_processor\_id());  
      
    如果没有定义  CONFIG\_TIMER\_STATS 这个宏，则这个   init\_timer\_stats() 函数为空函数。  
    init\_timer\_stats();  
      
    BUG\_ON(err == NOTIFY\_BAD);  
  
    在  CPU 管理的通知链表   cpu\_chain 注册通知结构  timers\_nb。  
    register\_cpu\_notifier(&timers\_nb);  
  
    注册时钟软中断  TIMER\_SOFTIRQ，它的处理函数为   run\_timer\_softirq（）。  
    这个处理函数如下所示。  
    open\_softirq(TIMER\_SOFTIRQ, run\_timer\_softirq, NULL);  
}     
＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊  
这个函数是 注册到  CPU 子系统的通知链表的回调函数。  
这个函数主要是在新的 CPU 开始工作或者停止工作（支持热插拔）时，通知  timer 管理器，  
使    timer 管理器管理器知道，并进行相应的处理。  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
static int \_\_cpuinit timer\_cpu\_notify(struct notifier\_block \*self,  
                unsigned long action, void \*hcpu)  
{  
    long cpu = (long)hcpu;  
    switch(action) {  
    case CPU\_UP\_PREPARE:  
    case CPU\_UP\_PREPARE\_FROZEN:  
        if (init\_timers\_cpu(cpu) < 0)  
            return NOTIFY\_BAD;  
        break;  
#ifdef CONFIG\_HOTPLUG\_CPU  
    case CPU\_DEAD:  
    case CPU\_DEAD\_FROZEN:  
        migrate\_timers(cpu);  
        break;  
#endif  
    default:  
        break;  
    }  
    return NOTIFY\_OK;  
}  
＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊  
注册到通知链表的回调函数支持两类事件（  UP 和   DEAD 类型的事件），  
下面为  支持   CPU\_UP\_PREPARE  和   CPU\_UP\_PREPARE\_FROZEN 类型事件的函数。  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
static int \_\_devinit init\_timers\_cpu(int cpu)  
{  
    int j;  
    tvec\_base\_t \*base;  
  
    定义静态局部变量    tvec\_base\_done［］ 数组，表示是否已经初始化了由索引号表示的  CPU。  
    static char \_\_devinitdata tvec\_base\_done[NR\_CPUS];  
  
    检测  CPU 的初始化是否已经做过，如果没有做过，继续，如果已经做过，则跳过。      
    if (!tvec\_base\_done[cpu]) {  
  
        定义静态全局变量，表示是否是第一个启动的  CPU 的初始化。  
        static char boot\_done;  
  
        如果不是启动  CPU，则走这个路径。  
        if (boot\_done) {  
  
            为每个  CPU 分配   struct tvec\_t\_base\_s 结构的空间。  
            base = kmalloc\_node(sizeof(\*base), GFP\_KERNEL,  
                        cpu\_to\_node(cpu));  
            if (!base)  
                return -ENOMEM;  
  
            if (tbase\_get\_deferrable(base)) {  
                WARN\_ON(1);  
                kfree(base);  
                return -ENOMEM;  
                               }  
            把这个结构体空间清  0。  
            memset(base, 0, sizeof(\*base));  
  
            设置  per\_cpu 变量中的指针指向分配的空间。  
            per\_cpu(tvec\_bases, cpu) = base;  
        } else {  
        这个路径是在  boot 阶段，启动的第一个  CPU 的路径。  
            设置标志已经启动过了。  
            由于在启动阶段   per\_cpu 数据没有准备好，所以使用静态定义的结构。  
            这是应为  per\_cpu 数据是一个指针，尽管  per\_cpu 中为指针预留了空间，  
            但是指针指向的空间，没有进行分配，都是执行  boot\_tvec\_bases 的。  
            boot\_done = 1;  
            base = &boot\_tvec\_bases;  
                     }  
  
        设置  CPU 已经设置过了。  
        tvec\_base\_done[cpu] = 1;  
    } else {  
        base = per\_cpu(tvec\_bases, cpu);  
         }  
  
  
    spin\_lock\_init(&base->lock);  
    lockdep\_set\_class(&base->lock, base\_lock\_keys + cpu);  
  
    初始化  tvec\_t\_base\_s 结构中每个链表的头节点。  
    for (j = 0; j < TVN\_SIZE; j++) {  
        INIT\_LIST\_HEAD(base->tv5.vec + j);  
        INIT\_LIST\_HEAD(base->tv4.vec + j);  
        INIT\_LIST\_HEAD(base->tv3.vec + j);  
        INIT\_LIST\_HEAD(base->tv2.vec + j);  
          }  
    for (j = 0; j < TVR\_SIZE; j++)  
        INIT\_LIST\_HEAD(base->tv1.vec + j);  
  
    base->timer\_jiffies = jiffies;  
    return 0;  
}  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
在这个   init\_timers\_cpu() 函数中，使用了全局变量   boot\_tvec\_bases。  
它的定义如下所示：  
  
typedef struct tvec\_s {  
    struct list\_head vec[TVN\_SIZE];  
} tvec\_t;  
  
typedef struct tvec\_root\_s {  
    struct list\_head vec[TVR\_SIZE];  
} tvec\_root\_t;  
  
  
struct tvec\_t\_base\_s {  
    spinlock\_t lock;  
    struct timer\_list \*running\_timer;  
    unsigned long timer\_jiffies;  
    tvec\_root\_t tv1;  
    tvec\_t tv2;  
    tvec\_t tv3;  
    tvec\_t tv4;  
    tvec\_t tv5;  
} \_\_\_\_cacheline\_aligned;  
  
typedef struct tvec\_t\_base\_s tvec\_base\_t;  
tvec\_base\_t boot\_tvec\_bases;  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
lock                 spinlock\_t                 用于同步操作  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
running\_timer     struct timer\_list \*     正在处理的定时器  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
timer\_jiffies     unsigned long             当前正在处理的定时器到期时间  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
tv1              struct tvec\_root         保存了到期时间从 timer\_jiffies 到（2^8 －1）  
                                                之间（包括边缘值）的所有定时器  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
tv2              struct tvec             保存了到期时间从 timer\_jiffies +（2^8）到  
                                                 timer\_jiffies + (2^14-1)之间（包括边缘值）的 所有定时器  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
tv3              struct tvec             保存了到期时间从 timer\_jiffies +(2^14)到  
                                                 timer\_jiffies +(2^20-1)之间（包括边缘值）的所有定时器  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
tv4              struct tvec             保存了到期时间从 timer\_jiffies +(2^20)到   
                                                timer\_jiffies + (2^26-1)之间（包括边缘值）的所有定时器  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
tv5              struct tvec             保存了到期时间从 timer\_jiffies +(2^16)到   
                                                timer\_jiffies +(2^32-1)之间（包括边缘值）的所有定时器  
＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊  
下面为  支持    CPU\_DEAD  和  CPU\_DEAD\_FROZEN 类型事件的函数。  
当没有配置这个宏   CONFIG\_HOTPLUG\_CPU  定义时，即系统不支持  CPU 热插拔时，  
这个  migrate\_timers（） 为空函数；  
当配置了这个宏定义时，这个函数如下所示。  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
static void \_\_devinit migrate\_timers(int cpu)  
{  
    tvec\_base\_t \*old\_base;  
    tvec\_base\_t \*new\_base;  
    int i;  
  
    首先检查这个 CPU 是否在工作，如果仍然在工作，则发出警告信息。  
    BUG\_ON(cpu\_online(cpu));  
  
    这个 tvec\_bases 是一个  per\_cpu 变量，是   tvec\_base\_t 的指针。  
    这个  per\_cpu() 函数取得这个 cpu 的私有变量，即tvec\_base\_t 的指针    
    old\_base = per\_cpu(tvec\_bases, cpu);  
  
    这个  get\_cpu\_var()函数取得当前 cpu 的tvec\_base\_t 的指针 。  
    new\_base = get\_cpu\_var(tvec\_bases);  
  
    禁止当前 cpu 的中断，同样也禁止了内核抢占。  
    local\_irq\_disable();  
  
    根据需求锁定这两个自旋锁 new\_base->lock, old\_base->lock 。  
    double\_spin\_lock(&new\_base->lock, &old\_base->lock, smp\_processor\_id() < cpu);  
  
    如果参数  cpu 上有正在发生的定时器，则发出 bug 信息。  
    BUG\_ON(old\_base->running\_timer);  
  
    把参数 cpu 上注册的 tv1 定时器链表上的定时器迁移到当前 cpu 的定时器链表上。     
    for (i = 0; i < TVR\_SIZE; i++)  
        migrate\_timer\_list(new\_base, old\_base->tv1.vec + i);  
  
    把参数 cpu 上注册的 tv2、 tv3、tv4、tv5 定时器链表上的定时器迁移到当前 cpu 的定时器链表上。  
    for (i = 0; i < TVN\_SIZE; i++) {  
        migrate\_timer\_list(new\_base, old\_base->tv2.vec + i);  
        migrate\_timer\_list(new\_base, old\_base->tv3.vec + i);  
        migrate\_timer\_list(new\_base, old\_base->tv4.vec + i);  
        migrate\_timer\_list(new\_base, old\_base->tv5.vec + i);  
           }  
  
    根据需求为这两个自旋锁 new\_base->lock, old\_base->lock 解锁。  
    double\_spin\_unlock(&new\_base->lock, &old\_base->lock, smp\_processor\_id() < cpu);  
  
    恢复本地 cpu 的中断。  
    local\_irq\_enable();  
    put\_cpu\_var(tvec\_bases);  
}  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
static void migrate\_timer\_list(tvec\_base\_t \*new\_base, struct list\_head \*head)  
{  
    struct timer\_list \*timer;  
  
    如果定时器链表为 空链表，则直接退出。  
    while (!list\_empty(head)) {  
        timer = list\_first\_entry(head, struct timer\_list, entry);  
        detach\_timer(timer, 0);  
        timer\_set\_base(timer, new\_base);  
        internal\_add\_timer(new\_base, timer);  
         }  
}  
＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊  
static inline void double\_spin\_lock(spinlock\_t \*l1, spinlock\_t \*l2,  
                    bool l1\_first)  
    \_\_acquires(l1)  
    \_\_acquires(l2)  
{  
    使用标志 l1\_first 判断先锁定这两个中的那个锁，以防止发生死锁现象。  
    if (l1\_first) {  
        spin\_lock(l1);  
        spin\_lock(l2);  
    } else {  
        spin\_lock(l2);  
        spin\_lock(l1);  
         }  
}  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
# define \_\_acquires(x)  \_\_attribute\_\_((context(x,0,1)))  
＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊  
如果没没有定义这个   CONFIG\_TIMER\_STATS 宏，这个  init\_timer\_stats（） 函数就是个空函数。  
这个      CONFIG\_TIMER\_STATS 宏的作用是是否在  /proc 文件系统中生成  timer\_stats 文件，  
这个文件允许你查看Linux内核里使用定时器的常规事件一些信息。  
通过查看这个文件，你可以看到那些常规事件使用定时器的次数最多，使用的频率是多少。  
更详细的信息可以参考内核源码树下面的  Documentation/filesystems/proc.txt 文件。  
  
如果定义了如下所示：  
这个  init\_timer\_stats() 函数对于每个  CPU 来说初始化了 timer\_stat 的自旋锁。  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
void \_\_init init\_timer\_stats(void)  
{     
    int cpu;  
      
    for\_each\_possible\_cpu(cpu)  
        spin\_lock\_init(&per\_cpu(lookup\_lock, cpu));  
}  
＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊  
在  CPU 管理的通知链表   cpu\_chain 注册通知结构  timers\_nb。  
  
static struct notifier\_block \_\_cpuinitdata timers\_nb = {  
    .notifier\_call  = timer\_cpu\_notify,  
};  
这个通知结构的回调函数为   timer\_cpu\_notify（） ，在上面也讲解过。  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
这个   Raw 类型的通知链表   cpu\_chain 是在  kernel/cpu.c 文件中定义的。  
static \_\_cpuinitdata RAW\_NOTIFIER\_HEAD(cpu\_chain);  
  
int \_\_cpuinit register\_cpu\_notifier(struct notifier\_block \*nb)  
{  
  
    int ret;  
    锁定   cpu\_add\_remove\_lock 互斥锁。  
    mutex\_lock(&cpu\_add\_remove\_lock);  
  
    在   Raw 类型的通知链表   cpu\_chain 上注册通知结构   nb。  
    ret = raw\_notifier\_chain\_register(&cpu\_chain, nb);  
    释放   cpu\_add\_remove\_lock 互斥锁。  
    mutex\_unlock(&cpu\_add\_remove\_lock);  
    return ret;  
}  
＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊  
这个函数   run\_timer\_softirq（） 是 TIMER\_SOFTIRQ 的处理函数。  
这个函数对当前  CPU 到期的定时器进行处理。  
参考《深入理解 linux 内核》第六章。  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
static void run\_timer\_softirq(struct softirq\_action \*h)  
{  
    首先获得到本地 CPU 的定时器链表的   base 地址。  
    tvec\_base\_t \*base = \_\_get\_cpu\_var(tvec\_bases);  
      
