第 6 章 内核活动

本章将介绍内核中一些稍小一点的基础设施（机制），它们不特定于哪一个进程，而是为所有进程或内核提供服务。这些基础设施虽然不像内存管理、进程管理、文件系统、设备驱动程序等这么大型，但是对内核的运行也是至关重要的。

本章主要介绍的内容有通用的 MIPS 异常处理机制、TLB 异常处理、系统调用实现机制、中断处理、软中断、工作队列、时间管理、内核锁机制等。

6.1 MIPS 异常

MIPS 体系结构异常包括处理器自身运行过程中产生的异常和外部中断。本节主要介绍 MIPS 体系结构异常的类型、各异常处理程序的入口地址、以及 Linux 内核填充异常向量的机制，后面将介绍 TLB 异常、系统调用和中断的处理程序的实现。

6.1.1 异常类型

MIPS 异常是指打断程序正常执行的事件，MIPS 异常包括下列类型：

（1）复位（Reset）、软复位（Soft Reset）、不可屏蔽中断（NMI）；

（2）普通异常：用户态 TLB 重填、核心态 TLB 重填、TLB 无效、TLB 修改、**中断**、执行障碍、读障碍、写障碍、缓存错误、系统调用等；

（3）调试（EJTAG Debug）异常，本书暂不介绍。

●复位：系统冷启动时进入，也就是 CPU 启动时进入的状态，各寄存器状态复位；

●软复位：当 CPU 接收到复位信号时进入软件复位，软复位是复位的一个子集；

●不可屏蔽中断：当 NMI 信号发送到 CPU 时进入不可屏蔽中断，NMI 是不可被软件屏蔽的异常。

以上三个异常的处理程序具有相同的入口地址，异常处理程序通过检测状态寄存器（Status）来区分是哪种异常。

●普通异常：当 CPU 发生普通异常时，原因寄存器（Cause）ExcCode 字段（Cause[6..2]）记录了产生异常的代码，ExcCode 字段最多可表示 32 种类型的异常，如下表所示：

序号	Exc Code	异常类型	描　　述
0	0x00	**Int**	中断（外部中断、软件中断）。
1	0x01	**Mod**	写操作时匹配的 TLB 项 D 位为 0，不允许写操作。
2	0x02	**TLBL**	TLB 重填异常（没有匹配 TLB 项）或 TLB 无效异常（TLB 匹配，但 V=0），加载或取指。
3	0x03	**TLBS**	TLB 重填异常（没有匹配 TLB 项）或 TLB 无效异常（TLB 匹配，但 V=0），存储。
4	0x04	AdEL	地址错误（未对齐，越界等），加载或取指操作。
5	0x05	AdES	地址错误，存储操作。
6	0x06	IBE	总线错误（取指）。
7	0x07	DBE	总线错误（存取数据）。
8	0x08	**Sys**	系统调用（执行 syscall 指令）。
9	0x09	Bp	断点异常。

10	0x0a	RI	执行了保留的指令码。
11	0x0b	CpU	协处理器访问错误（用户态没有访问权限）。
12	0x0c	Ov	运算溢出异常。
13	0x0d	Tr	陷阱异常（执行陷阱指令结果为真时进入）。
14	0x0e	MSAFPE	MSA 浮点异常。
15	0x0f	FPE	浮点异常，由浮点协处理器触发。
16-17	0x10-0x11		处理器可自定义的异常
18	0x12	C2E	协处器 2 触发异常。
19	0x13	**TLBRI**	TLB 读阻碍异常（TLB 匹配，但 RI=1）。
20	0x14	**TLBXI**	TLB 执行阻碍异常（TLB 匹配，但 XI=1）。
21	0x15	MSADis	MSA Disabled exception
22	0x16	MDMX	Previously MDMX Unusable Exception (MDMX ASE). MDMX deprecated with Revision 5.
23	0x17	WATCH	Reference to WatchHi/WatchLo address
24	0x18	MCheck	Machine check（TLB 缓存中多个匹配项）
25	0x19	Thread	Thread Allocation, Deallocation, or Scheduling Exceptions (MIPS® MT Module)
26	0x1a	DSPDis	DSP Module State Disabled exception
27	0x1b	GE	Virtualized Guest Exception
28-29	0x1c - 0x1d		Reserved
30	0x1e	CacheErr	缓存错误
31	0x1f		保留

注：异常更详细的信息请读者参考 MIPS 体系结构手册。

6.1.2 异常向量

MIPS 体系结构规定产生异常时，CPU 所做的主要工作如下（假设发生异常前 Status.EXL=0）：

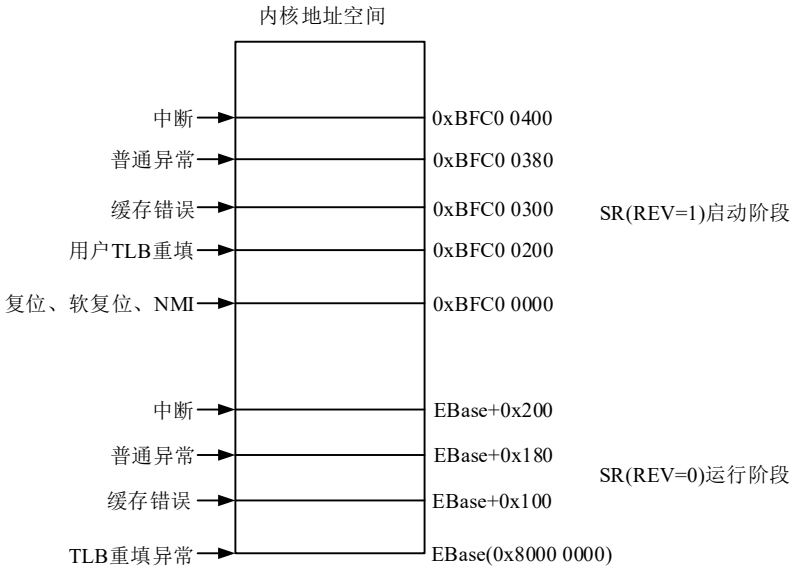
1、设置 EPC 寄存器，保存异常处理程序的返回地址，通常是引发异常的指令或下一条指令的地址。

2、设置 Status.EXL 位，CPU 进入内核态同时禁止中断（EXL 置位会禁止中断）。

3、设置 Cause 寄存器，使软件能探测到产生异常的原因，从而使 PC 指针跳转到不同的异常处理程序入口地址。如果是地址异常，则设置 BadVaddr 寄存器，存储管理系统（MMU）异常还要设置一些 MMU 相关的寄存器。

4、CPU 程序指针（PC）跳转到相应异常的处理程序入口处（异常向量）取指执行。

异常向量位于内核地址空间的直接映射区，直接映射到物理内存（低 512M 空间）。下图示意了各类型异常处理程序的入口地址（异常向量）：



由于启动阶段异常向量位于引导加载程序内存区，由引导加载程序实现，所以这里不做介绍（包含复位、软复位、NMI 异常的处理）。本书只关心内核运行阶段位于内核镜像区的异常向量，异常向量基地址为 0x8000 0000 或 EBase 寄存器保存的地址。

用户态 TLB 重填异常（没有匹配的 TLB 项）处理程序的偏移地址为 0x0，因为它是发生概率很高的异常，所以需要单独处理。缓存错误异常处理程序偏移地址为 0x100，处理器通过 kseg1 段地址访问（实际地址为 0xA000_0000+0x100），其它异常的处理程序入口地址偏移量为 0x180。如果中断异常采用了向量模式或外部中断控制器模式，则中断处理程序入口地址偏移量为 0x200。**兼容模式**的中断处理程序入口地址偏移量仍然为 0x180，与其它异常相同。

下表简列了异常向量（处理程序）对应的异常类型：

异常向量地址	处理的异常类型
EBase（0x8000 0000)	用户态 TLB 重填异常（TLBL、TLBS、SR（EXL）=0）
EBase+0x100	缓存错误
EBase+0x180	核心态 TLB 重填异常（TLBL、TLBS、SR（EXL）=1） 中断（兼容模式） 其它异常
EBase+0x200	向量模式或外部中断控制器模式中断

CPU 从异常返回的指令是 eret，执行此指令后 Status.EXL 位清零，PC 从 EPC 寄存器保存的返回地址处取指，继续执行。

1 初始化异常向量

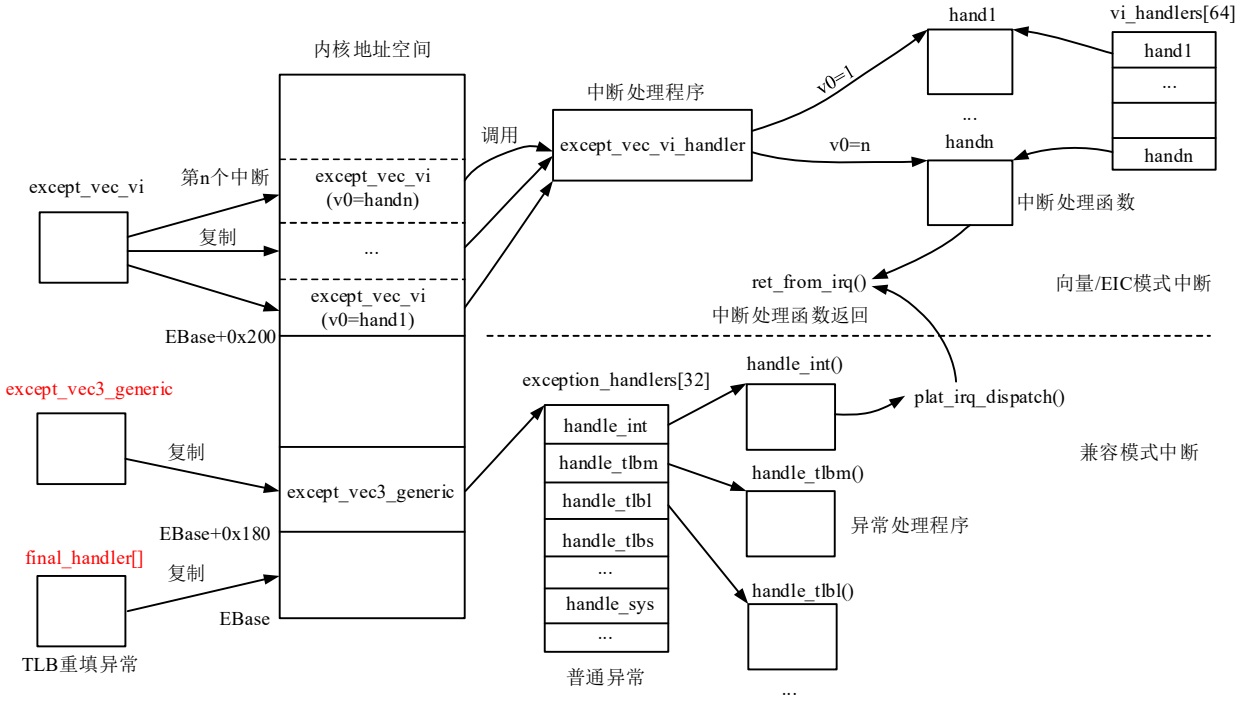
Linux 内核中 MIPS 异常处理程序如下图所示，TLB 重填异常处理程序保存在 final_handler[]数组，普通异常处理程序为 except_vec3_generic()函数。内核在初始化异常向量时会将处理程序代码复制到异常向量。

对于普通异常，内核定义了 exception_handlers[32]数组，数组项保存了 32 种普通异常处理程序的入口地址，except_vec3_generic()函数中根据原因寄存器 Cause 中保存的异常编码跳转到 exception_handlers[32]对应数组项保存的异常处理程序地址，对异常进行处理。

MIPS 中断处理分为兼容模式和向量模式（含外部中断控制器模式），兼容模式时中断由普通异常处

理程序中的 handle_int()函数处理，函数内调用 plat_irq_dispatch()函数，处理硬件中断。

对于向量模式中断处理程序，内核定义了 vi_handlers[64]数组，数组项中保存了各中断处理程序的入口地址（支持 64 个中断）。设置中断向量的 set_vi_handler()函数会将 except_vec_vi()函数代码复制到对应的中断向量中，并将中断对应的 vi_handlers[]数组项值（入口地址）传递给此函数。except_vec_vi()函数内以中断编号为参数调用 except_vec_vi_handler()函数，此函数内又将调用 vi_handlers[]数组中保存的中断处理函数（函数指针）。