  
    这个函数和高精度时钟定时器有关，检测高精度时钟定时器是否  active，如果是  active 的就替换到  
    hres tick机制。参考《linux高精度时钟分析》。  
    hrtimer\_run\_queues();  
  
    检测如果 jiffies大于等于 timer\_jiffies ，说明可能已经有软件时钟到期了，  
    此时就要进行软件时钟的处理，调用函数 \_\_run\_timers（） 函数 进行处理。  
    如果 jiffies 小于 timer\_jiffies ，表明没有软件时钟到期，则不用对软件时钟进行处理。函数返回。  
    if (time\_after\_eq(jiffies, base->timer\_jiffies))  
        \_\_run\_timers(base);  
}  
＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊  
在 TIMER\_SOFTIRQ 软中断的处理函数调用    \_\_run\_timers（） 对到期的定时器进行处理。  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
static inline void \_\_run\_timers(tvec\_base\_t \*base)  
{  
    struct timer\_list \*timer;  
  
    spin\_lock\_irq(&base->lock);  
    while (time\_after\_eq(jiffies, base->timer\_jiffies)) {  
        struct list\_head work\_list;  
        struct list\_head \*head = &work\_list;  
        int index = base->timer\_jiffies & TVR\_MASK;  
  
        if (!index &&  
            (!cascade(base, &base->tv2, INDEX(0))) &&  
                (!cascade(base, &base->tv3, INDEX(1))) &&  
                    !cascade(base, &base->tv4, INDEX(2)))  
            cascade(base, &base->tv5, INDEX(3));  
        ++base->timer\_jiffies;  
        list\_replace\_init(base->tv1.vec + index, &work\_list);  
  
        while (!list\_empty(head)) {  
            void (\*fn)(unsigned long);  
            unsigned long data;  
  
            timer = list\_first\_entry(head, struct timer\_list,entry);  
            fn = timer->function;  
            data = timer->data;  
  
            timer\_stats\_account\_timer(timer);  
  
            set\_running\_timer(base, timer);  
            detach\_timer(timer, 1);  
            spin\_unlock\_irq(&base->lock);  
                                 {  
                int preempt\_count = preempt\_count();  
                fn(data);  
                if (preempt\_count != preempt\_count()) {  
                    printk(KERN\_WARNING "huh, entered %p "  
                           "with preempt\_count %08x, exited"  
                           " with %08x?\n",  
                           fn, preempt\_count,  
                           preempt\_count());  
                    BUG();  
                                             }  
                                }  
            spin\_lock\_irq(&base->lock);  
                    }  
          }  
    set\_running\_timer(base, NULL);  
    spin\_unlock\_irq(&base->lock);  
}  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
   1.  获得 base 的同步锁  
   2.  如果 jiffies 大于等于 timer\_jiffies （当前正要处理的软件时钟的到期时间，  
    说明可能有软件时钟到期了），就一直运行3~7，否则跳转至8  
   3. 计算得到 tv1 的索引，该索引指明当前到期的软件时钟所在 tv1 中的链表（结构参见3.2节），代码：   
    int index = base->timer\_jiffies & TVR\_MASK;  
  
   1.  调用 cascade 函数对软件时钟进行必要的调整（稍后会介绍调整的过程）  
   2.  使得 timer\_jiffies 的数值增加1  
   3.  取出相应的软件时钟链表  
   4.  遍历该链表，对每个元素进行如下操作   
  
    \*  设置当前软件时钟为 base 中正在运行的软件时钟（即保存当前软件时钟到 base-> running\_timer 成员中）  
    \*  将当前软件时钟从链表中删除，即卸载该软件时钟  
    \*  释放锁，执行软件时钟处理程序  
    \*  再次获得锁   
  
   1.  设置当前 base 中不存在正在运行的软件时钟  
   2.  释放锁   
＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊  
问题：  
    1）在   init\_timers\_cpu（） 函数中，为什么不直接使用   per\_cpu（） 函数定义的静态数组  
          而是动态分配呢？  
          在源码注释中，解释由于内存分配器没有工作，所以启动时使用静态分配的，但是在  RCU\_init() 函数  
          中已经使用了  per\_cpu（）  函数了？  
          其实已经在     setup\_per\_cpu\_areas（） 函数为  per\_cpu 数据分配了空间。  
            这是应为  per\_cpu 数据是一个指针，尽管  per\_cpu 中为指针预留了空间，  
            但是指针指向的空间，没有进行分配，都是执行  boot\_tvec\_bases 的。  
            参考《每CPU变量的数据组织和访问》。  
  
    2）这个   \_\_acquires(x) 宏定义的意义？  
  
    3）与  timer\_stat 相关的代码没有看？

**linux-mips启动分析（14）**

在 star\_kernel() 函数中调用     hrtimers\_init（） 函数。  
  
这个       hrtimers\_init（） 函数对高精度时钟进行初始化。  
  
如果没有定义   CONFIG\_HIGH\_RES\_TIMERS 宏，则不支持高精度事件定时器。  
需要需要对高精度事件定时器的支持，需要打开这个   CONFIG\_HIGH\_RES\_TIMERS 宏开关。  
============================================================  
static struct notifier\_block \_\_cpuinitdata hrtimers\_nb = {  
    .notifier\_call = hrtimer\_cpu\_notify,  
};    
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
void \_\_init hrtimers\_init(void)  
{     
  
    初始当前  CPU 的高精度时钟。  
    hrtimer\_cpu\_notify(&hrtimers\_nb, (unsigned long)CPU\_UP\_PREPARE,  
              (void \*)(long)smp\_processor\_id());  
  
    在  CPU 管理子系统通知链表   cpu\_chain 上注册通知结构  hrtimers\_nb。  
    register\_cpu\_notifier(&hrtimers\_nb);  
  
#ifdef CONFIG\_HIGH\_RES\_TIMERS  
    如果打开了高精度事件定时器开关，则初始化   HRTIMER\_SOFTIRQ 软中断，  
    软中断处理函数为   run\_hrtimer\_softirq（）。  
    open\_softirq(HRTIMER\_SOFTIRQ, run\_hrtimer\_softirq, NULL);  
#endif   
}  
＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊  
这个函数是 注册到  CPU 子系统的通知链表的回调函数。  
这个函数主要在新的 CPU 开始工作或者停止工作（支持热插拔）时，通知  hrtimer 管理器，  
使    hrtimer 管理器管理器知道，并进行相应的处理。  
可以支持   CPU\_UP\_PREPARE、CPU\_UP\_PREPARE\_FROZEN、   
  CPU\_DEAD 和  CPU\_DEAD\_FROZEN 类型的事件。  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
static int \_\_cpuinit hrtimer\_cpu\_notify(struct notifier\_block \*self,  
                    unsigned long action, void \*hcpu)  
{  
    long cpu = (long)hcpu;  
  
    switch (action) {  
  
    case CPU\_UP\_PREPARE:  
    case CPU\_UP\_PREPARE\_FROZEN:  
        init\_hrtimers\_cpu(cpu);  
        break;  
  
#ifdef CONFIG\_HOTPLUG\_CPU  
    case CPU\_DEAD:  
    case CPU\_DEAD\_FROZEN:  
        clockevents\_notify(CLOCK\_EVT\_NOTIFY\_CPU\_DEAD, &cpu);  
        migrate\_hrtimers(cpu);  
        break;  
#endif  
  
    default:  
        break;  
         }  
  
    return NOTIFY\_OK;  
}  
＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊  
当 CPU 子系统发生     CPU\_UP\_PREPARE、CPU\_UP\_PREPARE\_FROZEN 事件时，  
下面的  init\_hrtimers\_cpu（） 函数被 CPU 子系统通知链表的回调函数所调用。  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
这个  per\_cpu 变量定义在   kernel/hrtimer.c 文件。  
  
DEFINE\_PER\_CPU(struct hrtimer\_cpu\_base, hrtimer\_bases) =  
{  
  
    .clock\_base =  
            {  
                       {  
            .index = CLOCK\_REALTIME,  
            .get\_time = &ktime\_get\_real,  
            .resolution = KTIME\_LOW\_RES,  
                     },  
                     {  
            .index = CLOCK\_MONOTONIC,  
            .get\_time = &ktime\_get,  
            .resolution = KTIME\_LOW\_RES,  
                    },  
            }  
};  
  
为   struct hrtimer\_cpu\_base 结构体类型，每个 CPU 一个。  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
static void \_\_devinit init\_hrtimers\_cpu(int cpu)  
{  
    取得参数  cpu 的  per\_cpu 数据  hrtimer\_bases。  
    struct hrtimer\_cpu\_base \*cpu\_base = &per\_cpu(hrtimer\_bases, cpu);  
    int i;  
  
    初始化自旋锁。  
    spin\_lock\_init(&cpu\_base->lock);  
    初始化锁的依赖关系。  
    lockdep\_set\_class(&cpu\_base->lock, &cpu\_base->lock\_key);  
      
    for (i = 0; i < HRTIMER\_MAX\_CLOCK\_BASES; i++)  
        cpu\_base->clock\_base[i].cpu\_base = cpu\_base;  
  
    如果没有定义   CONFIG\_HIGH\_RES\_TIMERS 宏，则下面为空函数。  
    hrtimer\_init\_hres(cpu\_base);  
}  
＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊  
问题：  
    1）对高精度事件定时器的处理需要更深入理解？  
    2）资料：Documentation/hrtimers/ 目录下文档？

**linux-mips启动分析（15）**

在 star\_kernel() 函数中调用  init\_timers（） 函数。  
这个函数初始化高优先级和低优先级的  tasklet 软中断处理函数。  
初始化高优先级  tasklet 软中断处理函数 tasklet\_action() 函数。  
初始化低优先级的  tasklet 软中断处理函数 tasklet\_hi\_action() 函数。  
============================================================  
void \_\_init softirq\_init(void)  
{  
    open\_softirq(TASKLET\_SOFTIRQ, tasklet\_action, NULL);  
    open\_softirq(HI\_SOFTIRQ, tasklet\_hi\_action, NULL);  
}  
＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊  
在 linux 系统中可以定义 32 个软中断，每一个软中断都是由 struct softirq\_action 类型的结构体表示的，  
所以在系统中定义了一个  softirq\_vec[32] 数组，用来管理软中断。  
  
struct softirq\_action  
{  
    void    (\*action)(struct softirq\_action \*);  
    void    \*data;  
};  
  
static struct softirq\_action softirq\_vec[32] \_\_cacheline\_aligned\_in\_smp;  
 －－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
这个  open\_softirq() 函数初始化相应软中断的 struct softirq\_action 类型的结构体。  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
void open\_softirq(int nr, void (\*action)(struct softirq\_action\*), void \*data)  
{  
    softirq\_vec[nr].data = data;  
    softirq\_vec[nr].action = action;  
}

**linux-mips启动分析（16）**

在   start\_kernel() 函数中调用  timekeeping\_init（） 函数来初始化系统计时器。  
==============================================================  
  
void \_\_init timekeeping\_init(void)  
{     
    unsigned long flags;  
    unsigned long sec = read\_persistent\_clock();  
          
    使用顺序锁   xtime\_lock 来锁定来包含与定时相关的内核变量。  
    锁定顺序锁   xtime\_lock ，并禁止中断，保存中断标志。  
    write\_seqlock\_irqsave(&xtime\_lock, flags);  
  
    清除  NTP（网络校时协议）接口的变量。  
    ntp\_clear();  
  
    这个   clock 是静态全局变量，是   current clocksource 的指针。  
    clock = clocksource\_get\_next();  
  
    计算每次时钟中断（时钟嘀嗒）给  xtime 变量  tv\_nsec 成员增加的纳秒值。  
    clocksource\_calculate\_interval(clock, NTP\_INTERVAL\_LENGTH);  
  
    读取上一次周期时钟嘀嗒值。  
    clock->cycle\_last = clocksource\_read(clock);  
  
    初始化  wall time 的值。  
    xtime.tv\_sec = sec;  
    xtime.tv\_nsec = 0;  
  
    把  xtime 上的时间调整成  struct timespec 格式，并保存到  wall\_to\_monotonic 变量中。  
    set\_normalized\_timespec(&wall\_to\_monotonic,  -xtime.tv\_sec, -xtime.tv\_nsec);  
  
    释放顺序锁   xtime\_lock ，恢复保存的中断标志。  
    write\_sequnlock\_irqrestore(&xtime\_lock, flags);  
}  
＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊  
这个  ntp\_clear（） 函数清除  NTP（网络校时协议）接口的变量。  
  