内核在/arch/mips/kernel/traps.c 文件内定义了以下全局变量：

unsigned long ebase; /*异常向量基地址*/

unsigned long exception_handlers[32]; /*保存普通异常处理程序入口地址*/

unsigned long vi_handlers[64]; /*保存向量中断处理程序入口地址，最多 64 个中断向量*/

内核在/arch/mips/kernel/traps.c 文件内定义以下复制代码/设置普通异常处理程序入口地址的函数：

●**set_handler**(unsigned long offset, void *addr, unsigned long size)：将地址 addr 处 size 大小的代码复制到异常向量 offset 偏移量处。

●**set_except_vector**(int **n**, void *addr)：将 addr 表示的函数入口地址（函数指针）保存到普通异常处理程序入口地址 exception_handlers[**n**]数组项。

下面来看一下异常向量初始化函数 trap_init()（由 start_kernel()函数调用）的实现，它是一个体系结构相关的函数，函数代码简列如下（/arch/mips/kernel/traps.c）：

```
void __init trap_init(void)
{
    extern char except_vec3_generic;    /*普通异常处理程序入口地址，/arch/mips/kernel/genex.S*/
    extern char except_vec4;
    extern char except_vec3_r4000;
    unsigned long i;
```

```
check_wait();

if (cpu_has_veic || cpu_has_vint) {
        /*若是向量或外部中断控制器模式中断，则改变异常向量基地址*/
    unsigned long size = 0x200 + VECTORSPACING*64;
    ebase = (unsigned long)__alloc_bootmem(size, 1 << fls(size), 0);   /*异常向量分配空间*/
} else {    /*兼容模式中断*/
#ifdef CONFIG_KVM_GUEST
  ...
#else
    ebase = CKSEG0;            /*兼容模式中断，异常向量基地址为 0x8000_0000*/
#endif
    if (cpu_has_mips_r2_r6)
        ebase += (read_c0_ebase() & 0x3ffff000);
}
...
per_cpu_trap_init(true);   /*填充启动 CPU 的 TLB 异常处理程序，见下节*/

set_handler(0x180, &except_vec3_generic, 0x80);    /*复制普通异常处理程序*/
                /*复制 except_vec3_generic 处 0x80 大小的代码至普通异常向量*/

for (i = 0; i <= 31; i++)
    set_except_vector(i, handle_reserved);      /*初始化 32 个普通异常处理程序*/

if (cpu_has_ejtag && board_ejtag_handler_setup)
    board_ejtag_handler_setup();

if (cpu_has_watch)
    set_except_vector(23, handle_watch);
                        /*设置 23 号异常处理程序入口地址至 exception_handlers[23]*/

if (cpu_has_veic || cpu_has_vint) {        /*初始化中断向量，见本章下文*/
    int nvec = cpu_has_veic ? 64 : 8;
    for (i = 0; i < nvec; i++)
        set_vi_handler(i, NULL);    /*填充中断向量*/
}
else if (cpu_has_divec)
    set_handler(0x200, &except_vec4, 0x8);

parity_protection_init();   /*对于龙芯 1B 主要执行 write_c0_ecc(0x80000000)，traps.c*/

if (board_be_init)    /*处理数据总线和指令总线错误异常*/
```

```
        board_be_init();        /*板级实现*/

set_except_vector(0, using_rollback_handler() ? rollback_handle_int: handle_int);
                                        /*设置兼容模式中断处理程序为 handle_int()*/
set_except_vector(1, handle_tlbm);   /*TLB 修改异常处理程序为 handle_tlbm()*/
set_except_vector(2, handle_tlbl);     /*加载或取指 TLB 异常*/
set_except_vector(3, handle_tlbs);     /*存储 TLB 异常*/

set_except_vector(4, handle_adel);   /*地址错误，加载*/
set_except_vector(5, handle_ades);   /*地址错误，存储*/

set_except_vector(6, handle_ibe);
set_except_vector(7, handle_dbe);

set_except_vector(8, handle_sys);     /*系统调用异常处理程序为 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);
set_except_vector(14, handle_msa_fpe);

if (current_cpu_type() == CPU_R6000 || current_cpu_type() == CPU_R6000A) {
        ...
}


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(16, handle_ftlb);

if (cpu_has_rixiex) {
        set_except_vector(19, tlb_do_page_fault_0);       /*读障碍异常处理程序*/
        set_except_vector(20, tlb_do_page_fault_0);       /*写障碍异常处理程序*/
}

set_except_vector(21, handle_msa);
```

```
        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);

        if (board_cache_error_setup)        /*缓存错误*/
            board_cache_error_setup();

        if (cpu_has_vce)
            set_handler(0x180, &except_vec3_r4000, 0x100);
        else if (cpu_has_4kex)
            set_handler(0x180, &except_vec3_generic, 0x80);
        else
            set_handler(0x080, &except_vec3_generic, 0x80);

        local_flush_icache_range(ebase, ebase + 0x400);        /*刷新本地指令缓存*/

        sort_extable(__start___dbe_table, __stop___dbe_table);

        cu2_notifier(default_cu2_call, 0x80000000);/* Run last    */
    }
```

trap_init()函数主要工作是填充异常向量，设置各普通异常处理程序入口地址至 exception_handlers[]数组，填充中断向量，初始化 vi_handlers[]数组等，后面小节中将详细介绍各异常处理程序的实现。

2 普通异常处理程序

普通异常处理程序的入口地址为 EBase+0x180，在 trap_init()函数中将 except_vec3_generic 处的代码复制到此异常向量处。**except_vec3_generic()**函数定义在/arch/mips/kernel/genex.S 文件内：

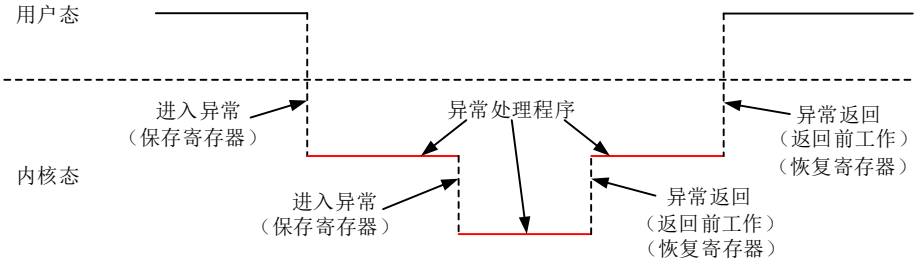
```
    NESTED(except_vec3_generic, 0, sp)
    .set    push
    .se t   noat
#if R5432_CP0_INTERRUPT_WAR
    ...
#endif
    mfc0    k1, CP0_CAUSE     /*读取原因寄存器*/
    andi    k1, k1, 0x7c      /*获取异常类型编码*/
#ifdef CONFIG_64BIT
```

```
    dsll      k1, k1, 1
#endif
    PTR_L k0, exception_handlers(k1)    /*跳转至对应异常处理程序处执行*/
    jr   k0
    .set    pop
    END(except_vec3_generic)
```

except_vec3_generic()函数比较简单，主要是充当分配器的功能。函数内首先读取协处理器 Cause 寄存器，获取当前异常的类型编码，由编码跳转至 exception_handlers[]对应数组项中保存的处理程序入口地址执行。

6.1.3 进出异常

MIPS 异常会打断程序的正常执行，使 CPU 进入异常处理程序。在异常处理程序中，首先要保存 CPU 当前寄存器信息（程序上下文信息）至内存，然后调用异常处理程序，最后异常处理程序返回前需要恢复 CPU 寄存器信息，以便继续执行异常发生时的程序。异常可能发生在 CPU 用户态也可能发生在内核态，用户态发生异常时，CPU 将进入内核态，如下图所示。



由于 Linux 内核代码只能在 CPU 内核态下运行，用户进程在 CPU 用户态下运行。因此 CPU 在运行用户进程时，只能通过异常使 CPU 进入内核态，执行内核代码。内核利用异常处理程序在 CPU 内核态运行的时机，将一些内核工作插入到异常处理程序返回前进行（在恢复寄存器前），例如，处理挂起信号、进程周期性调度、各种统计量的更新等等。内核态发生的异常，其处理程序返回前的工作与用户态发生的异常有所不同，后面将详细介绍。当然，也并不是所有的异常处理程序在返回前都会执行内核插入的工作，例如，用户态 TLB 重填异常要求快速地被处理，返回前就不会执行返回前工作。

本小节将介绍异常处理程序中保存寄存器、恢复寄存器是如何实现的，以及异常返回前需要执行的工作。

1 保存寄存器

CPU 进入异常处理程序后，第一件事就是把当前 CPU 寄存器信息保存至进程栈。所有异常处理程序都是在 CPU 内核态下运行，异常处理程序处于内核地址空间，因此寄存器信息需要保存到进程内核栈。

在第 5 章介绍的进程切换中调用的 **resume()**函数，会将下一个运行进程的初始内核栈顶地址写入全局数组 kernelsp[]，每个 CPU 核对应一个数组项。因此异常处理程序依据当前运行的 CPU 核，查找 kernelsp[]数组就能确定当前内核栈位置。

内核在/arch/mips/include/asm/stackframe.h 头文件内定义了保存 CPU 寄存器信息的宏，例如：

```
    .macro    SAVE_ALL      /*保存所有寄存器*/
    SAVE_SOME
    SAVE_AT
    SAVE_TEMP
```

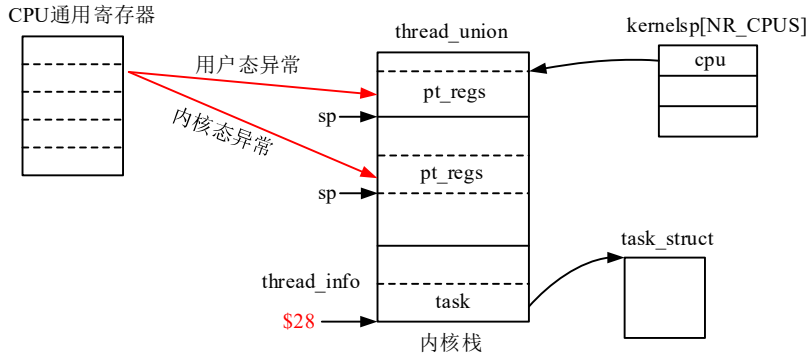
```
        SAVE_STATIC
        .endm
```

下面以 SAVE_SOME 宏为例，介绍其代码，如下所示：

```
        .macro    SAVE_SOME     （k0、k1 寄存器专用于异常处理程序，普通函数不使用）
        .set  push
        .set  noat
        .set  reorder
        mfc0   k0, CP0_STATUS      /*读取 CPU 状态寄存器至 k0*/
        sll    k0, 3            /*k0 左移 3 位，CU0 位移至最高位*/
        .set  noreorder
        bltz  k0, 8f         /*k0<0 则跳转到 8f 处（即 CU0=1，在内核态发生的异常）*/
        move k1, sp         /*保存异常前栈地址至 k1*/
        ...
        .set  reorder
        /*异常发生在用户态*/
        get_saved_sp       /*从 kernelsp[NR_CPUS]数组项中获取当前进程内核栈地址，保存至 k1*/
                           /*/arch/mips/include/asm/stackframe.h*/
#ifndef CONFIG_CPU_DADDI_WORKAROUNDS
8:      move      k0, sp            /*保存异常前栈地址至 k0*/
        PTR_SUBU sp, k1, PT_SIZE     /*从 k1 设置当前 sp 值*/
                                  /*如果是用户态发生的异常，sp 指向内核栈顶 pt_regs 实例基地址*/
                /*如果是内核态发生的异常，在当前内核栈预留 pt_regs 实例空间，sp 记录实例基地址*/
#else
        ...
#endif
        /*以下指令将当前部分 CPU 寄存器信息保存至内核栈 pt_regs 实例中*/
        LONG_S k0, PT_R29(sp)        /*异常前栈地址，可能是用户栈也可能是内核栈*/
        LONG_S $3, PT_R3(sp)
        LONG_S $0, PT_R0(sp)
        mfc0 v1, CP0_STATUS
        LONG_S $2, PT_R2(sp)
        LONG_S v1, PT_STATUS(sp)      /*保存状态寄存器，Status.EXL=1*/
        LONG_S $4, PT_R4(sp)
        mfc0 v1, CP0_CAUSE
        LONG_S $5, PT_R5(sp)
        LONG_S v1, PT_CAUSE(sp)      /*保存原因寄存器*/
        LONG_S $6, PT_R6(sp)
        MFC0     v1, CP0_EPC
        LONG_S $7, PT_R7(sp)
    #ifdef CONFIG_64BIT
        ...
```

```
#endif
    LONG_S   v1, PT_EPC(sp)        /*保存异常返回地址*/
    LONG_S $25, PT_R25(sp)
    LONG_S $28, PT_R28(sp)
    LONG_S $31, PT_R31(sp)
    ori   $28, sp, _THREAD_MASK    /*栈地址对齐后就是 thread_info 实例地址*/
    xori $28, _THREAD_MASK         /*$28 保存 thread_info 实例地址*/
  #ifdef CONFIG_CPU_CAVIUM_OCTEON
    ...
  #endif
    .set   pop
    .endm
```

SAVE_SOME 宏主要操作如下图所示，kernelsp[]数组保存了各 CPU 核当前运行进程内核栈初始栈顶地址，如果异常在用户态发生，则由 kernelsp[]数组项获取当前内核栈地址（需要切换栈），如果异常在内核态发生，则不需要切换栈。然后，在当前内核栈中为 pt_regs 实例预留空间，sp 记录实例基地址。随后，保存部分寄存器信息至 pt_regs 实例。最后，将 sp 地址对齐后赋予$28 寄存器，即当前进程 thread_info 实例地址。



SAVE_AT、SAVE_TEMP、SAVE_STATIC 宏都定义在/arch/mips/include/asm/stackframe.h 头文件内，它们在 SAVE_SOME 宏之后，将其它指定的通用寄存器信息保存至内核栈（pt_regs 实例），源代码请读者自行阅读。

注意，在进入异常处理程序时，内核只保存了通用寄存器，没有保存浮点寄存器等专用寄存器，因为内核代码中不会使用这些寄存器，所以不用保存。但是，如果在异常处理程序内部发生了进程调度，异常还没返回，CPU 就去运行下一个进程了，在进程切换时就需要保存这些专用寄存器了，因为下一个进程可能会使用这些专用寄存器，要保存前一进程的上下文信息。

2 恢复寄存器

异常处理程序在保存寄存器后，调用各异常处理函数，随后执行内核插入的工作（如果需要），最后恢复寄存器从异常返回。

异常处理程序最后会跳转到 restore_all 等标号处执行，恢复寄存器（异常返回），如下所示：

```
restore_all:                /*恢复所有寄存器，保存寄存器的逆操作，/arch/mips/kernel/entry.S*/
    .set   noat             /*sp 已指向内核栈 pt_regs 实例*/
```

```
        RESTORE_TEMP
        RESTORE_AT
        RESTORE_STATIC
    restore_partial:        /*只需恢复部分寄存器*/
    #ifdef CONFIG_TRACE_IRQFLAGS
        ...
    #endif
        RESTORE_SOME        /*恢复寄存器*/
        RESTORE_SP_AND_RET    /*从异常返回*/
```

恢复寄存器是保存寄存器的逆操作，而且顺序要相反，即先保存的后恢复，后保存的先恢复。下面看一下 RESTORE_SOME 和 RESTORE_SP_AND_RET 宏的定义（/arch/mips/include/asm/stackframe.h）：

```
        .macro    RESTORE_SOME
        .set  push
        .set  reorder
        .set  noat
        mfc0 a0, CP0_STATUS      /*读取当前 Status（状态寄存器）值至 a0*/
        ori   a0, STATMASK     /*a0|0x1f，即低 5 位全置 1*/
        xori  a0, STATMASK     /*低 5 位清零*/
        mtc0  a0, CP0_STATUS      /*写 Status，内核态，关中断（EXL=0，IE=0）*/
        li    v1, 0xff00
        and  a0, v1          /*a0 只留 bit[8...15]，其它位清零*/
        LONG_L v0, PT_STATUS(sp)        /*v0 保存刚进入异常时的状态寄存器值（EXL=1）*/
        nor  v1, $0, v1        /*或非，v1 中 bit[8...15]清零，其它位置 1*/
        and  v0, v1          /*刚进入异常状态寄存器 bit[8...15]值清零（屏蔽所有中断），其它位保留*/
        or    v0, a0          /*用当前 Status.bit[8...15]替换刚进入异常时 Status.bit[8...15]*/
                             /*也就是说异常返回前后 Status.bit[8...15]保持不变*/
        mtc0 v0, CP0_STATUS     /*写 Status，恢复刚进入异常时 Status 值（只处理了中断屏蔽标记）*/
        LONG_L v1, PT_EPC(sp)
        MTC0    v1, CP0_EPC        /*写 EPC 寄存器*/
        LONG_L $31, PT_R31(sp)
        LONG_L $28, PT_R28(sp)
        LONG_L $25, PT_R25(sp)
    #ifdef CONFIG_64BIT
        ...
    #endif
        LONG_L $7,    PT_R7(sp)
        LONG_L $6,    PT_R6(sp)
        LONG_L $5,    PT_R5(sp)
        LONG_L $4,    PT_R4(sp)
        LONG_L $3,    PT_R3(sp)
        LONG_L $2,    PT_R2(sp)
```

```
.set  pop
.endm     /*RESTORE_SOME 宏结束*/

.macro    RESTORE_SP_AND_RET
LONG_L sp, PT_R29(sp)       /*恢复异常前栈指针*/
.set  arch=r4000
eret          /*执行异常返回指令，PC 跳转至 EPC 寄存器保存的地址处取指*/
.set  mips0
.endm     /*宏定义结束*/
```

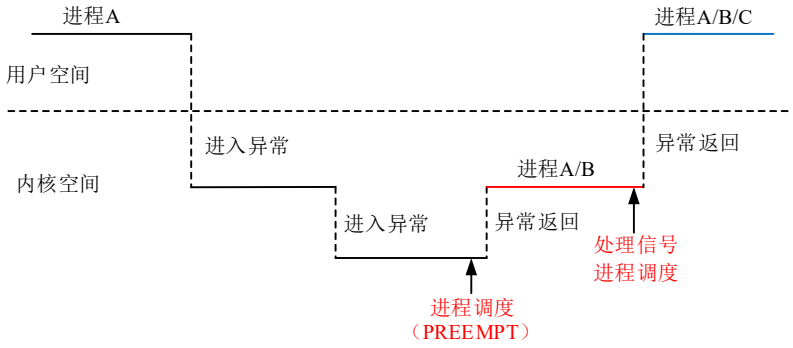
RESTORE_SOME 宏用于恢复 SAVE_SOME 宏中保存的寄存器，包括状态寄存器、EPC 寄存器，以及其它一些通用寄存器。恢复状态寄存器的值为刚进入异常时在 SAVE_SOME 宏中保存的值，只不过中断使能 Status.bit[8...15]标记位，保持异常处理程序中的状态不变（不恢复异常前的状态）。

RESTORE_SP_AND_RET 宏恢复异常发生前栈指针，执行 eret 指令，从异常返回，此后 PC 从 EPC 保存的地址处（异常返回地址）开始取指。

3 异常返回前工作

异常处理程序在恢复寄存器从异常返回前，还需要检查是否要执行内核插入的工作，主要是处理挂起的信号和进程调度。

异常处理程序返回用户空间前，通常都需要执行返回前工作，包括处理挂起的信号和进程调度。异常处理程序返回内核空间前，如果启用了内核抢占（选择 PREEMPT 选项），则需要处理进程调度，否则直接返回。异常处理程序返回内核空间前不会处理挂起的信号，如下图所示。



■返回用户空间前工作

异常处理程序在调用异常处理函数后，通常会跳转至 resume_userspace 等标号处，检查是否有需要执行的工作，如果有则执行，执行完后返回，如果没有则直接返回。

resume_userspace 等标号定义在/arch/mips/kernel/entry.S 文件内：

```
resume_userspace_check:     /*检查是否返回用户态*/
    LONG_L  t0, PT_STATUS(sp)          /*异常发生前处理器状态*/
    andi t0, t0, KU_USER            /*bit4 是否为 0*/
    beqz  t0, resume_kernel       /*bit4 为 0 返回内核空间跳转至 resume_kernel，否则往下执行*/
```

```
resume_userspace:            /*异常返回用户空间前执行的代码*/
    local_irq_disable   /*关闭本地中断（本 CPU 核，IE=0），/arch/mips/include/asm/asmmacro,h*/
    ...
    LONG_L  a2, TI_FLAGS($28)          /*a2=thread_info.flags*/
    andi t0, a2, _TIF_WORK_MASK          /*检测是否有挂起的工作*/
    bnez    t0, work_pending                  /*处理工作*/
    j     restore_all      /*没有需要处理的工作了，恢复寄存器，从异常返回*/
```

resume_userspace 宏检测 thread_info.flags 标记成员，判断是否有需要执行的工作，如果有则跳转至标号 work_pending 处执行，否则恢复寄存器，从异常返回。

_TIF_WORK_MASK 宏定义在/arch/mips/include/asm/thread_info.h 头文件：

```
#define _TIF_WORK_MASK          \                /*标记中断/异常返回前工作*/
    (_TIF_SIGPENDING | _TIF_NEED_RESCHED | _TIF_NOTIFY_RESUME)

/*标记返回用户空间的工作*/
#define _TIF_ALLWORK_MASK      (_TIF_NOHZ | _TIF_WORK_MASK |          \
                    _TIF_WORK_SYSCALL_EXIT | _TIF_SYSCALL_TRACEPOINT)
```

由以上标记位可知，需要处理的工作主要有挂起的信号和进程调度。

处理挂起工作的标号 work_pending 定义在/arch/mips/kernel/entry.S 文件内，如下所示：

```
work_pending:
    andi t0, a2, _TIF_NEED_RESCHED     /*是否需要重调度*/
    beqz  t0, work_notifysig      /*不需要执行重调度，跳转至 work_notifysig 处理挂起信号*/

work_resched:          /*需要执行重调度*/
    jal   schedule       /*执行进程调度，进程再次运行时，从此处开始*/

    local_irq_disable          /*关本地中断*/
    LONG_L    a2, TI_FLAGS($28)
    andi t0, a2, _TIF_WORK_MASK     /*是否有挂起工作*/
    beqz  t0, restore_all                  /*没有挂起工作，异常返回*/
    andi    t0, a2, _TIF_NEED_RESCHED        /*有挂起工作，是不是重调度*/
    bnez    t0, work_resched      /*是重调度，继续执行重调度*/

work_notifysig:          /*重调度都处理完了，处理挂起的信号*/
    move a0, sp     /*a0=sp（pt_regs 实例指针），do_notify_resume()函数第一个参数*/
    li    a1, 0        /*a1=0，第二个参数*/
    jal   do_notify_resume      /*处理挂起信号等，第三个参数为 a2，即 thread_info.flags 值*/
    j     resume_userspace_check
                /*处理了若干个挂起的信号（可能还有），再次跳转至 resume_userspace_check 处，
                *直至没有挂起的工作才返回用户空间。*/
```

由以上代码可知，在异常处理程序返回用户空间前，将检测 thread_info.flags 标记成员，看是否有挂起的信号和是否需要重调度，如果有则优先执行重调度，再处理挂起的信号，等到没有需要执行的工作后，才返回用户空间。

处理重调度就是调用 schedule()函数执行进程调度，挂起的信号等由 do_notify_resume()函数处理，这两个函数详见第 5 章。

■返回内核空间前工作

如果内核配置支持内核抢占（PREEMPT），在异常处理程序返回内核空间前，将检测是否需要执行进程调度，需要则执行进程调度，然后异常返回。如果内核配置不支持内核抢占，异常处理程序直接返回内核空间。异常处理程序返回内核空间前，不处理挂起的信号。

即使内核支持内核抢占，也需要当前进程的抢占计数值为 0，且设置了 TIF_NEED_RESCHED 标记位，才会在异常处理程序返回内核空间前执行进程调度。

异常处理程序返回内核空间前执行代码标号定义在/arch/mips/kernel/entry.S 文件内，如下所示：

```
#ifndef CONFIG_PREEMPT      /*不支持内核抢占*/
    #define resume_kernel  restore_all      /*恢复寄存器，直接异常返回*/
#else
    #define __ret_from_irq   ret_from_exception
#endif
    ...
#ifdef CONFIG_PREEMPT       /*支持内核抢占，异常返回前执行的代码*/
resume_kernel:      /*返回内核空间标号*/
    local_irq_disable       /*关闭本地中断*/
    lw    t0, TI_PRE_COUNT($28)     /*t0=thread_info.preempt_count，为 0 表示可以抢占*/
    bnez    t0, restore_all     /*不为 0，不可以抢占，直接返回内核空间*/

need_resched:               /*抢占计数为 0，再检测重调度标记位*/
    LONG_L   t0, TI_FLAGS($28)
    andi t1, t0, _TIF_NEED_RESCHED
    beqz   t1, restore_all              /*没有设置 TIF_NEED_RESCHED 标记位，返回内核空间*/
    LONG_L   t0, PT_STATUS(sp)   /*异常发生前（刚进入异常时），IE 是否为 0（关中断）*/
    andi    t0, 1
    beqz   t0, restore_all              /*关闭了中断，直接返回内核空间*/
    jal     preempt_schedule_irq     /*打开了中断，执行进程调度*/
    b       need_resched       /*再次检测是否需要执行重调度*/
#endif
```