这些变量在  adjtimex（） 系统调用时，能够使用到。  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
  
void ntp\_clear(void)  
{  
    time\_adjust = 0;        
    time\_status |= STA\_UNSYNC;        
    time\_maxerror = NTP\_PHASE\_LIMIT;  
    time\_esterror = NTP\_PHASE\_LIMIT;  
  
    ntp\_update\_frequency();  
  
    tick\_length = tick\_length\_base;  
    time\_offset = 0;  
}  
＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊  
这个  tick\_usec 变量表示 一个时钟嘀嗒经过的多少微秒。  
（tick\_usec \* NSEC\_PER\_USEC ）表示一个时钟嘀嗒经过的多少纳秒。  
(tick\_usec \* NSEC\_PER\_USEC \* USER\_HZ)表示    USER\_HZ 个时钟嘀嗒经过的多少纳秒。  
  
这个函数  ntp\_update\_frequency（） 使用 tick\_usec 计算  base tick 的长度，  
这个   tick\_length\_base 变量。  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
static void ntp\_update\_frequency(void)  
{     
    u64 second\_length = (u64)(tick\_usec \* NSEC\_PER\_USEC \* USER\_HZ)  
                << TICK\_LENGTH\_SHIFT;  
  
    加上这个  CLOCK\_TICK\_ADJUST  主要是为了防止    tick\_length\_base 不能够被  HZ 整除。  
    second\_length += (s64)CLOCK\_TICK\_ADJUST << TICK\_LENGTH\_SHIFT;  
    second\_length += (s64)time\_freq << (TICK\_LENGTH\_SHIFT - SHIFT\_NSEC);  
  
    tick\_length\_base = second\_length;  
      
    这个  do\_div（） 是一个宏，这个宏的主要作用是把  64 位的  second\_length 除以  HZ ，并赋值给  
    这个    second\_length  变量，应为这个 do\_div（） 是一个宏，所以 这个    second\_length  变量  
    可以不是指针。  
    do\_div(second\_length, HZ);  
  
    这个  tick\_nsec 表示一个时钟嘀嗒经过的多少纳秒。  
    tick\_nsec = second\_length >> TICK\_LENGTH\_SHIFT;  
  
    把  tick\_length\_base 变量，除以  NTP\_INTERVAL\_FREQ，并把结果赋值给  tick\_length\_base 变量。  
    do\_div(tick\_length\_base, NTP\_INTERVAL\_FREQ);  
}  
＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊  
  
这个函数取得时钟源，时钟源在  arch/mips/kernel/time.c 文件中的   time\_init（） 函数  
调用    init\_mips\_clocksource（） 函数注册的时钟源。  
参考《linux时钟管理机制》。  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
struct clocksource \*clocksource\_get\_next(void)  
{     
    unsigned long flags;  
      
    spin\_lock\_irqsave(&clocksource\_lock, flags);  
  
    检测是否有时钟源，并且已经启动完毕，如果是，则使用注册的时钟源。  
    if (next\_clocksource && finished\_booting) {  
        curr\_clocksource = next\_clocksource;  
        next\_clocksource = NULL;  
           }  
    spin\_unlock\_irqrestore(&clocksource\_lock, flags);  
  
    如果不是，则使用默认的时钟源。  
    return curr\_clocksource;  
}  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
这    curr\_clocksource 和      next\_clocksource 变量都是静态全局变量。  
在   kernel/time/clocksource.c 文件中定义。  
  
static struct clocksource \*curr\_clocksource = &clocksource\_jiffies;  
static struct clocksource \*next\_clocksource;  
＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊  
在  timekeeping\_init（） 函数中，主要用到的新的数据结构为  struct clocksource，  
下面对这个结构的成员进行一下讲解。  
  
struct clocksource {  
    char \*name;            时钟的名称  
    struct list\_head list;     时钟注册链表  
    int rating;  
    cycle\_t (\*read)(void);  读取精确的单调时间计数的接口,  
    cycle\_t mask;  
    u32 mult;  
    u32 shift;  
    unsigned long flags;  
    cycle\_t (\*vread)(void);  
    void (\*resume)(void);  
  
    cycle\_t cycle\_interval;  
    u64 xtime\_interval;  
    cycle\_t cycle\_last \_\_\_\_cacheline\_aligned\_in\_smp;  
    u64 xtime\_nsec;  
    s64 error;  
  
#ifdef CONFIG\_CLOCKSOURCE\_WATCHDOG  
    /\* Watchdog related data, used by the framework \*/  
    struct list\_head wd\_list;  
    cycle\_t wd\_last;  
#endif  
}  ;  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
这个     struct clocksource 是一个硬件时钟的抽象化，提供了硬件时钟的底层操作。  
  
最重要的成员是  read() 函数指针， cycle\_last  和  cycle\_interval 成员，  
分别定义了读取硬件时钟寄存器当前计数值接口，  保存上一次周期计数值和每个tick周期间隔值。  
这个结构的数组，都是以计数值（cycle） 为单位， 而不是  nsec，sec  和  jiffies 等。  
read()是整个  kernel  读取精确的单调时间计数的接口，内核根据这个函数返回的值，  
用来计算  jiffies  和   xtime。  
＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊  
问题：  
    1） NTP（网络校时协议）接口变量的使用？  
    2）这个  tick\_length\_base 变量的意义？  
    3）这个  clocksource\_calculate\_interval（） 函数的意义？

**linux-mips启动分析（17）**

在  start\_kernle() 函数中调用  time\_init() 函数，初始化系统时钟源，  
对  MIPS 体系结构的    clocksource\_mips  时钟源进行配置，和注册。  
  
这段代码是体系结构相关的，对于  MIPS 体系结构的位于   arch/mips/kernel/time.c 文件中。  
=============================================================  
  
void \_\_init time\_init(void)  
{  
  
    这个   board\_time\_init 变量是一个函数指针，在  plat\_mem\_setup（） 函数进行的赋值。  
    参考《linux-mips启动分析（4）》。  
    if (board\_time\_init)  
        board\_time\_init();  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
    这个   rtc\_mips\_set\_mmss 变量也是一个函数指针，如果没有设置，  
    则设置为     rtc\_mips\_set\_time（） 函数。  
    定义如下所示（在   arch/mips/kernel/time.c 文件中）：  
        unsigned long (\*rtc\_mips\_get\_time)(void) = null\_rtc\_get\_time;  
        int (\*rtc\_mips\_set\_time)(unsigned long) = null\_rtc\_set\_time;  
        int (\*rtc\_mips\_set\_mmss)(unsigned long);  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
    if (!rtc\_mips\_set\_mmss)  
        rtc\_mips\_set\_mmss = rtc\_mips\_set\_time;  
  
    xtime.tv\_sec = rtc\_mips\_get\_time();  
    xtime.tv\_nsec = 0;  
  
    把  xtime 中的时间转换成   timespec 格式，保存到   wall\_to\_monotonic 变量中。  
    set\_normalized\_timespec(&wall\_to\_monotonic, -xtime.tv\_sec, -xtime.tv\_nsec);  
  
    设置时钟源的   truct clocksource clocksource\_mips 的  read 函数指针。  
  
    if (!cpu\_has\_counter && !clocksource\_mips.read)  
        clocksource\_mips.read = null\_hpt\_read;  
    else if (!mips\_hpt\_frequency && !mips\_timer\_state) {  
        if (!clocksource\_mips.read)  
            clocksource\_mips.read = c0\_hpt\_read;  
    } else {  
        if (!clocksource\_mips.read) {  
            clocksource\_mips.read = c0\_hpt\_read;  
            if (!mips\_timer\_state) {  
                mips\_timer\_ack = c0\_timer\_ack;  
                cycles\_per\_jiffy = (mips\_hpt\_frequency + HZ / 2) / HZ;  
                c0\_hpt\_timer\_init();  
                                }  
                     }  
        if (!mips\_hpt\_frequency)  
            mips\_hpt\_frequency = calibrate\_hpt();  
  
        打印  timer 的计算时钟频率。  
        printk("Using %u.%03u MHz high precision timer.\n";  
          }  
  
    if (!mips\_timer\_ack)  
        mips\_timer\_ack = null\_timer\_ack;  
  
  
    这个   plat\_timer\_setup(） 函数是体系结构相关的，初始化硬件时钟。  
    参数是时钟中断   irqaction 描述符。  
    plat\_timer\_setup(&timer\_irqaction);  
    init\_mips\_clocksource();  
}  
＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊  
这个   board\_time\_init 变量是一个函数指针，在  plat\_mem\_setup（） 函数进行的赋值。  
参考《linux-mips启动分析（4）》。  
  
  board\_time\_init = cq8401\_time\_init;  
所以实际调用的为    cq8401\_time\_init（） 函数，这个函数是移植相关的，  
在移植内核时，写的代码。  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
unsigned int mips\_hpt\_frequency;  
int (\*mips\_timer\_state)(void);  
void (\*mips\_timer\_ack)(void);  
  
struct clocksource clocksource\_mips = {  
    .name       = "MIPS",  
    .mask       = CLOCKSOURCE\_MASK(32),  
    .flags      = CLOCK\_SOURCE\_IS\_CONTINUOUS,  
};  
这些变量都是在    arch/mips/kernel/time.c 文件中定义的。  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
  
void \_\_init cq8401\_time\_init(void)  
{  
    mips\_hpt\_frequency = OST\_CLK;  
    mips\_timer\_state = clxsoc\_timer\_state;  
    mips\_timer\_ack = taurus\_timer\_ack;  
    clocksource\_mips.read = clxsoc\_hpt\_read;  
}  
＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊  
这个   plat\_timer\_setup(） 函数是体系结构相关的，对硬件时钟进行初始化。  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
void \_\_init plat\_timer\_setup(struct irqaction \*irq)  
{  
  
    初始化全局   jiffies 变量为  0 。  
    jiffies = 0;  
  
    取得  1  个 时钟嘀嗒，需要多少个   OST 时钟周期。  
    latch = (OST\_CLK + (HZ>>1)) / HZ;  
  
    注册时钟中断，这个  setup\_irq(）函数为  request\_irq() 的内部函数， 参考下面讲解。  
    setup\_irq(IRQ\_OST0, irq);  
  
    \_\_ost\_disable\_channel(0);  
  
    设置 1 个时钟嘀嗒需要的   OST 时钟周期。  
    \_\_ost\_set\_reload(0, latch);  
    \_\_ost\_set\_count(0, latch);  
  
    使能中断，并选择  OST 的时钟源。  
#if USE\_EXTAL\_CLK  
    \_\_ost\_set\_mode(0, OST\_TCSR\_UIE | OST\_TCSR\_CKS\_EXTAL);  
#else  
    \_\_ost\_set\_mode(0, OST\_TCSR\_UIE | OST\_TCSR\_CKS\_PCLK\_4);  
#endif  
  
    等待  OST 设置好，后开始工作。  
    while(\_\_ost\_is\_busy(0))  
        /\* nothing \*/ ;  
    \_\_ost\_enable\_channel(0);  
}  
＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊  
这个  setup\_irq(） 是注册中断处理的内部函数，  request\_irq()  函数就是由这个函数构成的。  
这个函数由两个参数，一个是中断号，另一个为  irqaction 描述符（对中断进行处理）。  
  
时钟中断的     irqaction 描述符定义在   arch/mips/cq8401/common/time.c 文件中，如下所示：  
  
static struct irqaction timer\_irqaction = {  
    .handler = timer\_interrupt,  
    .flags = IRQF\_DISABLED | IRQF\_PERCPU,  
    .name = "timer",  
};  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
int setup\_irq(unsigned int irq, struct irqaction \*new)  
{     
    在  linux 内核中是由  irq 描述符数组   irq\_desc［］ 进行的管理。  
    由中断号取得  时钟中断所使用的  irq 描述符。  
    struct irq\_desc \*desc = irq\_desc + irq;  
    struct irqaction \*old, \*\*p;  
    const char \*old\_name = NULL;  
    unsigned long flags;  
    int shared = 0;  
      
  
    进行参数检测。  
    if (irq >= NR\_IRQS)  
        return -EINVAL;  
    if (desc->chip == &no\_irq\_chip)  
        return -ENOSYS;  
  
    如果中断的参数设置了   IRQF\_SAMPLE\_RANDOM 标志，对内核熵池有贡献。  
    if (new->flags & IRQF\_SAMPLE\_RANDOM) {  
        rand\_initialize\_irq(irq);  
          }  
  
    锁定当前   irq 描述符的自旋锁，并保存中断标志。  
    spin\_lock\_irqsave(&desc->lock, flags);  
    p = &desc->action;  
    old = \*p;  
  
    检测这个  irq 号上是否已经注册了中断，如果  old 不等于  NULL，则已经注册了中断。  
    if (old) {  
  
        当已经注册了中断时，检测原来的中断处理函数，和新的中断处理函数是否支持中断共享。  
        if (!((old->flags & new->flags) & IRQF\_SHARED) ||  
            ((old->flags ^ new->flags) & IRQF\_TRIGGER\_MASK)) {  
            old\_name = old->name;  
            goto mismatch;  
                      }  
  
#if defined(CONFIG\_IRQ\_PER\_CPU)  
        当已经注册了中断时，所有的中断必须符合这个   IRQF\_PERCPU 标志，否则出错。  
        if ((old->flags & IRQF\_PERCPU) !=  
            (new->flags & IRQF\_PERCPU))  
            goto mismatch;  
#endif  
  