由以上代码可知，在异常处理程序返回内核空间前执行进程调度需要同时满足以下条件：

（1）内核支持内核抢占。

（2）当前进程抢占计数 preempt_count 值为 0。

（3）当前进程设置了 TIF_NEED_RESCHED 标记位。

（4）本次异常发生前（刚进入异常处理程序时）CPU 处于开中断状态。

此处执行进程调度的 **preempt_schedule_irq()**函数定义在/kernel/sched/core.c 文件内：

```
asmlinkage __visible void __sched preempt_schedule_irq(void)       /*内核抢占调度*/
{
    enum ctx_state prev_state;
    BUG_ON(preempt_count() || !irqs_disabled());          /*当前 CPU 需是中断*/
    prev_state = exception_enter();
    do {
        preempt_active_enter();   /*抢占计数加 1，并设置抢占计数中 PREEMPT_ACTIVE 标记位*/
        local_irq_enable();       /*开中断*/
        __schedule();             /*进程调度*/
        local_irq_disable();      /*关中断*/
        preempt_active_exit();    /*抢占计数减 1，并清零抢占计数中 PREEMPT_ACTIVE 标记位*/
    } while (need_resched());       /*检测 TIF_NEED_RESCHED 标记位，是否需要重调度*/
    exception_exit(prev_state);
}
```

在 resume_kernel 标号处会先关闭本地中断，然后调用 preempt_schedule_irq()函数执行进程调度。本进程再次运行时，若没有设置重调度标记，则恢复寄存器，从异常返回（恢复原中断状态）。

6.2 TLB 异常

前面介绍了 MIPS 异常的类型和异常向量，以及异常处理程序的流程，从本节开始介绍几个具体的 MIPS 异常的处理，主要包括 TLB 异常、系统调用和中断。

CPU 产生的程序地址（虚拟地址）通常经过 MMU 转换成物理地址，然后输出到地址总线寻址内存。如果在 MMU 中没有找到程序地址匹配的 TLB 表项或匹配表项无效或访问权限错误，CPU 将产生 TLB 异常，异常处理程序需要将正确有效的页表项写入 TLB，以便实现地址的转换。

本节介绍 MIPS 体系结构的 TLB 异常处理程序。

6.2.1 TLB 异常类型

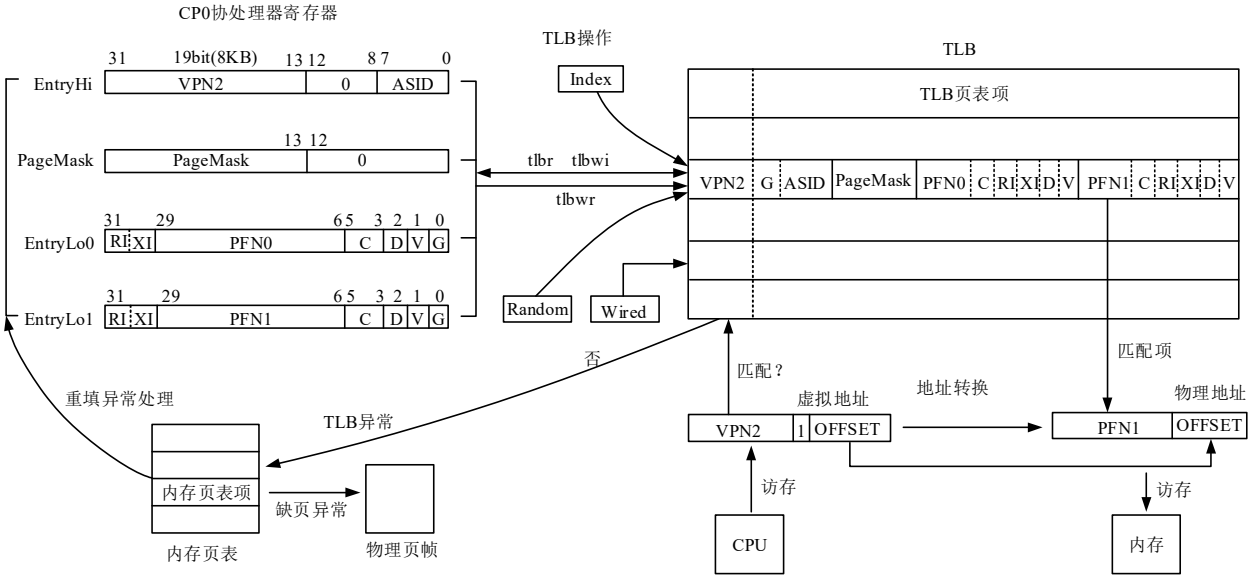
前面第 3 章介绍过 Linux 内核一般将物理内存以 4KB 为大小进行划分，称为页帧，物理内存的管理以页帧为单位。地址转换就是将程序产生的虚拟地址转换成寻址物理内存的物理地址，虚拟地址的页内偏移量（低 12 位）直接转为物理地址低位，而虚拟地址高位位域（高 20 位）用物理页帧号（PFN）替换，从而生成物理地址传输到地址总线寻址物理内存。简单地说就是用物理页帧号替代虚拟地址中的虚拟页帧号生成物理地址。

进程页表（保存在内存中）是进行地址转换的依据，记录了虚拟页对应的物理页，以及访问权限。进程页表可视为一个数组，每个数组项表示一个虚拟页与物理页的转换关系，虚拟页帧号作为数组的索引值。CPU 为提高地址转换的速度，在 MMU 内设置了页表项的缓存称之为转换后备缓存（TLB）。TLB 中每个缓存项保存了匹配的虚拟地址页号，转换后的物理页帧号，访问权限等信息。程序虚拟地址首先与 TLB 缓存项匹配，如果有匹配的有效项且访问权限也匹配，则直接进行地址转换，用缓存项中保存的物理页号替换虚拟地址中的虚拟页号，合成物理地址传输到地址总线寻址物理内存。

在地址转换过程中，如果在 TLB 中没有找到匹配的项，CPU 将产生 TLB 重填异常，异常处理程序需要从内存页表中查找虚拟地址对应的页表项，并加载到 TLB，异常返回。Status.EXL=0 时与 Status.EXL=1 时的 TLB 重填异常具有不同的处理程序入口地址。

如果有虚拟地址匹配的 TLB 项，但其有效性标记位 V 为 0（页表项无效），将产生 TLB 无效异常。如果有虚拟地址匹配的 TLB 项，但操作是对页表项 D 标记位为 0 的页进行写操作，将产生 TLB 修改异常。另外，还有读障碍、执行障碍等 TLB 异常。

MIPS 处理器 TLB 如下图所示，每条表项包含 16 个字节、共 4 个字：



第一个字包含 TLB 表项匹配的虚拟页号、地址空间标识符 ASID（相当于进程号，用于区分不同进程的地址空间）和全局标记位 G（G 置位表示适用于所有进程，不需要匹配 ASID 域）。

第二个字包含地址转换掩码 PageMask（4KB 页面时全为 0，具体参考体系结构手册），用于屏蔽虚拟地址 VPN 中的位域，以实现不同的页大小。

第三和第四个字分别包含映射物理页的页帧号（页大小为 4KB 时为高 20 位）及访问属性。MIPS 体系结构每个 TLB 表项包含两个映射页面信息，虚拟地址 bit12 位确定是选择偶数映射页还是奇数页。访问属性 C 表示物理页的缓存属性，RI 表示读阻碍属性，XI 表示执行阻碍属性，V 表示页表项有效性属性。

地址转换的步骤如下：

（1）将虚拟地址 VPN2 与 TLB 各表项中保存的 VPN2 位域同时进行比较，如果具有数值相同的表项，且表项内 ASID 值与 EntryHi 寄存器内 ASID 位域值相同或 G 位为 1，则表示虚拟地址与此页表项匹配。如果没有查寻到匹配表项，将发生 TLB 重填异常。

（2）读取匹配页表项内 PFN、C、RI、XI、D、V 位域数值，如果 V 位为 0，将产生 TLB 无效异常；如果 D 位为 0，且操作为存储指令，则产生 TLB 修改异常；如果 RI 位为 1，且为加载指令，将产生 TLB 无效异常或 TLBRI 异常；如果 XI 位为 1，且为取指操作，将产生 TLB 无效异常或 TLBXI 异常。

（3）如果没有发生异常，将用匹配 TLB 表项的 PFN 位域替换虚拟地址中 VPN2 位域（低位保持不变），合成的物理地址输出到地址总线寻址物理内存。

当产生 TLB 重填异常时，异常处理程序需要在进程的内存页表项中查找对应的页表项，通过协处理器 CP0 中的寄存器和 TLB 指令，将内存页表项写入 TLB 中。异常返回后将再次在 TLB 中查找虚拟地址匹配的 TLB 表项，此时匹配能成功（重填异常返回地址为发生异常的指令地址）。如果没有发生其它的 TLB 异常，将能正确地执行地址转换，程序继续执行。

如果 TLB 中有匹配的页表项，但是还是发生了 TLB 异常，则需要对内存页表项进行处理，这称为缺页异常处理。内存页表项处理包括分配/查找物理页帧、设置内存页表项、写入 TLB 页表项等。

CPU 协处理器 CP0 中与 TLB 操作相关的寄存器和 TLB 指令，请读者自行参考 MIPS 体系结构手册。这里总结一下 TLB 异常类型，如下表所示：

异常编号	发生条件	异常类型	异常向量
TLBL	TLB 中没有匹配项，加载数据或取指令，Status(EXL)=0。	TLB 重填异常	**EBase**
	TLB 中没有匹配项，加载数据或取指令，Status(EXL)=1。	异常级 TLB 重填异常（处于核心态）	EBase+0x180
	TLB 中有匹配项，V=0，加载数据或取指令。	TLB 无效异常	
	TLB 中有匹配项，RI=1， PageGrain(RIE)=1，PageGrain(IEC)=0，加载数据。	读障碍异常	
	TLB 中有匹配项，XI=1， PageGrain(XIE)=1，PageGrain(IEC)=0，取指令。	执行障碍异常	
TLBS	TLB 中没有匹配项，存储数据，Status(EXL)=0。	TLB 重填异常	**EBase**
	TLB 中没有匹配项，存储数据，Status(EXL)=1。	异常级 TLB 重填异常	EBase+0x180
	TLB 中有匹配的项，V=0，存储数据。	TLB 无效异常	
TLBM	LB 中有匹配的项且有效，存储操作，D=0（页不可写）。	TLB 修改异常	
TLBRI	TLB 中有匹配项，RI=1， PageGrain(RIE)=1，PageGrain(IEC)=1，加载数据。	读障碍异常	
TLBXI	TLB 中有匹配项，XI=1， PageGrain(XIE)=1，PageGrain(IEC)=1，取指。	执行障碍异常	

各类型 TLB 异常处理程序入口地址（函数指针）如下表所示：

异常编号	处理程序入口地址	备注
TLBL	handle_tlbl	/arch/mips/mm/tlb-funcs.S
	ebase	**TLB 重填异常，final_handler**
TLBS	handle_tlbs	/arch/mips/mm/tlb-funcs.S
	ebase	**TLB 重填异常，final_handler**
TLBM	handle_tlbm	修改异常，/arch/mips/mm/tlb-funcs.S
TLBRI	tlb_do_page_fault_0	读障碍异常，/arch/mips/mm/tlb-fault.S
TLBXI	tlb_do_page_fault_0	执行障碍异常，/arch/mips/mm/tlb-fault.S

以上 TLB 异常处理程序中，除 TLB 重填异常（Status.EXL=0）处理程序直接填充在异常向量 ebase 处外，其它异常处理程序入口地址都保存在 exception_handlers[]对应数组项中，由 except_vec3_generic() 普通异常处理程序调用。

6.2.2 TLB 异常处理程序

MIPS 体系结构中，内核为 TLB 重填异常（Status.EXL=0）设置了专门的入口地址（ebase），TLB 重填异常可以发生在用户态（Status.KSU=0b10），也可以发生在内核态（Status.KSU=0b00）