        当已经注册了中断时，并符合上面的条件时，把中断处理描述符加入  action 链表的最后。  
        do {  
            p = &old->next;  
            old = \*p;  
        } while (old);  
        shared = 1;  
           }  
  
    \*p = new;  
  
    检测新注册的中断处理函数的标志，设置中断描述符的标志。  
    这个标志表示这个中断处理函数不能够被中断平衡所影响。  
    if (new->flags & IRQF\_NOBALANCING)  
        desc->status |= IRQ\_NO\_BALANCING;  
  
    如果  irq 号上没有注册中断，则执行下面的操作，  
    如果已经注册了中断，则   shared = 1  跳过这个条件判断。  
    if (!shared) {  
  
        检测中断的底层操作  enable、disable、startup、shutdown 等函数是否存在，  
        如果不存在使用默认的函数。  
        irq\_chip\_set\_defaults(desc->chip);  
  
#if defined(CONFIG\_IRQ\_PER\_CPU)  
        这个  IRQF\_PERCPU 标志表示这个中断是所有   CPU 共享的。  
        if (new->flags & IRQF\_PERCPU)  
            desc->status |= IRQ\_PER\_CPU;  
#endif  
  
        检测是否设置了中断触发模式。  
        if (new->flags & IRQF\_TRIGGER\_MASK) {  
            if (desc->chip && desc->chip->set\_type)  
                desc->chip->set\_type(irq, new->flags & IRQF\_TRIGGER\_MASK);  
            else  
                printk( 打印警告信息）  ;  
        } else  
            compat\_irq\_chip\_set\_default\_handler(desc);  
  
        清除   irq 描述符的   IRQ\_AUTODETECT 、IRQ\_WAITING 和  IRQ\_INPROGRESS 标志位。  
        desc->status &= ~(IRQ\_AUTODETECT | IRQ\_WAITING | IRQ\_INPROGRESS);  
  
        检测这个  irq 号是否在注册时自动使能的，是的话则执行下面。  
        if (!(desc->status & IRQ\_NOAUTOEN)) {  
            这个   depth 表示中断关闭的深度，  0  表示中断使能。  
            desc->depth = 0;  
            desc->status &= ~IRQ\_DISABLED;  
            if (desc->chip->startup)  
                desc->chip->startup(irq);  
            else  
                desc->chip->enable(irq);  
        } else  
            表示中断现在是关闭的。  
            desc->depth = 1;  
            }  
  
    这个   irq\_count 表示中断发生次数，这个  irqs\_unhandled  表示中断相应次数。  
    desc->irq\_count = 0;  
    desc->irqs\_unhandled = 0;  
    解锁，恢复中断标识。  
    spin\_unlock\_irqrestore(&desc->lock, flags);  
  
    new->irq = irq;  
  
    在  proc 文件系统中进行注册，在此不进行详细讲解。  
    register\_irq\_proc(irq);  
    new->dir = NULL;  
  
    这个也和 proc 文件系统有关，在此也不进行详细讲解。  
    register\_handler\_proc(irq, new);  
  
    return 0;  
  
mismatch:  
#ifdef CONFIG\_DEBUG\_SHIRQ  
    if (!(new->flags & IRQF\_PROBE\_SHARED)) {  
        printk(KERN\_ERR "IRQ handler type mismatch for IRQ %d\n", irq);  
        if (old\_name)  
            printk(KERN\_ERR "current handler: %s\n", old\_name);  
        dump\_stack();  
    }  
#endif  
    spin\_unlock\_irqrestore(&desc->lock, flags);  
    return -EBUSY;  
}  
＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊  
这个   init\_mips\_clocksource（） 函数被   time\_init（） 函数所调用，  
  
这个函数主要是注册时钟源，这个时钟源是硬件时钟的抽象。  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
static void \_\_init init\_mips\_clocksource(void)  
{  
    u64 temp;  
    u32 shift;  
  
    进行检测，这个   mips\_hpt\_frequency 表示硬件时钟的计数频率。  
    这个   clocksource\_mips.read 为读出硬件时钟的计数接口函数。  
    if (!mips\_hpt\_frequency || clocksource\_mips.read == null\_hpt\_read)  
        return;  
  
    计算时钟源的等级，这个等级主要是硬件时钟的计数频率有关，频率越高，则等级越高，时钟精度越高。  
    clocksource\_mips.rating = 200 + mips\_hpt\_frequency / 10000000;  
  
    这个   NSEC\_PER\_SEC 表示每秒钟的纳秒数，  
    for (shift = 32; shift > 0; shift--) {  
      
        这个  temp 表示  2 的   shift 次方秒钟的纳秒数。  
        temp = (u64) NSEC\_PER\_SEC << shift;  
  
        这个  do\_div(） 把  temp 除以  mips\_hpt\_frequency ，并把结构给   temp。  
        这是一个宏，所以不使用   temp 的指针。  
        除以   mips\_hpt\_frequency，相当于乘以硬件时钟    1    周期的时间。  
        do\_div(temp, mips\_hpt\_frequency);  
        if ((temp >> 32) == 0)  
            break;  
           }  
    clocksource\_mips.shift = shift;  
    clocksource\_mips.mult = (u32)temp;  
  
  
    注册时钟源   clocksource\_mips。  
    参考《 linux时钟管理机制 》。  
    clocksource\_register(&clocksource\_mips);  
}  
＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊  
  
问题：  
    1）在中断注册中的   register\_irq\_proc（） 和    register\_handler\_proc() 函数的处理过程。  
    2）这个时钟源的   rating 等级有什么作用？  
        答：这个等级主要是硬件时钟的计数频率有关，频率越高，则等级越高，时钟精度越高。  
    3）这个时钟源结构的   shift 和  mult  的作用？

**linux-mips启动分析（18）**

在  start\_kernel() 函数调用   profile\_init（） 函数，这个函数对内核的  profile 功能  
进行初始化，这是一个内核调式工具，通过这个可以发现内核在内核态的什么地方花费时间最多，  
即发现内核的“hot spot”——执行最频繁的内核代码。  
  
这个  profile\_init()  分配一段内存，用来存放  profile 信息，每条指令都有一个计数器。  
========================================================  
  
内核代码监控使用的全局变量：  
static atomic\_t \*prof\_buffer;  
static unsigned long prof\_len, prof\_shift;  
  
int prof\_on \_\_read\_mostly;  
EXPORT\_SYMBOL\_GPL(prof\_on);  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
void \_\_init profile\_init(void)  
{     
    这个  prof\_on 是一个全局变量，默认为  0 ，表示了  profile 的类型，  
    这个  prof\_on 变量在  profile\_setup（） 函数进行赋值。  
    if (!prof\_on)   
        return;  
      
    这两个变量   \_stext 和  \_etext 在链接文件  arch/mips/kernel/vmlinux.lds 中定义。  
    表示代码段的开始地址可结束地址。  
    通过    prof\_shift 来决定分配  profile 空间的大小。  
    prof\_len = (\_etext - \_stext) >> prof\_shift;  
  
    分配  profile 空间。  
    prof\_buffer = alloc\_bootmem(prof\_len\*sizeof(atomic\_t));  
}  
＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊  
这个    profile\_setup（） 函数比较特别，并不是在这里进行的调用，  
而是在  do\_early\_param（） 函数执行时进行了调用。  
参考《linux-mips启动分析（4）》。  
  
因为内核定义了   \_\_setup("profile=", profile\_setup);  
把这个  profile\_setup（） 函数 ，链接入     .init.setup 段，  
在   do\_early\_param（） 函数中，判断内核的启动参数命令行，如果有  
“profile=” 字符串，则调用     profile\_setup（） 函数。  
  
    参考《Linux启动参数及实现》和 《启动时内核参数解析》  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
这个函数主要对    prof\_on  和   prof\_shift 变量进行赋值。  
  
定义了  prof\_on 全局变量，表示  profiling 的类型，有如下四种：  
  
#define CPU\_PROFILING   1  
#define SCHED\_PROFILING 2  
#define SLEEP\_PROFILING 3  
#define KVM\_PROFILING   4  
  
这个    prof\_shift 变量，是静态全局变量，表示  prefile 时，监控的内核代码的精度。  
也就是几条指令使用一个计数器。  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
static int \_\_init profile\_setup(char \* str)  
{  
    static char \_\_initdata schedstr[] = "schedule";  
    static char \_\_initdata sleepstr[] = "sleep";  
    static char \_\_initdata kvmstr[] = "kvm";  
    int par;  
  
    当命令行启动参数为    profile=schedule,[n] （n 为一个数字）  时。  
    if (!strncmp(str, sleepstr, strlen(sleepstr))) {  
        prof\_on = SLEEP\_PROFILING;  
        if (str[strlen(sleepstr)] == ',')  
            str += strlen(sleepstr) + 1;  
      
        取得  schedule,[n] 中  n 数值的大小，并设置  prof\_shift 变量。  
        if (get\_option(&str, &par))  
            prof\_shift = par;  
  
    当命令行启动参数为    profile=sleep,[n] （n 为一个数字） 时。  
    } else if (!strncmp(str, schedstr, strlen(schedstr))) {  
        prof\_on = SCHED\_PROFILING;  
        if (str[strlen(schedstr)] == ',')  
            str += strlen(schedstr) + 1;  
  
        取得 sleep,[n] 中  n 数值的大小，并设置  prof\_shift 变量。  
        if (get\_option(&str, &par))  
            prof\_shift = par;  
  
    当命令行启动参数为    profile=kvm,[n] （n 为一个数字） 时。  
    } else if (!strncmp(str, kvmstr, strlen(kvmstr))) {  
        prof\_on = KVM\_PROFILING;  
        if (str[strlen(kvmstr)] == ',')  
            str += strlen(kvmstr) + 1;  
  
        取得   kvm,[n] 中  n 数值的大小，并设置  prof\_shift 变量。  
        if (get\_option(&str, &par))  
            prof\_shift = par;  
  
    当命令行启动参数为    profile=[n] （n 为一个数字） 时。  
    } else if (get\_option(&str, &par)) {  
        prof\_shift = par;  
        prof\_on = CPU\_PROFILING;  
           }  
    return 1;  
}  
＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊  
参考：  
  
内核版本2.6.18-RC7  
profile只是内核的一个调试性能的工具，这个可以通过menuconfig中的Instrumentation Support->profile打开。  
1. 如何使用profile：  
首先确认内核支持profile，然后在内核启动时加入以下参数：profile=1或者其它参数， 新的内核支持profile=schedule 1  
2. 内核启动后会创建/proc/profile文件，这个文件可以通过readprofile读取，  
如readprofile -m /proc/kallsyms | sort -nr > ~/cur\_profile.log,  
或者readprofile -r -m /proc/kallsyms |sort -nr,  
或者readprofile -r && sleep 1 && readprofile -m /proc/kallsyms |sort -nr >~/cur\_profile.log  
3. 读取/proc/profile可获得哪些内容？  
根据启动配置profile＝？的不同，获取的内容不同：  
如果配置成profile=? 可以获得每个函数执行次数，用来调试函数性能很有用  
如果设置成profile=schedule ?可以获得每个函数调用schedule的次数，用来调试schedule很有用  
  
profile的实现：  
在内核中创建一个/proc/profile接口，在系统启动时用profile\_init()分配好存放profile信息的内存，每条指令都有一个计数器。  
如果设置的是profile=? 统计每条指令执行的次数。在时钟中断中调用        profile\_tick(CPU\_PROFILING, regs)，  
将当前指令regs->eip的计数值+1。这个统计有点不准，因为一个jiffies之间，可能执行很多函数，  
而统计的只是恰好发生时钟中断时的那个函数。但取样点多了，这些信息还是能说明问题。  
如果设置的是profile=schedule ? 统计每个指令调用schedule()的次数，  
在schedule()中调用profile\_hit(SCHED\_PROFILING, \_\_builtin\_return\_address(0));  
其实真正调用schedule的指令只有有限的几个，但这些信息可以获得调度点的精确信息。  
  
profile\_hit()的作用是将当前指令的计数值加1  
profile\_tick()是在每个时钟tick的时候将响应的指令计数值加1  
time\_hook 一般被其它profile工具，如oprofile用来在每次中断发生时，添加自己的处理函数。  
  
profile信息其实包括任务的所有统计信息，所以可以用profile\_event\_register()在任务退出或者用户空间内存释放时，挂载自己的回调函数，以统计这些信息。  
  
profile信息的统计在smp和up下不同，即profile\_hit的实现不同，smp的实现中有一个PerCPU cache，  
这可避免多个CPU在profile统计时效率低下问题。具体可以察看源代码kernel/profile.c, 可以看看driver/oprofile的实现。

**linux-mips启动分析（19）**

在  start\_kernel() 函数中调用    console\_init() 函数，这个    console\_init() 函数  
初始化内核显示终端，在这里仅仅进行一些初级的初始化。  
  
应用程序通过终端接口设备使用特定的接口规程与终端进行交互，与操作系统内核本身交互的终端称为控制台，  
它可以是内核本身的内部显示终端，也可以是通过串口连接的外部哑终端。   
由于大多数情况下控制台都是内核显示终端，因此内核显示终端也常常直接称为控制台。  
内核终端对用户来说具有若干个虚拟终端子设备，它们共享同一物理终端，  
但同一时刻只能有一个虚拟终端操作硬件屏幕。  
宏  CONFIG\_VT 的意思是否支持虚拟终端。  
当配置了宏  CONFIG\_VGA\_CONSOLE 时为内核本身的内部显示终端。  
当配置了宏  CONFIG\_DUMMY\_CONSOLE 时为通过串口连接的外部哑终端。   
  
内核终端的主设备号为  4( TTY\_MAJOR) ，1  至  63  号子设备表示不同的虚拟终端，  
 0   号子设备代表当前活动的虚拟终端， 内核终端的物理显示设备接口用   struct consw  结构描述，  
在配置了VGA文本控制台时   (   CONFIG\_VGA\_CONSOLE），  
系统的缺省显示设备指针   conswitchp  指向  vga\_con 设备。  
在配置串口连接的外部哑终端文本控制台时   (  CONFIG\_DUMMY\_CONSOLE），  
系统的缺省显示设备指针   conswitchp  指向  dummy\_con 设备。  
参考《linux-mips启动分析（3）》。  
=============================================================  
void \_\_init console\_init(void)  
{  
    这个  initcall\_t 类型是一个函数指针的类型。  
    initcall\_t \*call;  
  
    (void) tty\_register\_ldisc(N\_TTY, &tty\_ldisc\_N\_TTY);  
      
    这个  \_\_con\_initcall\_start 变量在链接文件中定义，  
    表示了  .con\_initcall.init 段的其实地址。  
    call = \_\_con\_initcall\_start;  
  
    依次调用  .con\_initcall.init 段的函数。  
    通过分析在这个   .con\_initcall.init 段有下面的三个函数。  
        con\_init()   
        serial8250\_console\_init()  
        early\_uart\_console\_init()  
  
    while (call < \_\_con\_initcall\_end) {  
        (\*call)();  
        call++;  
           }  
}  
＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊  
这个   tty\_register\_ldisc() 函数，初始化一个  tty\_ldiscs［］ 数字中的一个成员。  
  
static DEFINE\_SPINLOCK(tty\_ldisc\_lock);  
static DECLARE\_WAIT\_QUEUE\_HEAD(tty\_ldisc\_wait);  
static struct tty\_ldisc tty\_ldiscs[NR\_LDISCS];   
  
使用这个   tty\_ldisc\_lock 自旋锁来保护  tty\_ldiscs［］ 数组。  
这个   tty\_ldiscs［］ 数组表示了控制终端的线路规范和  TTY 例程之间的接口。  
用来在文件接口与输入设备和输出设备之间进行调控。  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
在  linux 系统中，打开终端的文件操作表为  tty\_fops，  
终端的打开结构用  tty\_struct  结构描述  
文件结构的   private\_data  指针指向打开的终端结构，  
如果当前进程的终端指针  (  current->tty )  指向某个打开的终端结构，  
则表示该进程与此终端相关联，tty\_driver  结构描述终端的输出设备，  
  
终端线路规程 tty\_ldiscs[NR\_LDISCS]  用来在文件接口与输入设备和输出设备之间进行调控。  
TTY规程  ( N\_TTY )   用于连接终端输入驱动设备和终端显示驱动设备。  
结构名中的  “ldisc” 是 “Line Discipline” 的缩写，表示“链录规则”的意思。  
这个结构体中，不但有供上层调用的  open、read、write 等等，还有供下层调用的函数指针  
   receive\_buf、receive\_room 以及  write\_wakeup 等等。  
  
使用不同的  ldisc 结构体，表示不同的链路规则。  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
int tty\_register\_ldisc(int disc, struct tty\_ldisc \*new\_ldisc)  
{  
    unsigned long flags;  
    int ret = 0;  
  
    if (disc < N\_TTY || disc >= NR\_LDISCS)  
        return -EINVAL;  
      
    spin\_lock\_irqsave(&tty\_ldisc\_lock, flags);  
  
    结构体整体赋值，这个   struct tty\_ldisc 结构体的成员赋值为  tty\_ldisc\_N\_TTY 的成员值。  
    tty\_ldiscs[disc] = \*new\_ldisc;  
    tty\_ldiscs[disc].num = disc;  
    tty\_ldiscs[disc].flags |= LDISC\_FLAG\_DEFINED;  
    tty\_ldiscs[disc].refcount = 0;  
    spin\_unlock\_irqrestore(&tty\_ldisc\_lock, flags);  
  
    return ret;  
}  
＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊  
这个  con\_init() 函数位于内核镜像的    .con\_initcall.init 段，被 console\_init() 函数  
所调用。  
这个函数在   drivers/char/vt.c 文件中定义。  
  
这个函数初始化  console 中断。  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
static int \_\_init con\_init(void)  
{  
    const char \*display\_desc = NULL;  
    struct vc\_data \*vc;  
    unsigned int currcons = 0, i;  
  
    取得信号量，这个信号量用来保护  console 子系统，和   console\_drivers 链表。  
    acquire\_console\_sem();  
  
    这个  conswitchp 变量是一个   struct  consw 结构的全局变量。  
    表示 内核终端的物理显示设备接口的结构描述。  
    这个     conswitchp 变量在   setup\_arch(） 函数中进行赋值为   vga\_con 或者  dummy\_con。  
    参考《linux-mips启动分析（3）》。  
    在这里假设为  conswitchp ＝＝  dummy\_con。  
    这个   dummy\_con 变量在  drivers/video/console/dummycon.c 文件中定义。  
    if (conswitchp)  
        display\_desc = conswitchp->con\_startup();  
  
    这个  display\_desc 指针为控制台的描述，如果为  NULL，则内核没有配置控制台。  
    if (!display\_desc) {  
        fg\_console = 0;  
        release\_console\_sem();  
        return 0;  
           }  
  
    这个   registered\_con\_driver[] 数组表示已经注册了的  console 驱动数组。  
    这个一个   struct con\_driver 结构全局变量。  
  
    在    registered\_con\_driver[] 数组中注册一个  console 驱动。  
    for (i = 0; i < MAX\_NR\_CON\_DRIVER; i++) {  
        struct con\_driver \*con\_driver = &registered\_con\_driver[i];  
        if (con\_driver->con == NULL) {  
            con\_driver->con = conswitchp;  
            con\_driver->desc = display\_desc;  
            con\_driver->flag = CON\_DRIVER\_FLAG\_INIT;  
            con\_driver->first = 0;  
            con\_driver->last = MAX\_NR\_CONSOLES - 1;  
            break;  
                     }  
           }  
  
    这个  MAX\_NR\_CONSOLES 等于  63，表示不同的虚拟终端。  
    这个  con\_driver\_map[] 数组为  struct consw 结构体的指针数组。  
    for (i = 0; i < MAX\_NR\_CONSOLES; i++)  
        con\_driver\_map[i] = conswitchp;  
  
    这个  blankinterval 变量表示刷新的间隔时间。  
    这个   blank\_state  变量表示刷新控制的状态。  
    这个 console\_timer 为刷新操作的定时器。  
    if (blankinterval) {  
        blank\_state = blank\_normal\_wait;  
        mod\_timer(&console\_timer, jiffies + blankinterval);  
           }  
  
    这个  vc\_cons[] 数组为  struct vc 结构的数组，这个数组对控制台的显示进行控制。  
    分配虚拟控制台的参数结构空间。并对虚拟控制台的参数进行初始化。  
    这个  MIN\_NR\_CONSOLES 为 1， 表示建立一个虚拟终端。  
    这个  struct  vc\_data 实际是一个虚拟缓冲区。  
    for (currcons = 0; currcons < MIN\_NR\_CONSOLES; currcons++) {  
        vc\_cons[currcons].d = vc = alloc\_bootmem(sizeof(struct vc\_data));  
  
        初始化虚拟控制台的工作结构体。这个工作结构体的处理函数为   vc\_SAK（） 函数。  
        INIT\_WORK(&vc\_cons[currcons].SAK\_work, vc\_SAK);  
        visual\_init(vc, currcons, 1);  
        vc->vc\_screenbuf = (unsigned short \*)alloc\_bootmem(vc->vc\_screenbuf\_size);  
        vc->vc\_kmalloced = 0;  
        vc\_init(vc, vc->vc\_rows, vc->vc\_cols, currcons || !vc->vc\_sw->con\_save\_screen);  
           }  
    currcons = fg\_console = 0;  
  
    用以识别当前的前景控制台  
    master\_display\_fg = vc = vc\_cons[currcons].d;  
  
    设置终端的起始显示点。  
    set\_origin(vc);  
  
    保存硬件屏幕到终端屏幕缓冲区。  
    save\_screen(vc);  
  
    定位光标。  
    gotoxy(vc, vc->vc\_x, vc->vc\_y);  
  
    清除光标到屏幕尾部的字符。  
    csi\_J(vc, 0);  
  
    使用终端屏幕缓冲区重新绘制屏幕。  
    update\_screen(vc);  
    printable = 1;  
  
    释放信号量。  
    release\_console\_sem();  
  
#ifdef CONFIG\_VT\_CONSOLE  
    将内核终端注册到内核控制台。  
    register\_console(&vt\_console\_driver);  
#endif  
    return 0;  
}  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
这个  vt\_console\_driver 在  drivers/char/vt.c 中定义。  
  
static struct console vt\_console\_driver = {  
    .name       = "tty",  
    .write      = vt\_console\_print,  
    .device     = vt\_console\_device,  
    .unblank    = unblank\_screen,  
    .flags      = CON\_PRINTBUFFER,  
    .index      = -1,  
};    
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
在  con\_init() 函数中调用  conswitchp->con\_startup() 函数，  
实际为    dummy\_con 结构的       .con\_startup =  dummycon\_startup() 函数。  
  
static const char \*dummycon\_startup(void)  
{     
    return "dummy device";  
}     
返回一个字符串的地址起始地址。  
＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊  
这个  serial8250\_console\_init() 函数位于内核镜像的    .con\_initcall.init 段，  
被 console\_init() 函数所调用。  
这个函数在  drivers/serial/8250.c 文件中定义。  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
  
static int \_\_init serial8250\_console\_init(void)  
{     
    初始化  8250 串口。  
    serial8250\_isa\_init\_ports();  
    将  8250 终端注册到内核控制台。  
    register\_console(&serial8250\_console);  
    return 0;  
}  
＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊  
这个 early\_uart\_console\_init() 函数位于内核镜像的    .con\_initcall.init 段，  
被 console\_init() 函数所调用。  
这个函数在    drivers/serial/8250\_early.c 文件中定义。  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
static int \_\_init early\_uart\_console\_init(void)  
{     
    if (!early\_uart\_registered) {  
  
        将串口控制台注册到内核控制台。  
        register\_console(&early\_uart\_console);  
        early\_uart\_registered = 1;  
           }  
    return 0;  
}  
＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊  
通过分析上面的代码发现在这三个函数当中，  
  
        con\_init()   
        serial8250\_console\_init()  
        early\_uart\_console\_init()  
主要是调用  register\_console(）函数，  在内核控制台注册终端驱动。  
  
register\_console确实是把console结构体加入到console\_drivers，需要注意的是，  
 console\_drivers是一个队列，那么也就意味着可以在该队列中有多个console。  
如果console\_drivers中注册有多个设备，那么究竟是将printk的信息输出到哪个呢？  
事实上，printk要判断消息级别是否小于console\_loglevel，如果小于的话，就发送  
到   console\_drivers  中的所有设备中去。  
到这里，我们已经基本清晰了， register\_console() 函数的功用，注册到  
  console\_drivers  中的的  console，  仅仅是为了显示  printk（） 的信息，  
而不是传统意义所指的控制台—在众多终端中唯一可以进入单用户模式进行系统维护的某个终端设备。  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
void register\_console(struct console \*console)  
{  
    int i;  
    unsigned long flags;  
    struct console \*bootconsole = NULL;  
  
    这个   console\_drivers 变量是一个  struct console 指针类型的全局变量。  
    在    struct console  结构体中，有一个  next 成员，指向下一个  console 。  
  
    如果这个   console\_drivers  链表不是  NULL 的。  
    if (console\_drivers) {  
  
        而要注册的   console 为  boot 阶段的  console ，则直接返回，而不注册。  
        if (console->flags & CON\_BOOT)  
            return;  
  
        如果已经注册了  boot 阶段的  console ，则取得这个  console 的指针。  
        在注册  first real console 时，这个    boot 阶段的  console 会被注销掉。  
        if (console\_drivers->flags & CON\_BOOT)  
            bootconsole = console\_drivers;  
           }  
  
    如果已经注册了   boot 阶段的  console，或者已经注册了  console ，或者  ＊＊＊＊＊＊？  
    if (preferred\_console < 0 || bootconsole || !console\_drivers)  
        preferred\_console = selected\_console;  
  