TLBL、TLBS、TLBM 异常为普通异常，异常处理程序分别为 handle_tlbl、handle_tlbs、handle_tlbm。在这三个函数的定义处只是为函数预留了空间，并没有代码。用户态 TLB 重填异常处理程序也是一样的，只预留了空间没有代码。

初始化函数 trap_init()将调用 per_cpu_trap_init(true)函数为以上函数体填充代码（指令），因为对于不同的处理器，异常处理函数可能不同，因此在内核启动时动态地对函数体写入指令。

TLBRI 和 TLBXI 异常的处理程序都设为 tlb_do_page_fault_0，这是内核静态定义好的函数。

除 TLB 重填异常外，TLBL、TLBS、TLBM、TLBRI 和 TLBXI 异常都表示内存中原有的页表项不可用（或访问权限不对），需要修改页表项，即要建立物理页映射或改变访问权限等，也就是需要对映射物理页进行处理，因此都称之为缺页异常（不能直接访问映射的页，就缺页了）。

本小节先介绍内核代码中填充 TLB 异常处理程序的机制，然后简要介绍 TLB 重填异常和缺页异常处理程序。

1 填充异常处理程序

内核在/arch/mips/mm/tlb-funcs.S 文件内为异常处理程序 handle_tlbl、handle_tlbs 和 handle_tlbm 预留了内存空间。内核在/arch/mips/mm/tlbex.c 文件内定义了 final_handler[]数组用于暂存 TLB 重填异常处理程序，指令生成后会复制到 TLB 重填异常向量中。

在前面介绍的初始化异常向量的 trap_init()函数中将调用 **per_cpu_trap_init(true)**函数，为异常处理程序 handle_tlbl、handle_tlbs 和 handle_tlbm 等生成代码（指令），生成 TLB 重填异常处理程序填充至数组 final_handler[]，并复制到异常向量基地址 ebase 处。

```
void __init trap_init(void)
{
    ...
    per_cpu_trap_init(true);     /*生成/填充 TLB 异常处理程序*/
    ...
}
```

per_cpu_trap_init()函数定义在/arch/mips/kernel/traps.c 文件内，代码简列如下：

```
void per_cpu_trap_init(bool is_boot_cpu)
/*is_boot_cpu：当前是否是启动 CPU 核，非启动核启动时也会调用此函数*/
{
    unsigned int cpu = smp_processor_id();

    configure_status();          /*设置 CPU 状态寄存器，traps.c*/
    configure_hwrena();          /*设置 CPU 硬件寄存器使能寄存器，traps.c*/

    configure_exception_vector();  /*非兼容中断模式时，设置向量中断控制寄存器 intctl 等*/

    if (cpu_has_mips_r2_r6) {        /*MIPSr2 或 MIPSr6*/
        cp0_compare_irq_shift = CAUSEB_TI - CAUSEB_IP;
        cp0_compare_irq = (read_c0_intctl() >> INTCTLB_IPTI) & 7;     /*硬件定时器中断号*/
        cp0_perfcount_irq = (read_c0_intctl() >> INTCTLB_IPPCI) & 7;   /*性能计数中断号*/
        cp0_fdc_irq = (read_c0_intctl() >> INTCTLB_IPFDC) & 7;
        if (!cp0_fdc_irq)
            cp0_fdc_irq = -1;
```

```
    } else {
        ...
    }

    if (!cpu_data[cpu].asid_cache)
        cpu_data[cpu].asid_cache = ASID_FIRST_VERSION;      /*初始 ASID 值*/

    atomic_inc(&init_mm.mm_count);      /*增加初始地址空间引用计数*/
    current->active_mm = &init_mm;
    BUG_ON(current->mm);
    enter_lazy_tlb(&init_mm, current);      /*空操作*/

    if (!is_boot_cpu)
        cpu_cache_init();     /*非启动 CPU 核初始化 CPU 缓存，/arch/mips/mm/cache.c*/
    tlb_init();      /*填充 TLB 异常处理程序，/arch/mips/mm/tlb-r4k.c*/
    TLBMISS_HANDLER_SETUP();
                /*设置 CONTEXT 寄存器，/arch/mips/include/asm/mmu_context.h*/
}
```

per_cpu_trap_init()函数内主要是调用 tlb_init()函数初始化 TLB，以及完成 TLB 异常处理程序的填充。tlb_init()函数根据 CPU 类型的不同具有不同的实现。在/arch/mips/Kconfig 配置文件内有如下定义：

```
config CPU_R4K_CACHE_TLB
    bool
    default y if !(CPU_R3000 || CPU_R8000 || CPU_SB1 || CPU_TX39XX || CPU_CAVIUM_OCTEON)
```

龙芯 1B 处理器适应 CPU_R4K_CACHE_TLB 配置选项。

在/arch/mips/mm/Makfile 构建文件内有如下语句：

```
obj-$(CONFIG_CPU_R4K_CACHE_TLB) += c-r4k.o cex-gen.o tlb-r4k.o
```

因此对于龙芯 1B 处理器，将调用/arch/mips/mm/tlb-r4k.c 文件内定义的 tlb_init()函数，代码如下：

```
void tlb_init(void)
{
    r4k_tlb_configure();  /*设置 TLB 相关 CP0 寄存器，刷新 TLB 等，/arch/mips/mm/tlb-r4k.c*/

    if (ntlb) {          /*设置 wired、index 寄存器*/
        if (ntlb > 1 && ntlb <= current_cpu_data.tlbsize) {
            int wired = current_cpu_data.tlbsize - ntlb;
            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();  /*填充 TLB 异常处理程序，/arch/mips/mm/tlbex.c*/
}
```

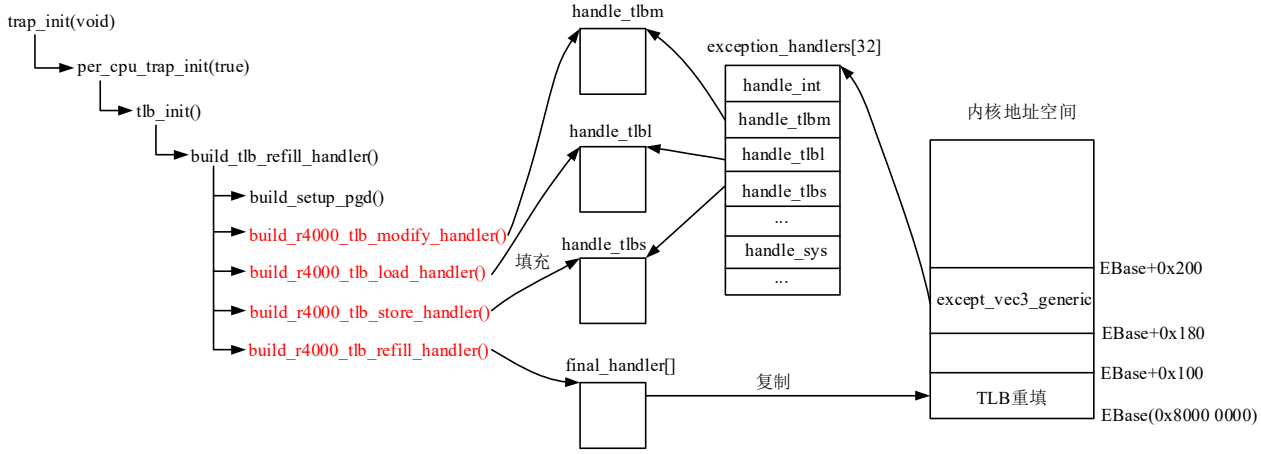
tlb_init()函数内设置完 CP0 中 TLB 相关寄存器后，调用 **build_tlb_refill_handler()**函数填充 TLB 异常处理程序。

build_tlb_refill_handler()函数定义在/arch/mips/mm/tlbex.c 文件内，代码简列如下：

```
void build_tlb_refill_handler(void)
{
    static int run_once = 0;
    output_pgtable_bits_defines();

  #ifdef CONFIG_64BIT
    ...
  #endif

    switch (current_cpu_type()) {      /*根据 CPU 类型调用不同的填充函数*/
     ...
     default:                          /*龙芯 1B 处理器匹配项*/
        if (!run_once) {       /*if 内语句只执行一遍*/
            scratch_reg = allocate_kscratch();
            build_setup_pgd();        /*填充 tlbmiss_handler_setup_pgd()函数，/arch/mips/mm/tlbex.c*/
                              /*tlbmiss_handler_setup_pgd()函数将当前 PGD 地址 pgd_current[cpu]数组项*/
            build_r4000_tlb_load_handler();       /*填充 TLBL 异常处理程序*/
            build_r4000_tlb_store_handler();      /*填充 TLBS 异常处理程序*/
            build_r4000_tlb_modify_handler();   /*填充 Mod 异常处理程序*/
            if (!cpu_has_local_ebase)
                build_r4000_tlb_refill_handler();   /*填充 TLB 重填异常处理程序*/
            flush_tlb_handlers();
            run_once++;
        }
        if (cpu_has_local_ebase)
            build_r4000_tlb_refill_handler();
        if (cpu_has_xpa)
            config_xpa_params();
        if (cpu_has_htw)
            config_htw_params();
    }
}
```

build_tlb_refill_handler()函数内根据 CPU 类型，调用不同的填充 TLB 异常处理程序函数，填充 TLB

异常处理程序。build_tlb_refill_handler()函数执行的工作如下图所示，下文中将介绍各异常处理程序代码：



2 填充 tlbmiss_handler_setup_pgd()函数

在/arch/mips/mm/tlb-funcs.S 文件内为 tlbmiss_handler_setup_pgd()函数预留了空间，build_setup_pgd()函数负责向 tlbmiss_handler_setup_pgd()函数体写入代码。

内核在/arch/mips/mm/init.c 文件内定义了整型数组 pgd_current[NR_CPUS]，用于保存各 CPU 核当前活跃地址空间全局页表基地址。

tlbmiss_handler_setup_pgd((unsigned long)(pgd))函数的作用就是将参数 pgd 表示的 PGD 页表基地址写入当前 CPU 核对应的 pgd_current[]数组项。

build_setup_pgd()函数就是为 tlbmiss_handler_setup_pgd()函数生成代码，函数定义如下：

```
static void build_setup_pgd(void)        /*/arch/mips/mm/tlbex.c*/
{
    ...
    u32 *p = tlbmiss_handler_setup_pgd_start;     /*tlbmiss_handler_setup_pgd()函数入口地址*/
    ...
#ifndef CONFIG_MIPS_PGD_C0_CONTEXT
    long pgdc = (long)pgd_current;       /*指向 pgd_current[]数组*/
#endif
        ...
    #ifdef CONFIG_SMP       /*保存 PGD 页表基地址至 pgd_current[cpu] 数组项，a0 为基地址*/
        UASM_i_CPUID_MFC0(&p, a1, SMP_CPUID_REG);       /*生成并写入指令*/
        UASM_i_SRL_SAFE(&p, a1, a1, SMP_CPUID_PTRSHIFT);       /*a1 保存 cpu 编号*/
        UASM_i_LA_mostly(&p, a2, pgdc);       /*a2 为 pgd_current[]数组基地址*/
        UASM_i_ADDU(&p, a2, a2, a1);        /*a2 表示 cpu 对应的 pgd_current[]数组项*/
        UASM_i_SW(&p, a0, uasm_rel_lo(pgdc), a2);       /*a0 写入数组项*/
    #else
        UASM_i_LA_mostly(&p, a2, pgdc);       /*单核 CPU，直接保存*/
        UASM_i_SW(&p, a0, uasm_rel_lo(pgdc), a2);
#endif      /* SMP */
    uasm_i_jr(&p, 31);       /*跳转，函数返回*/
```

```
    ...
}
```

build_setup_pgd()函数中 UASM_i_XXX()宏是生成并写入指令的机制，XXX 通常表示指令名称，宏内第一个参数是指令存入的地址，后面参数是指令的操作数。后面介绍的 TLB 异常处理程序也是用这种机制生成和写入指令的。

build_setup_pgd()函数为 tlbmiss_handler_setup_pgd()函数体写入指令，函数体其实很简单就是从 CPU 寄存器中获取当前 CPU 核编号，将 a0 参数传递的 PGD 页表基地址写入 pgd_current[cpu]数组项，函数返回。

在切换地址空间的 switch_mm()函数中会调用 tlbmiss_handler_setup_pgd()函数，把将要使用地址空间的 PGD 基地址写入 pgd_current[cpu]数组项。