    这个  preferred\_console 和   selected\_console 变量在   console\_setup() 函数中设置。  
    if (preferred\_console < 0) {  
        if (console->index < 0)  
            console->index = 0;  
  
        如果这个需要注册的  console 的 setup 函数成员，可以执行，则设置这个  console 可以注册。  
        if (console->setup == NULL || console->setup(console, NULL) == 0) {  
            console->flags |= CON\_ENABLED | CON\_CONSDEV;  
            preferred\_console = 0;  
                     }  
           }  
  
    这个循环是在    console\_driver 中  
  
    for (i = 0; i < MAX\_CMDLINECONSOLES && console\_cmdline[i].name[0]; i++) {  
  
        如果这个要注册的  console 的名称和内核启动参数命令行  console 名称不同，查看下一个。  
        if (strcmp(console\_cmdline[i].name, console->name) != 0)  
            continue;  
  
        如果要注册的  console 的  index 和内核启动参数命令行的  console 的  index 不相同，查看下一个。  
        if (console->index >= 0 && console->index != console\_cmdline[i].index)  
            continue;  
  
        设置要注册的  console 的  index 值。  
        if (console->index < 0)  
            console->index = console\_cmdline[i].index;  
  
        如果这个  console 的  setup 成员函数不能够正确执行，则跳出循环。  
        if (console->setup && console->setup(console, console\_cmdline[i].options) != 0)  
            break;  
  
        设置要注册的  console 的标志。  
        console->flags |= CON\_ENABLED;  
        console->index = console\_cmdline[i].index;  
  
        这个   selected\_console 变量在   console\_setup(） 函数中设置。  
        if (i == selected\_console) {  
            console->flags |= CON\_CONSDEV;  
            preferred\_console = selected\_console;  
                     }  
        break;  
            }  
  
  
    如果要注册  console 的  flags 没有设置使能标志，则这个  console 就不执行注册。  
    通过上面的分析，可以看出，只有当  第一次调用     register\_console（） 函数，或者  
    这个  console 的名称和内核启动命令行参数相同，才会注册。  
    if (!(console->flags & CON\_ENABLED))  
        return;  
  
    如果已经注册了  boot 阶段的  console ，则取得这个  console 的指针。  
    在注册  first real console 时，这个    boot 阶段的  console 会被注销掉。  
    if (bootconsole) {  
        unregister\_console(bootconsole);  
        console->flags &= ~CON\_PRINTBUFFER;  
            }  
  
    取得 这个  console\_drivers 的信号量。  
    acquire\_console\_sem();  
  
    如果这个  console 的  flags 设置了  CON\_CONSDEV 标志，或者这个  console\_drivers 链表为空。  
    把这个   console 加入这个  console\_drivers 链表的最前面。  
    这个console\_drivers 链表的第一个节点，为当前选择的输出  console ，是当前控制台。  
    是通过   selected\_console 全局变量，来控制的。  
    否则加入这个  console\_drivers 链表的第二个节点。  
  
    从上面可以看出只有在第一次调用   register\_console（） 函数，或者  
    情况下个  console 的  flags 设置才能够设置   CON\_CONSDEV 标志。  
    if ((console->flags & CON\_CONSDEV) || console\_drivers == NULL) {  
        console->next = console\_drivers;  
        console\_drivers = console;  
        if (console->next)  
            console->next->flags &= ~CON\_CONSDEV;  
    } else {  
        console->next = console\_drivers->next;  
        console\_drivers->next = console;  
          }  
  
    if (console->flags & CON\_PRINTBUFFER) {  
  
        spin\_lock\_irqsave(&logbuf\_lock, flags);  
        con\_start = log\_start;  
        spin\_unlock\_irqrestore(&logbuf\_lock, flags);  
          }  
  
    释放  这个  console\_drivers 的信号量。  
    release\_console\_sem();  
}  
＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊  
在      register\_console（） 函数中，使用到了  console\_cmdline［］ 数组，  
这个数组在内核命令行的处理函数中设置。  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
在  printk.c  里面定义了     
\_\_setup("console=", console\_setup);  
当内核命令行的启动参数中含有 "console="  字符时，就调用  console\_setup（） 函数。  
这个   console\_setup（） 就记录下来  console  驱动的 name，以及一些选项参数  
到  console\_cmdline［］  数组中(如波特率)， 设置   preferred\_console  参数。  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
static int \_\_init console\_setup(char \*str)  
{  
    char name[sizeof(console\_cmdline[0].name)];  
    char \*s, \*options;  
    int idx;  
  
    取得  console 的名称，存入  name[] 数组中。  
    if (str[0] >= '0' && str[0] <= '9') {  
        strcpy(name, "ttyS");  
        strncpy(name + 4, str, sizeof(name) - 5);  
    } else {  
        strncpy(name, str, sizeof(name) - 1);  
           }  
    name[sizeof(name) - 1] = 0;  
  
    取得这个  console 的参数，比如串口的波特率等等。  
    if ((options = strchr(str, ',')) != NULL)  
        \*(options++) = 0;  
  
#ifdef \_\_sparc\_\_  
    if (!strcmp(str, "ttya"))  
        strcpy(name, "ttyS0");  
    if (!strcmp(str, "ttyb"))  
        strcpy(name, "ttyS1");  
#endif  
  
    取得  console 的号码，如  ttyS0，ttyS1 中的  0  或者 1。  
    for (s = name; \*s; s++)  
        if ((\*s >= '0' && \*s <= '9') || \*s == ',')  
            break;  
    idx = simple\_strtoul(s, NULL, 10);  
    \*s = 0;  
  
    add\_preferred\_console(name, idx, options);  
    return 1;  
}  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
int \_\_init add\_preferred\_console(char \*name, int idx, char \*options)  
{  
    struct console\_cmdline \*c;  
    int i;  
  
    循环  这个  console\_cmdline［］ 数组，查看是否这个  console 已经注册上。  
    for(i = 0; i < MAX\_CMDLINECONSOLES && console\_cmdline[i].name[0]; i++)  
        if (strcmp(console\_cmdline[i].name, name) == 0 &&  
              console\_cmdline[i].index == idx) {  
                selected\_console = i;  
                return 0;  
                      }  
  
    没有空闲的  slot 了。  
    if (i == MAX\_CMDLINECONSOLES)  
        return -E2BIG;  
  
    设置   selected\_console  变量。在  register\_console() 函数中使用到了。  
    selected\_console = i;  
  
    对  console\_cmdline[i] 成员中的各项进行设置。  
    c = &console\_cmdline[i];  
    memcpy(c->name, name, sizeof(c->name));  
    c->name[sizeof(c->name) - 1] = 0;  
    c->options = options;  
    c->index = idx;  
    return 0;  
}  
＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊  
问题：  
    1）这个  console\_drivers 链表的第一个节点有什么不同？  
    2）这个  con\_start、log\_start 变量的作用？  
    3）这个  selected\_console 和  preferred\_console 的作用？

**linux-mips启动分析（20）**

在    start\_kernel（）函数中调用   vfs\_caches\_init\_early()  函数，  
再由    vfs\_caches\_init\_early()  调用    dcache\_init\_early()    
和   inode\_init\_early（） 函数，完成虚拟文件系统的初始化。  
  
初始化   dentry  和   inode  缓冲队列的  hash 表。  
===============================================================  
  
void \_\_init vfs\_caches\_init\_early(void)  
{     
    对目录项哈希表进行早期的初始化。  
    dcache\_init\_early();  
    对文件节点哈希表进行早期初始化。  
    inode\_init\_early();  
}     
＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊  
这个   hashdist 变量在   mm/page\_alloc.c 文件中定义。参考下面分析。  
这个   dentry\_hashtable 变量是一个  hashtable 的头。  
在   fs/dcache.c 文件中定义。  
static struct hlist\_head \*dentry\_hashtable \_\_read\_mostly;  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
static \_\_initdata unsigned long dhash\_entries;  
static int \_\_init set\_dhash\_entries(char \*str)  
{  
    if (!str)  
        return 0;  
    dhash\_entries = simple\_strtoul(str, &str, 0);  
    return 1;  
}  
\_\_setup("dhash\_entries=", set\_dhash\_entries);  
  
这个  dhash\_entries 变量表示目录项哈希表入口数目，默认为  0 。  
可以通过这个内核启动参数的配置函数为      set\_dhash\_entries()  函数进行设置。  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
static void \_\_init dcache\_init\_early(void)  
{     
    int loop;  
    如果   hashes 是和   NUMA 节点来配置的，延迟   hash 的分配，  
    直到   vmalloc 空间有效后，再分配。  
    if (hashdist)  
        return;  
  
    返回的哈希表头地址为    dentry\_hashtable 。  
    哈希表入口项数目   numentries 值  2   的对数，保存到    d\_hash\_shift 变量。  
    这个   d\_hash\_mask 保存入口项的掩码。  
    dentry\_hashtable =  
        alloc\_large\_system\_hash("Dentry cache",  
                    sizeof(struct hlist\_head),  
                    dhash\_entries,  
                                                      13,  
                    HASH\_EARLY,  
                    &d\_hash\_shift,  
                    &d\_hash\_mask,  
                                                      0);  
  
    对哈希表的每个链表头进行初始化。  
    for (loop = 0; loop < (1 << d\_hash\_shift); loop++)  
        INIT\_HLIST\_HEAD(&dentry\_hashtable[loop]);  
}  
＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊  
  
这个   hashdist 变量在   mm/page\_alloc.c 文件中定义。参考下面分析。  
这个   dentry\_hashtable 变量是一个  hashtable 的头。  
在   fs/dcache.c 文件中定义。  
static struct hlist\_head \*dentry\_hashtable \_\_read\_mostly;  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
static \_\_initdata unsigned long ihash\_entries;  
static int \_\_init set\_ihash\_entries(char \*str)  
{  
    if (!str)  
        return 0;  
    ihash\_entries = simple\_strtoul(str, &str, 0);  
    return 1;  
}  
\_\_setup("ihash\_entries=", set\_ihash\_entries);  
  
这个 ihash\_entries 变量表示文件节点哈希表入口数目，默认为  0 。  
可以通过这个内核启动参数的配置函数为     set\_ihash\_entries()  函数进行设置。  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
  
void \_\_init inode\_init\_early(void)  
{  
    int loop;  
  
  
    这个   hashdist 变量在   mm/page\_alloc.c 文件中定义。  
    如果   hashes 是和   NUMA 节点来配置的，延迟   hash 的分配，  
    直到   vmalloc 空间有效后，再分配。  
    if (hashdist)  
        return;  
  
    返回的哈希表头地址为   inode\_hashtable 。  
    哈希表入口项数目   numentries 值  2   的对数，保存到    i\_hash\_shift 变量。  
    这个   i\_hash\_mask 保存入口项的掩码。  
    inode\_hashtable =  
        alloc\_large\_system\_hash("Inode-cache",  
                    sizeof(struct hlist\_head),  
                    ihash\_entries,  
                                                     14,  
                    HASH\_EARLY,  
                    &i\_hash\_shift,  
                    &i\_hash\_mask,  
                                                     0);  
    对哈希表的每个链表头进行初始化。  
    for (loop = 0; loop < (1 << i\_hash\_shift); loop++)  
        INIT\_HLIST\_HEAD(&inode\_hashtable[loop]);  
}  
＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊  
这个   hashdist 变量在   mm/page\_alloc.c 文件中定义。  
默认值为  HASHDIST\_DEFAULT，这个      hashdist 变量的值可以通过内核启动参数设置，  
这个内核启动参数的配置函数为     set\_hashdist()  函数。  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
#if defined(CONFIG\_NUMA) && (defined(CONFIG\_IA64) || defined(CONFIG\_X86\_64))  
#define HASHDIST\_DEFAULT 1  
#else  
#define HASHDIST\_DEFAULT 0  
#endif    
  
可以看出     HASHDIST\_DEFAULT 的值，在我们的系统中默认为  0  。  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
int hashdist = HASHDIST\_DEFAULT;  
  
#ifdef CONFIG\_NUMA   
static int \_\_init set\_hashdist(char \*str)  
{   static struct hlist\_head \*dentry\_hashtable \_\_read\_mostly;  
  
    if (!str)  
        return 0;  
    hashdist = simple\_strtoul(str, &str, 0);  
    return 1;  
}             
\_\_setup("hashdist=", set\_hashdist);  
#endif        
＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊  
参数含义：  
1）  tablename          哈希表名字  
2）   bucketsize        这个是每个元素的尺寸  
3）   numentries        元素的个数，可以取  0，由系统来确定，  
                        这时可能即使你给了值，系统也会把它变为最接近的  2  的幂  
4）   scale            取值有 13、14、15、17 之类的，如果   numentries 不为 0，这个参数没有作用。  
5）   lags                可取   HASH\_EARLY  或0，分配内存的地方根据这个有不同  
6）    \_hash\_shift    用于返回元素个数的以  2  为底的对数，也就是表示元素个数这个数值所用的比特数  
7）    \_hash\_mask        用于返回   \*\_hash\_shift  个比特所能表示的最大数   -1  
8）   limit            哈希表表元数上限，不是分配内存的总尺寸，不要弄混了。  
                        如果给个   0   值，那么系统使用   1/16    内存所能容纳的元素数作为哈希表表元数。  
  
参数    numentries  和  limit 有关系的，参数   numentries 的值不能超出  limit 的值  
（如果不为0，为  0  则不能超出系统计算的哈希表元数目），所以不给   numentries    也行。  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
void \*\_\_init alloc\_large\_system\_hash(const char \*tablename,  
                     unsigned long bucketsize,  
                     unsigned long numentries,  
                     int scale,  
                     int flags,  
                     unsigned int \*\_hash\_shift,  
                     unsigned int \*\_hash\_mask,  
                     unsigned long limit)  
{  
    unsigned long long max = limit;  
    unsigned long log2qty, size;  
    void \*table = NULL;  
  
    如果传进来的参数   numentries 为 0，则  
    if (!numentries) {  
  
        这个  nr\_kernel\_pages 变量在  free\_area\_init\_core（） 函数中赋值，  
        参考《linux-mips启动分析（4-3）》，表示不包含高端内存的系统内存共有的内存页面数。  
        计算不包含高端内存的系统内存共有   多少  MB（不足  1MB，以  1MB 计算）。  
        numentries = nr\_kernel\_pages;  
        numentries += (1UL << (20 - PAGE\_SHIFT)) - 1;  
        numentries >>= 20 - PAGE\_SHIFT;  
  
        取得  不足  1MB，以  1MB 计算后的页面数目。  
        numentries <<= 20 - PAGE\_SHIFT;  
  
        对  numentries 进行缩放。  
        if (scale > PAGE\_SHIFT)  
            numentries >>= (scale - PAGE\_SHIFT);  
        else  
            numentries <<= (PAGE\_SHIFT - scale);  
  
        不足一个页面的空间，以一个页面的空间为最小。  
        if (unlikely((numentries \* bucketsize) < PAGE\_SIZE))  
            numentries = PAGE\_SIZE / bucketsize;  
           }  
  
    向上取得   numentries  为  2  的幂次方数值。  
    numentries = roundup\_pow\_of\_two(numentries);  
  
    这个  max 等于  limit，如果等于 0 ，就计算   numentries 的最大值  max 。  
    if (max == 0) {  
  
        这个  nr\_all\_pages 变量表示所有内存的页面   nr\_all\_pages 变量。  
        即哈希表的内存最大为整个内存的   1/16  的空间。  
        max = ((unsigned long long)nr\_all\_pages << PAGE\_SHIFT) >> 4;  
        这个   do\_div() 宏，使  max 除以   bucketsize，并把结果赋值给   max 变量。  
        do\_div(max, bucketsize);  
          }  
  
    变量  numentries 不能够大于  max 变量，即哈希表的入口不能够大于  max 变量。  
    if (numentries > max)  
        numentries = max;  
  
    对   numentries  取  2 的对数。  
    log2qty = ilog2(numentries);  
  
    do {  
        取得要分配空间的大小。  
        size = bucketsize << log2qty;  
  
        可以看出，可以在三个地方，分配需要的空间。  
        当  flags 为  HASH\_EARLY 标志时，表示内核没有启动完毕，使用启动内存分配器来分配空间。  
        if (flags & HASH\_EARLY)  
            table = alloc\_bootmem(size);  
  
        如果  hashdist 不等于  0 ，从   vmalloc 空间分配内存。  
        else if (hashdist)  
            table = \_\_vmalloc(size, GFP\_ATOMIC, PAGE\_KERNEL);  
  
        下面的在系统启动完毕后，从伙伴内存管理器上分配页面。  
        else {  
            unsigned long order;  
            for (order = 0; ((1UL << order) << PAGE\_SHIFT) < size; order++) ;  
            table = (void\*) \_\_get\_free\_pages(GFP\_ATOMIC, order);  
                      }  
    } while (!table && size > PAGE\_SIZE && --log2qty);  
  
      
    如果  table 等于  NULL ，表示没有足够的空间来进行分配。  
    if (!table)  
        panic("Failed to allocate %s hash table\n", tablename);  
  
    返回 哈希表入口项数目   numentries 值  2   的对数。  
    if (\_hash\_shift)  
        \*\_hash\_shift = log2qty;  
    if (\_hash\_mask)  
        \*\_hash\_mask = (1 << log2qty) - 1;  
  
    return table;  
}

**linux-mips启动分析（21）**

在  start\_kernel() 函数中调用   mem\_init()  函数初始化内存页描述符。  
设置内存上下界和页表项初始值  
================================================================  
  
void \_\_init mem\_init(void)  
{  
    unsigned long codesize, reservedpages, datasize, initsize;  
    unsigned long tmp, ram;  
  
    这个   max\_mapnr 变量表示内存空间的结束地址页面祯号（PFN）。  
#ifdef CONFIG\_HIGHMEM  
    max\_mapnr = highend\_pfn;  
#else  
    max\_mapnr = max\_low\_pfn;  
#endif  
  
    这个  high\_memory 变量是一个全局变量，是一个虚拟地址。  
    表示大于这个地址的虚拟地址空间，为非逻辑地址空间，没有物理地址和这段空间地址一一对应。  
    high\_memory = (void \*) \_\_va(max\_low\_pfn << PAGE\_SHIFT);  
  
    这个  totalram\_pages 变量为全局变量。  
    这个   free\_all\_bootmem() 函数参考《linux-mips启动分析（4-5）》。  
    这个   free\_all\_bootmem() 函数返回低端空闲页面的总数。  
    totalram\_pages += free\_all\_bootmem();  
  
    建立几个空白页面。  
    totalram\_pages -= setup\_zero\_pages();     
  
    使用  ram 变量记录可用的物理  RAM 的页面数目。  
    使用  reservedpages 表示页面留给内核代码或者为内存  hole 的数目。  
    reservedpages = ram = 0;  
  
    变量  max\_low\_pfn 表示低端内存可用的最大的  PFN。  
    for (tmp = 0; tmp < max\_low\_pfn; tmp++)  
        检测页面是否是可用的  RAM，还是内存空洞。  
        if (page\_is\_ram(tmp)) {  
            ram++;  
            if (PageReserved(pfn\_to\_page(tmp)))  
                reservedpages++;  
        }  
    num\_physpages = ram;  
  
    如果系统中有高端内存，定义了    CONFIG\_HIGHMEM 宏。  
    使用   totalhigh\_pages  变量表示物理高端内存的页面数目。  
#ifdef CONFIG\_HIGHMEM  
  
    循环遍历高端内存的物理页面。  
    for (tmp = highstart\_pfn; tmp < highend\_pfn; tmp++) {  
        struct page \*page = mem\_map + tmp;  
      
        如果这个页面不是物理  RAM ，设置这个为  Reserved。  
        if (!page\_is\_ram(tmp)) {  
            SetPageReserved(page);  
            continue;  
                      }  
        如果这个页面是物理  RAM 。对这个页面进行设置。  
        ClearPageReserved(page);  
        设置引用计数为  1。  
        init\_page\_count(page);  
        释放页面，把高端内存的页面加入伙伴内存系统的空闲链表。  
        在    free\_all\_bootmem() 函数中只是对低端内存进行了处理。  
        \_\_free\_page(page);  
        totalhigh\_pages++;  
           }  
    totalram\_pages += totalhigh\_pages;  
    num\_physpages += totalhigh\_pages;  
#endif  
  
    这三个变量分别表示内核代码段，数据段，  init 段的大小。  
    codesize =  (unsigned long) &\_etext - (unsigned long) &\_text;  
    datasize =  (unsigned long) &\_edata - (unsigned long) &\_etext;  
    initsize =  (unsigned long) &\_\_init\_end - (unsigned long) &\_\_init\_begin;  
  
#ifdef CONFIG\_64BIT  
    if ((unsigned long) &\_text > (unsigned long) CKSEG0)  
        kclist\_add(&kcore\_kseg0, (void \*) CKSEG0, 0x80000000 - 4);  
#endif  
  
    这个  kcore\_mem 和  kcore\_vmalloc 变量定义在   arch/mips/mm/init.c 文件，  
    是一个    kcore\_list 链表的节点。  
    下面的两个函数把      kcore\_mem 和  kcore\_vmalloc 节点，添加到      kcore\_list 链表。  
    把新的节点添加到链表的最前面。  
    kclist\_add(&kcore\_mem, \_\_va(0), max\_low\_pfn << PAGE\_SHIFT);  
    kclist\_add(&kcore\_vmalloc, (void \*)VMALLOC\_START, VMALLOC\_END-VMALLOC\_START);  
}  
＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊  
unsigned long setup\_zero\_pages(void)  
{  
    unsigned int order;  
    unsigned long size;  
    struct page \*page;  
  
    如果  CPU 支持  cpu\_has\_vce 的，则分配内存块  order 为 3 ，即  8 个内存页面。  
    if (cpu\_has\_vce)  
        order = 3;  
    else  
        order = 0;  
  
    从伙伴内存系统分配内存页面，返回所分配内存的空间的第一个页的线性地址。  
    这个  empty\_zero\_page  变量是一个全局变量。  
    empty\_zero\_page = \_\_get\_free\_pages(GFP\_KERNEL | \_\_GFP\_ZERO, order);  
    if (!empty\_zero\_page)  
        panic("Oh boy, that early out of memory?");  
  
    把线性地址转换成   struct page 指针。  
    page = virt\_to\_page((void \*)empty\_zero\_page);  
  
    把所分配内存的页面    struct page 的引用计数都置 1 。  
    split\_page(page, order);  
  
    把所分配内存的页面    struct page 的  flags 设置  PG\_reserved 标志。  
    while (page < virt\_to\_page((void \*)(empty\_zero\_page + (PAGE\_SIZE << order)))) {  
        SetPageReserved(page);  
        page++;  
          }  
    size = PAGE\_SIZE << order;  
  
    这个  zero\_page\_mask  变量是一个全局变量。  
    zero\_page\_mask = (size - 1) & PAGE\_MASK;  
  
    返回分配的空白页面的数目。  
    return 1UL << order;  
}  
＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊  
struct kcore\_list {  
    struct kcore\_list \*next;  
    unsigned long addr;  
    size\_t size;  
};  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
这个  kcore\_mem 和  kcore\_vmalloc 变量定义在   arch/mips/mm/init.c 文件，  
是一个    kcore\_list 链表的节点。  
static struct kcore\_list kcore\_mem, kcore\_vmalloc;  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
static struct kcore\_list \*kclist;  
这个   kclist 变量表示      kcore\_list 链表的头节点指针。  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
如果没有定义   CONFIG\_PROC\_KCORE 这个宏，则这个   kclist\_add() 函数为空函数。  
  
void kclist\_add(struct kcore\_list \*new, void \*addr, size\_t size)  
{     
    new->addr = (unsigned long)addr;  
    new->size = size;  
  
    把新的节点添加到链表的最前面。  
    write\_lock(&kclist\_lock);  
    new->next = kclist;  
    kclist = new;   
    write\_unlock(&kclist\_lock);  
}   
＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊  
问题：  
    1）这个  total\_pages 变量在什么地方定义？  
    2)  这个宏   cpu\_has\_vce 的意义？  
    3）这个       kcore\_list 链表的意义？

**linux-mips启动分析（21-2）**

在  start\_kernel() 函数中调用   mem\_init()  函数，  
而  mem\_init() 函数调用  setup\_zero\_pages()  函数，  
而    setup\_zero\_pages()  函数调用   \_\_get\_free\_pages() 函数来分配页面。  
  
本章讲述从伙伴系统分配页面的具体过程。  
=============================================================  
  
fastcall unsigned long \_\_get\_free\_pages(gfp\_t gfp\_mask, unsigned int order)  
{     
    struct page \* page;  
  
    用这个函数分配  2^order 个连续的页面。返回分配空间的第一个页面的描述符地址。  
    page = alloc\_pages(gfp\_mask, order);  
    if (!page)  
        return 0;  
  
    返回  page 所对应页的线性地址。  
    return (unsigned long) page\_address(page);  
}  
＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊  
如果定义了  WANT\_PAGE\_VIRTUAL 这个宏，  
这个宏决定是否在  struct page 中定义  virtual 成员，这个成员等于这个页面的线性地址。  
如果是高端内存，在分配内存后，把    virtual 成员赋值为页面的虚拟地址。  
所以可以直接返回页面的虚拟地址。  
#define page\_address(page) ((page)->virtual)  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
如果定义了  HASHED\_PAGE\_VIRTUAL 这个宏，这个宏依赖于   CONFIG\_HIGHMEM 宏，  
表明系统有高端内存，并且系统没有定义   WANT\_PAGE\_VIRTUAL 这个宏。  
void \*page\_address(struct page \*page)  
{  
    unsigned long flags;  
    void \*ret;  
    struct page\_address\_slot \*pas;  
  
    如果这个页面不是高端内存，可以直接通过物理地址直接转换成虚拟地址。  
    if (!PageHighMem(page))  
        return lowmem\_page\_address(page);  
  