3 TLB 重填异常

内核中生成并填充 TLB 重填异常处理程序的函数为 build_r4000_tlb_refill_handler()，函数代码简列如下（/arch/mips/mm/tlbex.c）：

```
static void build_r4000_tlb_refill_handler(void)
{
    u32 *p = tlb_handler;       /*整数数组，暂存生成的指令*/
    ...
    if (...)    {
        ...
    } else {
        ...
    #ifdef CONFIG_64BIT
        ...
    #else
        build_get_pgde32(&p, K0, K1);     /*获取 PGD 页表项地址写入 K1，/arch/mips/mm/tlbex.c*/
    #endif
        ...
        build_get_ptep(&p, K0, K1);              /*获取 PTE 页表项地址，写入 K1*/
        build_update_entries(&p, K0, K1);        /*写 TLB 相关寄存器*/
        build_tlb_write_entry(&p, &l, &r, tlb_random);      /*写入 TLB 表项*/
        uasm_l_leave(&l, p);
        uasm_i_eret(&p);        /*异常返回指令*/
    }
    ...      /*tlb_handler[]指令复制到 final_handler[]数组*/
    memcpy((void *)ebase, final_handler, 0x100);       /*final_handler[]指令复制到异常向量*/
    local_flush_icache_range(ebase, ebase + 0x100);     /*刷新本地指令缓存*/
    ...
}
```

build_r4000_tlb_refill_handler()函数将生成的指令先写入 tlb_handler[]数组，然后复制到 final_handler[]数组，最后将 final_handler[]中指令复制到 ebase 表示的异常向量处。

TLB 重填异常发生时，Status.EXL 设为 1，禁止中断，CPU 处于内核态，BadVAddr 寄存器保存了引发异常的地址。生成 TLB 重填异常处理程序（指令）的步骤如下（处理程序中只使用 K0 和 K1 寄存器）：

（1）build_get_pgde32(&p, K0, K1)：生成的指令根据当前 CPU 核编号获取 pgd_current[cpu]数组项值，即 PGD 基地址；根据 BadVAddr 寄存器值，查找引发异常的地址对应的 PGD 表项，表项地址写入 K1。

（2）build_get_ptep(&p, K0, K1)：生成的指令根据 K1 保存的 PGD 页表项和 Context 寄存器中保存的引发异常的地址，获取 PTE 页表项地址，写入 K1。

（3）build_update_entries(&p, K0, K1)：生成的指令将 K1 指向的 PTE 页表项（2 项）分别右移（6 位）后写入 EntryLo0、EntryLo1 寄存器。

（4）build_tlb_write_entry(&p, &l, &r, tlb_random)：生成的指令主要是写 TLB 指令，这里是 tlbwr 指令，即随机选择一个 TLB 表项，将 EntryLo0、EntryLo1 等寄存器中信息写入此表项。

（5）uasm_i_eret(&p)：生成 eret 指令，从异常返回。

由于 TLB 重填异常是发生频率很高的异常，因此异常处理程序要求快速执行。在异常处理程序中只使用了 K0 和 K1 寄存器，因此不需要保存和恢复寄存器，异常处理程序在关中断的状态下进行。

4 缺页异常

MIPS 体系结构中，除用户态 TLB 重填异常外，其它 TLB 异常都视为普通异常。

在 build_tlb_refill_handler(void)函数中将会为普通 TLB 异常处理程序填充代码，如下所示：读者可自

（1）build_r4000_tlb_load_handler()：为 TLBL 异常处理程序 handle_tlbl 填充代码。

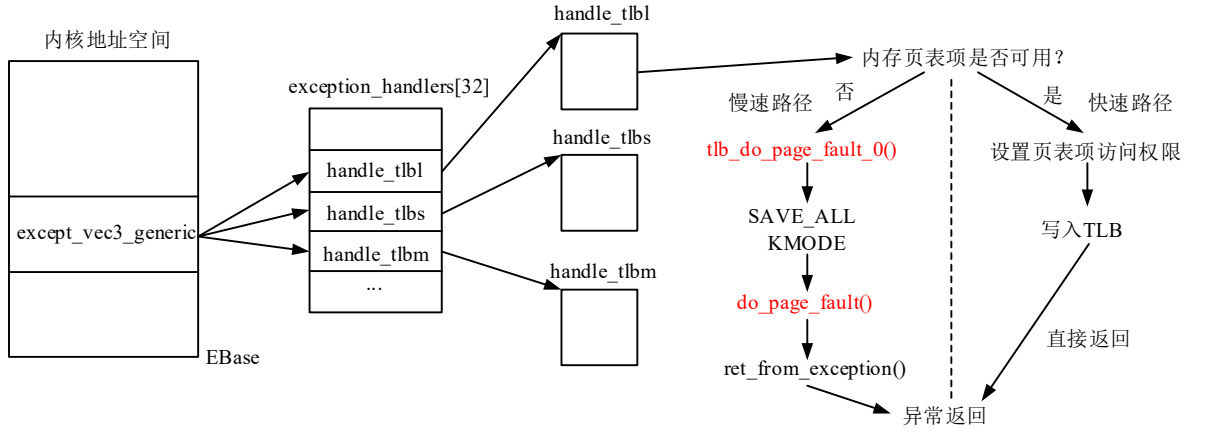
（2）build_r4000_tlb_store_handler()：为 TLBS 异常处理程序 handle_tlbs 填充代码。

（3）build_r4000_tlb_modify_handler()：为 TLBM 异常处理程序 handle_tlbm 填充代码。

这三个 TLB 异常的处理程序都类似，下面以 handle_tlbl 处理程序为例，说明生成的代码主要执行的工作。如下图所示，handle_tlbl 处理程序首先检查引发异常的虚拟地址对应的内存页表项，对本次访问是否可继续使用，然后根据检查结果进入快速路径或慢速路径，如下所述。

（1）如果内存页表项可用，则进入快速路径，设置内存页表项访问权限，将内存页表项写入 TLB，异常返回，这类似于 TLB 重填异常（隐含异常级 TLB 重填异常）。

（2）如果内存页表项不可用，则进入慢速路径。慢速路径由 tlb_do_page_fault_0()函数执行，函数内首先保存 CPU 寄存器信息，然后调用 do_page_fault()函数处理缺页异常，最后恢复寄存器从异常返回。



handle_tlbs 和 handle_tlbm 处理程序与 handle_tlbl 类似，只不过慢速路径调用 tlb_do_page_fault_1()函数。在进入慢速路径之前和快速路径中，程序代码中都只使用 K0 和 K1 寄存器（禁止中断状态），因此不用保存和恢复寄存器。

tlb_do_page_fault_0()和 tlb_do_page_fault_1()函数其实是一个函数，只不过参数不同。前者用于处理读操作（或取指）产生的异常，后者用于处理写操作产生的异常。

这两个函数定义在/arch/mips/mm/tlb-fault.S 文件内：

```
.macro tlb_do_page_fault, write                 /*write 为 0 表示读，1 表示写*/
NESTED(tlb_do_page_fault_\write, PT_SIZE, sp)
SAVE_ALL
                /*保存 CPU 寄存器信息至进程栈 pt_regs 实例，/arch/mips/include/asm/stackframe.h*/
MFC0    a2, CP0_BADVADDR        /*a2 保存引发异常的虚拟地址，第三个参数*/
KMODE                           /*CPU 进入内核态，中断状态不变，/arch/mips/include/asm/stackframe.h*/
                                /*Status：CU0=1，KSU=00（内核态），EXL=0，IE 保持不变*/
move    a0, sp                  /*a0 保存 pt_regs 实例指针，do_page_fault()函数第一个参数*/
REG_S   a2, PT_BVADDR(sp)       /*保存引发异常地址至 pt_regs 实例*/
li      a1, \write              /*a1 保存引发异常的是写操作还是读操作，第二个参数*/
PTR_LA ra, ret_from_exception   /*跳转延迟指令，异常返回地址 ret_from_exception 保存至 ra*/
j       do_page_fault           /*调用通用的缺页异常处理函数，a0，a1，a2 为参数*/
                                /*do_page_fault()函数返回后，从 ret_from_exception 处开始执行*/
END(tlb_do_page_fault_\write)
.endm

tlb_do_page_fault 0     /*定义 tlb_do_page_fault_0()函数，读操作异常处理函数*/
tlb_do_page_fault 1     /*定义 tlb_do_page_fault_1()函数，写操作异常处理函数*/
```

tlb_do_page_fault 在保存寄存器后，执行 KMODE 宏表示的代码，其功能是设置 Status，以保存 CPU 处于内核态，退出异常等级，关中断；然后调用 do_page_fault()函数处理缺页异常；最后从异常返回。

缺页异常处理函数 do_page_fault()声明如下：

```
asmlinkage void __kprobes do_page_fault(struct pt_regs *regs,unsigned long write, unsigned long address);
```

regs 参数指向内核栈 pt_regs 实例，write 参数表示是读还是写操作，address 参数表示引发异常的地址。

do_page_fault()函数的定义详见第 4 章，其主要工作是为虚拟页创建正确的映射，生成页表项写入内存页表和 TLB。

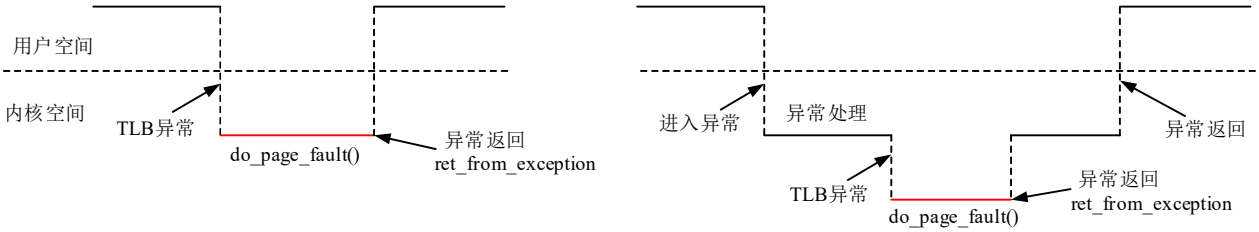
do_page_fault()函数返回后，从 ret_from_exception 处开始执行，这是准备从异常处理程序中返回。

注意：在调用 do_page_fault()函数前 CPU 的中断状态没有改变，还是 TLB 异常前的状态。

在 trap_init()函数中将 TLBRI 和 TLBXI 异常的处理程序设为 tlb_do_page_fault_0()函数，这两个异常的处理程序没有快速路径，直接进入慢速路径。

6.2.3 TLB 异常返回

TLB 异常可能发生在用户态，也可能发生在内核态。在异常处理程序的快速路径中，直接执行 eret 指令从异常返回。在慢速路径中，则在 do_page_fault()函数返回后跳转至 ret_from_exception 标号处执行，如下图所示。



ret_from_exception 标号定义在/arch/mips/kernel/entry.S 文件内，代码如下：

```
#ifndef CONFIG_PREEMPT      /*不支持内核抢占*/
    #define resume_kernel   restore_all
#else                               /*支持内核抢占*/
    #define __ret_from_irq  ret_from_exception
#endif

#ifndef CONFIG_PREEMPT      /*不支持内核抢占*/
FEXPORT(ret_from_exception)
    local_irq_disable               /*关本地中断*/
    b     __ret_from_irq
#endif
FEXPORT(ret_from_irq)
    LONG_S s0, TI_REGS($28)
FEXPORT(__ret_from_irq)

resume_userspace_check:         /*检果是返回内核空间还是用户空间*/
    LONG_L t0, PT_STATUS(sp)            # returning to kernel mode?
    andi t0, t0, KU_USER
    beqz t0, resume_kernel          /*返回内核空间，异常返回*/

resume_userspace:               /*返回用户空间*/
    local_irq_disable           /*关本地中断*/
    ...
    LONG_L a2, TI_FLAGS($28)      # current->work
    andi t0, a2, _TIF_WORK_MASK # (ignoring syscall_trace)
    bnez    t0, work_pending        /*处理挂起的工作*/
    j       restore_all       /*没有挂起工作了，恢复寄存器，异常返回*/
```

TLB 异常返回与前面介绍的通用的异常返回操作相同，不再重复介绍了。

6.3 系统调用

系统调用是内核提供给的用户程序调用内核功能的接口，内核通过系统调用向用户程序提供功能调用。系统调用通常封装在库函数中，用户程序通过库函数间接执行系统调用。系统调用对用户程序来说可视为函数调用，只不过这个函数由内核定义，在内核空间运行，并且所有进程共用这些函数。