    如果这个页面是高端内存，则需要  
  
    返回的是  page\_address\_htable［］ 数组的一个元素指针。  
    pas = page\_slot(page);  
    ret = NULL;  
  
    锁定返回的page\_address\_htable［］ 数组的元素。  
    spin\_lock\_irqsave(&pas->lock, flags);  
    if (!list\_empty(&pas->lh)) {  
        struct page\_address\_map \*pam;  
  
        遍历这个链表查找  page 的虚拟地址。  
        list\_for\_each\_entry(pam, &pas->lh, list) {  
  
            如果这个链表阶段的   page 地址等于我们要查找的   page 的地址。  
            对  ret 赋值为这个  page 的虚拟地址。  
            if (pam->page == page) {  
                ret = pam->virtual;  
                goto done;  
                                }  
                      }  
            }  
  
    如果这个 page 没有在链表上，则返回  NULL。  
done:  
    释放返回的page\_address\_htable［］ 数组的元素。  
    spin\_unlock\_irqrestore(&pas->lock, flags);  
    return ret;  
}  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
表明系统中没有高端内存，并且系统没有定义   WANT\_PAGE\_VIRTUAL 这个宏。  
#define page\_address(page) lowmem\_page\_address(page)  
  
static \_\_always\_inline void \*lowmem\_page\_address(struct page \*page)  
{  
    return \_\_va(page\_to\_pfn(page) << PAGE\_SHIFT);  
}  
  
由于没有高端内存，可以直接通过物理地址直接转换成虚拟地址。  
＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊  
定义了一个哈希表，大小为   （1<<PA\_HASH\_ORDER）。  
这个哈希表的散列函数为   hash\_ptr() 函数。  
这个散列函数是   page 地址乘以   0x9E370001，这个值是接近  2^32 的黄金比例的一个素数。  
在使它小于  哈希表的大小。  
如果发生冲突，就在这个链表中添加节点。或在链表中查找节点。  
  
所以   page\_slot() 函数返回的是  page\_address\_htable［］ 数组的一个元素指针。  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
static struct page\_address\_slot {  
    struct list\_head lh;           
    spinlock\_t lock;             
} \_\_\_\_cacheline\_aligned\_in\_smp page\_address\_htable[1<<PA\_HASH\_ORDER];  
  
static struct page\_address\_slot \*page\_slot(struct page \*page)  
{  
    return &page\_address\_htable[hash\_ptr(page, PA\_HASH\_ORDER)];  
}  
＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊  
如果  gfp\_mask 没有设置   GFP\_\_WAIT 标志，分配器将不会阻塞，而是在内存紧张时，返回  NULL。  
  
#define alloc\_pages(gfp\_mask, order) \  
        alloc\_pages\_node(numa\_node\_id(), gfp\_mask, order)  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
static inline struct page \*alloc\_pages\_node(int nid, gfp\_t gfp\_mask,  
                                                                unsigned int order)  
{  
    if (unlikely(order >= MAX\_ORDER))  
        return NULL;  
  
    使用  numa\_node\_id() 宏函数，取得要分配内存的内存的节点号。  
    if (nid < 0)  
        nid = numa\_node\_id();  
  
    这个  node\_zonelists 成员是一个  zone\_t 的链表，每个管理区分配对应了一个  zonelist 。  
    参考《linux-mips启动分析（7）》。  
    return \_\_alloc\_pages(gfp\_mask, order,  
        NODE\_DATA(nid)->node\_zonelists + gfp\_zone(gfp\_mask));  
}  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
#define numa\_node\_id()         (cpu\_to\_node(raw\_smp\_processor\_id()))  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
#define raw\_smp\_processor\_id() (current\_thread\_info()->cpu)  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
#define cpu\_to\_node(cpu)       (0)  
＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊  
  
这个  \_\_alloc\_pages() 函数负责遍历   zonelist （回退管理区链表），如果选择一个合适  
的管理区进行内存分配。  
如果内存紧张，就唤醒   kswapd 内核线程，释放内存页面。  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
struct page \* fastcall  
\_\_alloc\_pages(gfp\_t gfp\_mask, unsigned int order, struct zonelist \*zonelist)  
{  
  
    这个   \_\_GFP\_WAIT 表明请求页面分配的进程是否可以睡眠等待。  
    const gfp\_t wait = gfp\_mask & \_\_GFP\_WAIT;  
    struct zone \*\*z;  
    struct page \*page;  
    struct reclaim\_state reclaim\_state;  
  
    请求页面分配进程的进程描述符指针。  
    struct task\_struct \*p = current;  
    int do\_retry;  
    int alloc\_flags;   
    int did\_some\_progress;  
  
    如果设置了  \_\_GFP\_WAIT 标志，就执行内核抢占。  
    might\_sleep\_if(wait);  
  
    如果没有定义   CONFIG\_FAIL\_PAGE\_ALLOC 这个宏，这个   should\_fail\_alloc\_page() 函数就直接返回 0。  
    先跳过这里。  
    if (should\_fail\_alloc\_page(gfp\_mask, order))  
        return NULL;  
  
    如果在当前内存管理区，没有成功分配内存，就从   zonelist （回退管理区链表）取下一个内存管理区，进行分配。  
restart:  
    z = zonelist->zones;  /\* the list of zones suitable for gfp\_mask \*/  
  
    当  z 为 NULL 时，表示   zonelist （回退管理区链表）已经结束。  
    if (unlikely(\*z == NULL)) {  
        return NULL;  
          }  
  
    page = get\_page\_from\_freelist(gfp\_mask|\_\_GFP\_HARDWALL, order,  
                zonelist, ALLOC\_WMARK\_LOW|ALLOC\_CPUSET);  
    if (page)  
        goto got\_pg;  
  
    if (NUMA\_BUILD && (gfp\_mask & GFP\_THISNODE) == GFP\_THISNODE)  
        goto nopage;  
  
    for (z = zonelist->zones; \*z; z++)  
        wakeup\_kswapd(\*z, order);  
  
    alloc\_flags = ALLOC\_WMARK\_MIN;  
    if ((unlikely(rt\_task(p)) && !in\_interrupt()) || !wait)  
        alloc\_flags |= ALLOC\_HARDER;  
    if (gfp\_mask & \_\_GFP\_HIGH)  
        alloc\_flags |= ALLOC\_HIGH;  
    if (wait)  
        alloc\_flags |= ALLOC\_CPUSET;  
  
    page = get\_page\_from\_freelist(gfp\_mask, order, zonelist, alloc\_flags);  
    if (page)  
        goto got\_pg;  
  
rebalance:  
    if (((p->flags & PF\_MEMALLOC) || unlikely(test\_thread\_flag(TIF\_MEMDIE)))  
            && !in\_interrupt()) {  
        if (!(gfp\_mask & \_\_GFP\_NOMEMALLOC)) {  
nofail\_alloc:  
            page = get\_page\_from\_freelist(gfp\_mask, order,  
                zonelist, ALLOC\_NO\_WATERMARKS);  
            if (page)  
                goto got\_pg;  
            if (gfp\_mask & \_\_GFP\_NOFAIL) {  
                congestion\_wait(WRITE, HZ/50);  
                goto nofail\_alloc;  
                                }  
                      }  
        goto nopage;  
           }  
  
    if (!wait)  
        goto nopage;  
  
    cond\_resched();  
  
    cpuset\_memory\_pressure\_bump();  
    p->flags |= PF\_MEMALLOC;  
    reclaim\_state.reclaimed\_slab = 0;  
    p->reclaim\_state = &reclaim\_state;  
  
    did\_some\_progress = try\_to\_free\_pages(zonelist->zones, gfp\_mask);  
  
    p->reclaim\_state = NULL;  
    p->flags &= ~PF\_MEMALLOC;  
  
    cond\_resched();  
  
    if (likely(did\_some\_progress)) {  
        page = get\_page\_from\_freelist(gfp\_mask, order,  
                        zonelist, alloc\_flags);  
        if (page)  
            goto got\_pg;  
    } else if ((gfp\_mask & \_\_GFP\_FS) && !(gfp\_mask & \_\_GFP\_NORETRY)) {  
        page = get\_page\_from\_freelist(gfp\_mask|\_\_GFP\_HARDWALL, order,  
                zonelist, ALLOC\_WMARK\_HIGH|ALLOC\_CPUSET);  
        if (page)  
            goto got\_pg;  
  
        out\_of\_memory(zonelist, gfp\_mask, order);  
        goto restart;  
           }  
  
    do\_retry = 0;  
    if (!(gfp\_mask & \_\_GFP\_NORETRY)) {  
        if ((order <= 3) || (gfp\_mask & \_\_GFP\_REPEAT))  
            do\_retry = 1;  
        if (gfp\_mask & \_\_GFP\_NOFAIL)  
            do\_retry = 1;  
           }  
    if (do\_retry) {  
        congestion\_wait(WRITE, HZ/50);  
        goto rebalance;  
           }  
  
nopage:  
    if (!(gfp\_mask & \_\_GFP\_NOWARN) && printk\_ratelimit()) {  
        printk(KERN\_WARNING "%s: page allocation failure."  
            " order:%d, mode:0x%x\n",  
            p->comm, order, gfp\_mask);  
        dump\_stack();  
        show\_mem();  
           }  
got\_pg:  
    return page;  
}  
＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊  
static struct page \*  
get\_page\_from\_freelist(gfp\_t gfp\_mask, unsigned int order,  
                                            struct zonelist \*zonelist, int alloc\_flags)  
{  
    struct zone \*\*z;  
    struct page \*page = NULL;  
  
    这个   zone\_idex() 对于  ZONE\_DMA 返回  0， 对于 ZONE\_NORMAL 返回 1。  
    int classzone\_idx = zone\_idx(zonelist->zones[0]);  
    struct zone \*zone;  
    nodemask\_t \*allowednodes = NULL;  
    int zlc\_active = 0;       
    int did\_zlc\_setup = 0;        
  
    遍历  zonelist，查找有足够可分配空间的   zone 。  
zonelist\_scan:  
    z = zonelist->zones;  
  
    do {  
        在我们的系统中没有配置   CONFIG\_NUMA，为一致内存管理模型（UMA），所以我们不对 NUMA 系统进行分析。  
        下面的条件永远返回  FALSE。  
        if (NUMA\_BUILD && zlc\_active &&  !zlc\_zone\_worth\_trying(zonelist, z, allowednodes))  
                continue;  
  
        zone = \*z;  
  
        下面的条件永远返回  FALSE。  
        if (unlikely(NUMA\_BUILD && (gfp\_mask & \_\_GFP\_THISNODE) &&  
                                        zone->zone\_pgdat != zonelist->zones[0]->zone\_pgdat))  
                break;  
  
        如果  alloc\_flags 设置了  ALLOC\_CPUSET 标志。  
        这个  cpuset\_zone\_allowed\_softwall() 函数根据系统是否定义  CONFIG\_CPUSETS 这个宏有关系。  
        if ((alloc\_flags & ALLOC\_CPUSET) &&  
            !cpuset\_zone\_allowed\_softwall(zone, gfp\_mask))  
                goto try\_next\_zone;  
  
        如果  alloc\_flags 没有设置  ALLOC\_NO\_WATERMARKS 标志。  
        if (!(alloc\_flags & ALLOC\_NO\_WATERMARKS)) {  
            unsigned long mark;  
          
            根据  flags 来设置  mark 的值。  
            if (alloc\_flags & ALLOC\_WMARK\_MIN)  
                mark = zone->pages\_min;  
            else if (alloc\_flags & ALLOC\_WMARK\_LOW)  
                mark = zone->pages\_low;  
            else  
                mark = zone->pages\_high;  
  
            如果这个内存管理区的   free page 的数目，大于  mark，这个函数返回    1。  
            if (!zone\_watermark\_ok(zone, order, mark,  
                    classzone\_idx, alloc\_flags)) {  
                if (!zone\_reclaim\_mode ||  
                    !zone\_reclaim(zone, gfp\_mask, order))  
                    goto this\_zone\_full;  
                                 }  
                    }  
  
        page = buffered\_rmqueue(zonelist, zone, order, gfp\_mask);  
        if (page)  
            break;  
this\_zone\_full:  
        if (NUMA\_BUILD)  
            zlc\_mark\_zone\_full(zonelist, z);  
try\_next\_zone:  
        if (NUMA\_BUILD && !did\_zlc\_setup) {  
            /\* we do zlc\_setup after the first zone is tried \*/  
            allowednodes = zlc\_setup(zonelist, alloc\_flags);  
            zlc\_active = 1;  
            did\_zlc\_setup = 1;  
                     }  
    } while (\*(++z) != NULL);  
  
    if (unlikely(NUMA\_BUILD && page == NULL && zlc\_active)) {  
  
        zlc\_active = 0;  
        goto zonelist\_scan;  
    }  
    return page;  
}

＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊  
                                           蛐蛐  
                    http://qgjie456.blog.163.com/  
                        MSN：qgjie@hotmail.com  
                                本文适用于  
                            linux-2.6.22.8  
                                 V 0.1  
                        欢迎转载，但请保留作者信息  
＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