对于用户进程来说，内核就是一个黑匣子，对用户进程不可见，系统调用是进程与内核的唯一接口，提供了可供用户进程调用的内核功能。

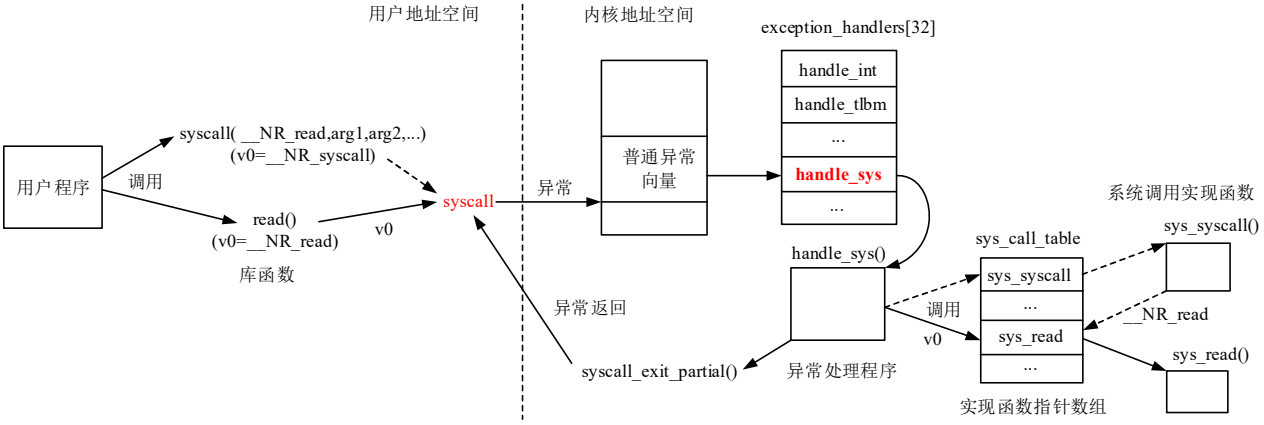
内核中的每个系统调用都有一个编号，系统调用通过执行 syscall 指令（引发 syscall 异常，寄存器 v0 传递系统调用编号）进入内核。在 syscall 异常处理程序中（handle_sys），通过系统调用编号查找内核定义的系统调用实现函数指针列表，调用相应的实现函数处理系统调用，最后从异常返回用户空间。

6.3.1 执行系统调用

系统调用执行流程如下图所示。用户程序通过库函数执行系统调用，如下图中的 read()函数。库函数内将 read 系统调用的编号__NR_read 写入 v0 寄存器，系统调用参数的传递同函数调用时参数的传递，然后执行 syscall 指令。

syscall 指令将引发异常，CPU 进入普通异常处理程序，普通异常处理程序判断出是 syscall 异常后，跳转至 handle_sys 处执行。handle_sys 就是 syscall 异常的处理程序。

异常处理程序 handle_sys()中根据系统调用编号（v0），查找内核定义的系统调用实现函数指针数组 sys_call_table[]，v0 寄存器保存的编号用于索引数组项，然后跳转至数组项中保存的系统调用实现函数处执行，执行完后从 syscall 异常返回用户空间库函数。



内核在头文件/arch/mips/include/uapi/asm/unistd.h 内定义了系统调用编号，头文件中根据_MIPS_SIM 取值，确定系统调用编号的编排和数量，如下所示：

```
#if _MIPS_SIM == _MIPS_SIM_ABI32      /*适用于 32 位处理器，非 64 位处理器*/

#define __NR_Linux          4000            /*系统调用编号起始值*/
#define __NR_syscall        (__NR_Linux +   0)  /*syscall 系统调用编号*/
#define __NR_exit           (__NR_Linux +   1)  /*exit()*/
#define __NR_fork           (__NR_Linux +   2)  /*fork()*/
#define __NR_read           (__NR_Linux +   3)  /*read 系统调用编号*/
#define __NR_write          (__NR_Linux +   4)
#define __NR_open           (__NR_Linux +   5)
#define __NR_close          (__NR_Linux +   6)
#define __NR_waitpid        (__NR_Linux +   7)
#define __NR_creat          (__NR_Linux +   8)
....
#define __NR_Linux_syscalls         356         /*系统调用总数*/
#endif    /* _MIPS_SIM == _MIPS_SIM_ABI32 */

#define    __NR_O32_Linux           4000        /*系统调用起始编号*/
```

```
#define   __NR_O32_Linux_syscalls        356        /*系统调用数量*/
```

系统调用实现函数入口地址保存在 **sys_call_table[]**数组中（/arch/mips/kernel/scall32-o32.S），简列如下，数组中的函数指针与系统调用编号一一对应：

```
EXPORT(sys_call_table)      /*系统调用执行函数指针列表*/
    PTR  sys_syscall            /*编号 4000（__NR_syscall）系统调用实现函数指针（入口地址）*/
    PTR  sys_exit
    PTR   __sys_fork
    PTR   sys_read
    PTR  sys_write
    PTR   sys_open              /* 4005 */
    ...
```

系统调用实现函数名称形如 sys_xxx()，xxx 表示系统调用名称。系统调用实现函数在内核源代码中定义。内核在/include/linux/syscalls.h 头文件内定义了声明系统调用实现函数的宏，简列如下：

```
#define SYSCALL_DEFINE0(sname)                  \
    SYSCALL_METADATA(_##sname, 0);          \    /*没有选择 FTRACE_SYSCALLS 为空*/
    asmlinkage long sys_##sname(void)           /*没有参数的系统调用实现函数*/

#define SYSCALL_DEFINE1(name, ...) SYSCALL_DEFINEx(1, _##name, __VA_ARGS__)
                                                        /*定义一个参数的系统调用*/
#define SYSCALL_DEFINE2(name, ...) SYSCALL_DEFINEx(2, _##name, __VA_ARGS__)
#define SYSCALL_DEFINE3(name, ...) SYSCALL_DEFINEx(3, _##name, __VA_ARGS__)
#define SYSCALL_DEFINE4(name, ...) SYSCALL_DEFINEx(4, _##name, __VA_ARGS__)
#define SYSCALL_DEFINE5(name, ...) SYSCALL_DEFINEx(5, _##name, __VA_ARGS__)
#define SYSCALL_DEFINE6(name, ...) SYSCALL_DEFINEx(6, _##name, __VA_ARGS__)
                                                            /*系统调用最多 6 个参数*/
```

例如，read 系统调用实现函数定义如下（/fs/read_write.c）：

```
SYSCALL_DEFINE3(read, unsigned int, fd, char __user *, buf, size_t, count)
{
    ...
}
```

read 系统调用实现函数名称为 sys_read()（必须与 sys_call_table[]数组中相同），实现函数共有三个参数，分别是 unsigned int fd，char __user * buf 和 size_t count。

系统调用实现函数声明在/include/linux/syscalls.h，例如：

```
...
asmlinkage long sys_read(unsigned int fd, char __user *buf, size_t count);
asmlinkage long sys_readahead(int fd, loff_t offset, size_t count);
...
```

用户系统（编译器）头文件中声明了封装系统调用的库函数，例如，read()库函数声明如下：

```
extern    ssize_t read (int __fd, void *__buf, size_t __nbytes) __wur;       /*/usr/include/unistd.h*/
```

read()函数是一个库函数，函数内将 read 系统调用编号__NR_read 保存至 v0 寄存器，然后执行 syscall 指令，进入 syscall 异常处理程序。异常处理程序根据系统调用编号__NR_read 查找 sys_call_table[]数组，执行__NR_read 编号对应的实现函数 **sys_read()**，执行完 sys_read()函数后跳转至 **syscall_exit_partial** 处，从异常处理程序中返回库函数 read()。库函数与系统调用实现函数之间的参数传递与普通的函数调用相同。

除了通过同名称的库函数执行系统调用外，内核还提供了直接通过系统调用编号执行系统调用的机制，它也是由系统调用完成，即 syscall 系统调用，编号为__NR_syscall。

syscall 系统调用实现函数 sys_syscall()第一个参数表示需要执行系统调用编号，后面的参数与普通的系统调用相同，实现函数定义在/arch/mips/kernel/scall32-o32.S 文件内。该函数是充当一个分配器的功能，它根据第一个参数传递的系统调用编号（不是 v0 寄存器，而是 a0 寄存器），查找系统调用实现函数指针数组 sys_call_table[]中对应的实现函数并执行，执行完后，最后从 sys 异常返回（如上图中虚线所示）。

库函数中 syscall 系统调用对应的函数为（间接系统调用）：

```
int syscall( __NR_XXX，arg1，arg2，...)      /*__NR_xxx 为实际需要执行的系统调用编号*/
```

例如：read 系统调用可通过 syscall(**__NR_read**,fd, buf,count)函数执行。

6.3.2 syscall 异常

库函数在 syscall 指令前，将系统调用编号写入 v0 寄存器，然后执行 syscall 指令，触发 syscall 异常。syscall 异常处理程序为 handle_sys。

handle_sys 定义在/arch/mips/kernel/scall32-o32.S 文件内：

```
    .align      5                                /*a0、a1、a2 等保存系统调用参数*/
NESTED(handle_sys, PT_SIZE, sp)
    .set   noat
    SAVE_SOME              /*保存 cpu 寄存器信息至 pt_reg 实例（内核态栈）*/
    TRACE_IRQS_ON_RELOAD
    STI                    /*设置 CPU 为内核态，开中断，/arch/mips/include/asm/stackframe.h*/
                           /*Status：CU0=1，KSU=00（内核态），EXL=0，IE=1（开中断）*/
    .set   at

    lw    t1, PT_EPC(sp)              /*t1 保存 syscall 指令地址*/
    subu   v0, v0, __NR_O32_Linux     /*v0 保存系统调用编号，计算相对于起始编号的偏移量*/
    sltiu   t0, v0, __NR_O32_Linux_syscalls + 1   /*系统调用编号超过最大值，则 t0=0（编号无效）*/
    addiu     t1, 4                   /*t1 保存异常返回地址，加 4 表示 syscall 下一条指令*/
    sw    t1, PT_EPC(sp)        /*保存异常返回地址至 pt_regs 实例*/
    beqz   t0, illegal_syscall     /*无效的系统调用编号，跳转至 illegal_syscall 处执行*/

    sll    t0, v0, 2             /*由编号（偏移量）计算对应 sys_call_table[]数组项*/
    la     t1, sys_call_table    /*加载 sys_call_table[]数组基地址至 t1*/
    addu    t1, t0               /*系统调用实现函数指针在 sys_call_table[]中数组项地址*/
    lw    t2, (t1)               /*实现函数入口地址保存至 t2*/
    beqz    t2, illegal_syscall      /*t2 为 0，表示无效的系统调用（或没有实现的系统调用）*/
```

```
        sw    a3, PT_R26(sp)          /*a3 参数保存到 pt_regs 实例*/
        lw    t0, PT_R29(sp)          /*t0 保存系统调用前用户态栈指针*/

        lw    t5, TI_ADDR_LIMIT($28)    /*t5 保存用户进程虚拟地址最大限制值*/
        addu    t4, t0, 32
        and  t5, t4
        bltz  t5, bad_stack         /*栈错误*/

        .set    push
        .set    noreorder
        .set   nomacro

    load_a4: user_lw(t5, 16(t0))        /*从用户栈读取其它系统调用参数值，不是由寄存器传递的*/
    load_a5: user_lw(t6, 20(t0))
    load_a6: user_lw(t7, 24(t0))
    load_a7: user_lw(t8, 28(t0))
loads_done:

        sw    t5, 16(sp)          /*系统调用参数复制到内核栈*/
        sw    t6, 20(sp)
        sw    t7, 24(sp)
        sw    t8, 28(sp)
        .set   pop

        .section __ex_table,"a"
        PTR    load_a4, bad_stack_a4
        PTR    load_a5, bad_stack_a5
        PTR    load_a6, bad_stack_a6
        PTR     load_a7, bad_stack_a7
        .previous

        lw    t0, TI_FLAGS($28)           /*获取进程底层标记值，thread_info.flags*/
        li     t1, _TIF_WORK_SYSCALL_ENTRY        /*标记系统调用前执行的工作*/
        and  t0, t1
        bnez    t0, syscall_trace_entry    # -> yes

        jalr  t2          /*跳转至实现函数处，返回地址保存至$ra，返回后继续执行下面代码*/

        li     t0, -EMAXERRNO - 1   /*处理系统调用执行函数返回值，v0 保存执行函数返回值*/
        sltu  t0, t0, v0
        sw    t0, PT_R7(sp)        # set error flag
```

```
    beqz    t0, 1f          /*返回值正常跳转至 1f*/

    lw   t1, PT_R2(sp)      /*t1 保存系统调用编号*/
    negu  v0                # error
    sw   t1, PT_R0(sp)          /*系统调用编号，保存到 pt_regs.regs[0]，用于重启系统调用*/
1:  sw   v0, PT_R2(sp)          /*返回值保存至 pt_regs.regs[2]*/

o32_syscall_exit:
    j    syscall_exit_partial     /*syscall 异常返回*/
    ...
    END(handle_sys)
```

syscall 异常处理程序首先保存进程用户态上下文信息至进程栈空间（pt_regs），设置 CPU 为内核态，开中断，然后根据 v0 传递的系统调用编号查找系统调用实现函数指针列表，调用相应的实现函数，并将函数返回值保存至 pt_regs.regs[2]（v0）（异常返回时从中恢复寄存器），最后跳转至 syscall_exit_partial 处从异常处理程序返回用户空间。

这里需要说明一下的是库函数与系统调用实现函数之间的参数传递，根据 MIPS 体系结构标准，函数调用的前几个参数通过 a0、a1、a2 等寄存器传递，更多的参数则通过栈传递。由于在 syscall 异常前后，a0、a1、a2 等寄存器并没有变化，因此在系统调用实现函数中可以直接使用。保存在栈中的参数，由于 syscall 异常前使用的是进程用户态栈，异常后使用的是进程内核态栈，因此在调用实现函数前，需要将用户态栈的参数复制到内核态栈。

6.3.4 syscall 异常返回

syscall 异常处理程序在系统调用实现函数返回后，运行至 syscall_exit_partial 处，从异常返回。

syscall_exit_partial 标号定义在/arch/mips/kernel/entry.S 文件内，代码如下：

```
FEXPORT(syscall_exit_partial)
    local_irq_disable                   /*关本地中断*/
    LONG_L    a2, TI_FLAGS($28)     /*a2=thread_info.flags*/
    li    t0, _TIF_ALLWORK_MASK
    and  t0, a2                 /*检测是否有挂起工作*/
    beqz    t0, restore_partial /*没有挂起的工作，跳转至 restore_partial，恢复寄存器从异常返回*/

    SAVE_STATIC                     /*有挂起的的工作，保存其它寄存器至 pt_regs*/
syscall_exit_work:
    LONG_L  t0, PT_STATUS(sp)       /*保存系统调用前 CPU 状态寄存器至 t0*/
    andi   t0, t0, KU_USER
    beqz    t0, resume_kernel     /*系统调用前为内核态（正常应为用户态），跳转至 resume_kernel*/
    li     t0, _TIF_WORK_SYSCALL_EXIT    /*此标记的工作在这里不处理（屏蔽）*/
    and  t0, a2                 /*相与*/
    beqz    t0, work_pending      /*为 0，表示有_TIF_WORK_SYSCALL_EXIT 标记之外的工作，
                                  *跳转至处理挂起工作，见上文。*/
    local_irq_enable          /*开本地中断*/
```

```
    move    a0, sp
    jal    syscall_trace_leave
    b      resume_userspace      /*返回用户空间*/
```

_TIF_ALLWORK_MASK 和_TIF_WORK_SYSCALL_EXIT 宏定义如下：

```
/*/arch/mips/include/asm/thread_info.h*/
#define _TIF_WORK_MASK              \
                (_TIF_SIGPENDING | _TIF_NEED_RESCHED | _TIF_NOTIFY_RESUME)
#define _TIF_ALLWORK_MASK    (_TIF_NOHZ | _TIF_WORK_MASK |          \
                        _TIF_WORK_SYSCALL_EXIT | _TIF_SYSCALL_TRACEPOINT)
#define _TIF_WORK_SYSCALL_EXIT  (_TIF_NOHZ | _TIF_SYSCALL_TRACE | \
                                _TIF_SYSCALL_AUDIT | _TIF_SYSCALL_TRACEPOINT)
```

syscall_exit_partial 标号处程序先检测 thread_info.flags 是不是有_TIF_ALLWORK_MASK 标记的工作，没有则恢复寄存器，从异常返回。如果有则继续往下执行，屏蔽掉_TIF_WORK_SYSCALL_EXIT 标记的工作后，看 thread_info.flags 中是否还有标记挂起的工作，有则跳转至 work_pending 处执行，否则恢复寄存器，从 syscall 异常返回。

6.3.2 添加系统调用

在熟悉了内核系统调用的机制后，用户甚至可以在内核中添加自己定义的系统调用。下面以增加名称为 mysyscall 的系统调用为例，简要介绍向内核添加系统调用的方法。

1、修改头文件：在/arch/mips/include/uapi/asm/unistd.h 头文件末尾添加 mysyscall 系统调用的编号，并增加系统调用总数。

```
#if _MIPS_SIM == _MIPS_SIM_ABI32     /*纯粹 32 位处理器，非 64 位处理器*/
    ...
    #define   __NR_memfd_create          (__NR_Linux + 354)
    #define   __NR_bpf                   (__NR_Linux + 355)
    #define   __NR_execveat              (__NR_Linux + 356)
    #define   __NR_mysyscall             (__NR_Linux + 357)

    #define   __NR_Linux_syscalls        357          /*系统调用总数加 1*/
#endif    /* _MIPS_SIM == _MIPS_SIM_ABI32 */

#define   __NR_O32_Linux             4000        /*系统调用起始编号*/
#define   __NR_O32_Linux_syscalls    357
```

在声明系统调用实现函数的/include/linux/syscalls.h 头文件内，添加系统调用实现函数的声明：

```
asmlinkage long sys_mysyscall(arg0,arg1,...);
```

2、定义系统调用实现函数：在内核源代码中用 SYSCALL_DEFINEX(mysyscall, ...)宏或直接定义函数 sys_mysyscall()，函数内实现系统调用需要完成的工作。

3、在/arch/mips/kernel/scall32-o32.S 文件内 sys_call_table 列表的末尾添加 sys_mysyscall 函数指针：

```
    .align      2
    .typesys_call_table, @object
EXPORT(sys_call_table)
    PTR   sys_syscall               /* 4000 */
    PTR    sys_exit
    ...
    PTR    sys_bpf                  /* 4355 */
    PTR   sys_execveat
    PTR    sys_mysyscall
```

重新构建内核并运行后，用户程序即可通过 syscall(__NR_mysyscall,arg0,arg1,...)函数执行 mysyscall 系统调用。

6.4 中断

中断通常由系统外部设备触发，将打断程序的正常执行。CPU 响应中断需要满足一定的条件，程序也可以控制 CPU 是响应中断还是屏蔽中断。

MIPS 体系结构将中断视为异常的一种。中断处理有两种模式一种是兼容模式，即中断处理程序入口地址就是普通异常的入口地址，二是向量模式，即中断有自己专门的处理程序入口地址。本节主要以兼容模式为例介绍中断处理的实现。

内核中通过中断描述符（irq_desc 结构体）来管理每一个中断，中断描述符中记录了中断的一些硬件属性，如怎样打开/关闭某个硬件中断等，这些由平台（处理器）相关代码设置。中断描述符中还记录了中断的软件属性，即中断处理函数等，这由设备驱动程序注册。

中断处理程序首先保存 CPU 寄存器信息，关中断，然后确定硬件中断对应内核中哪个中断描述符，调用描述符中指定的通用函数，通用函数调用设备驱动程序注册的中断处理函数，处理完成后恢复 CPU 寄存器（含开中断），中断返回。

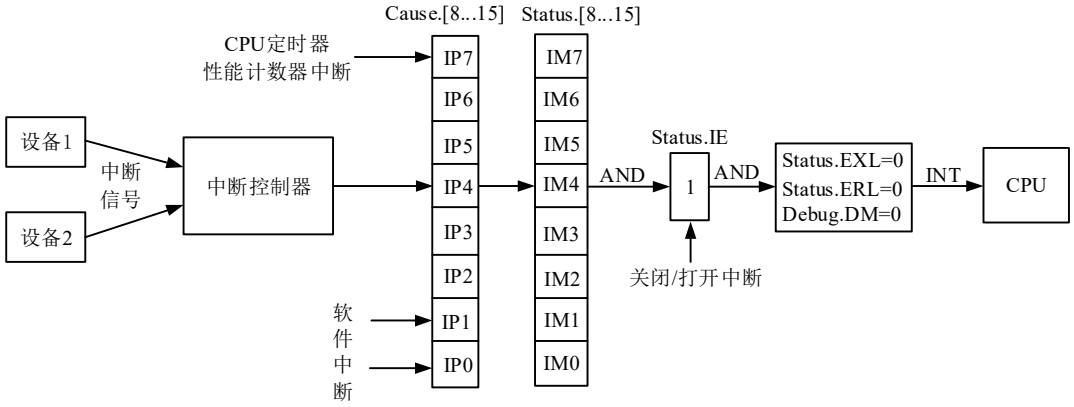
6.4.1 概述

本小节首先介绍 MIPS 处理器中断产生的条件（使 CPU 响应中断），然后介绍内核关闭/打开中断的接口函数，最后概述内核对中断的管理，以及中断处理流程。后面小节将详细介绍内核中断管理和中断处理程序。

1 中断产生

下图示意了 MIPS 处理器中断产生的条件。MIPS 处理器支持 8 个外部中断源，含 2 个软件中断和 6 个硬件中断。Cause 寄存器[8...15]表示有没有等待（挂起）的中断，IP0 和 IP1 表示软件中断，程序通过写这 2 个位来触发软件中断，IP2-IP7 可接硬件中断源，IP7 复用了 CPU 的定时器中断和性能计数器中断。IP2-IP7 位只读，不可写，由硬件控制。

通常一个硬件中断源可被多个外部设备共用，如下图所示，设备 1 和设备 2 通过中断控制器接在 IP4 中断源上，也就是说设备 1 和设备 2 通过中断控制器共用 IP4 中断源。



CPU 状态寄存器 Status.[8...15]表示是否使能（不屏蔽）对应的中断源，1 表示使能（不屏蔽），0 表示禁止（屏蔽）。Status.IE 是所有中断源的总开关，1 表示打开所有中断，0 表示关闭所有中断，内核代码中关闭/打开本地（当前 CPU 核）中断的操作就是通过操作这个标记位来实现的。

要使 CPU 响应外部中断，不仅要有等待的中断源（Cause.[8...15]中有置 1 的位），Status.[8...15]中没有屏蔽相应的中断，总开关 Status.IE 要打开，还要求 CPU 不能处于异常级别、错误级别或调试模式，如此 CPU 才会响应中断。

这里要说明一下的是异常级别，发生异常时 Status.EXL 位会置为 1（同时 CPU 进入内核态），在异常处理程序中没有清零此位之前中断都是关闭的，执行异常返回指令 eret 会清 0 此位。

如果要在异常处理程序内开中断，在保存 Status 值（KSU=0b10 用户态，EXL=1，因异常从用户态进入内核态）后，需重新设置当前状态 Status 值（KSU=0b00 内核态，EXL=0，IE=1）。异常返回前恢复刚进入异常时的 Status 值，执行 eret 指令，从异常返回（返回用户态，EXL=0）。

2 内核中断控制

内核代码中经常需要关闭、打开本地 CPU 核的中断。对于 MIPS 体系结构，关闭本地中断就是对 CPU 状态寄存器 Status.IE 位写 0，打开本地中断就是对 Status.IE 位写 1。

内核提供了关闭/打开中断的接口函数，显然这些是体系结构相关的函数，例如：

- **local_irq_enable()**：打开本地中断（当前 CPU 核）。
- **local_irq_disable()**：关闭本地中断。
- **local_irq_save(flags)**：将当前中断状态（Status.IE）保存至整型数 flags，然后关闭中断。
- **local_irq_restore(flags)**：恢复整型数 flags 中保存的中断状态。
- **local_save_flags(flags)**：保存当前中断状态至整型数 flags。

以上函数声明在/include/linux/irqflags.h 头文件，函数内调用在/arch/mips/include/asm/irqflags.h 头文件内定义的体系结构相关的实现函数。

3 中断处理

MIPS 中断处理具有两种模式，一种是兼容模式，中断由普通异常处理程序 except_vec3_generic 调用 handle_int()函数处理；另一种是向量模式（含外部中断控制器模式），中断具有独立的处理程序入口地址，基地址为 EBase+0x200。在基地址之上每个中断具有自己的中断处理程序入口地址（中断向量）。

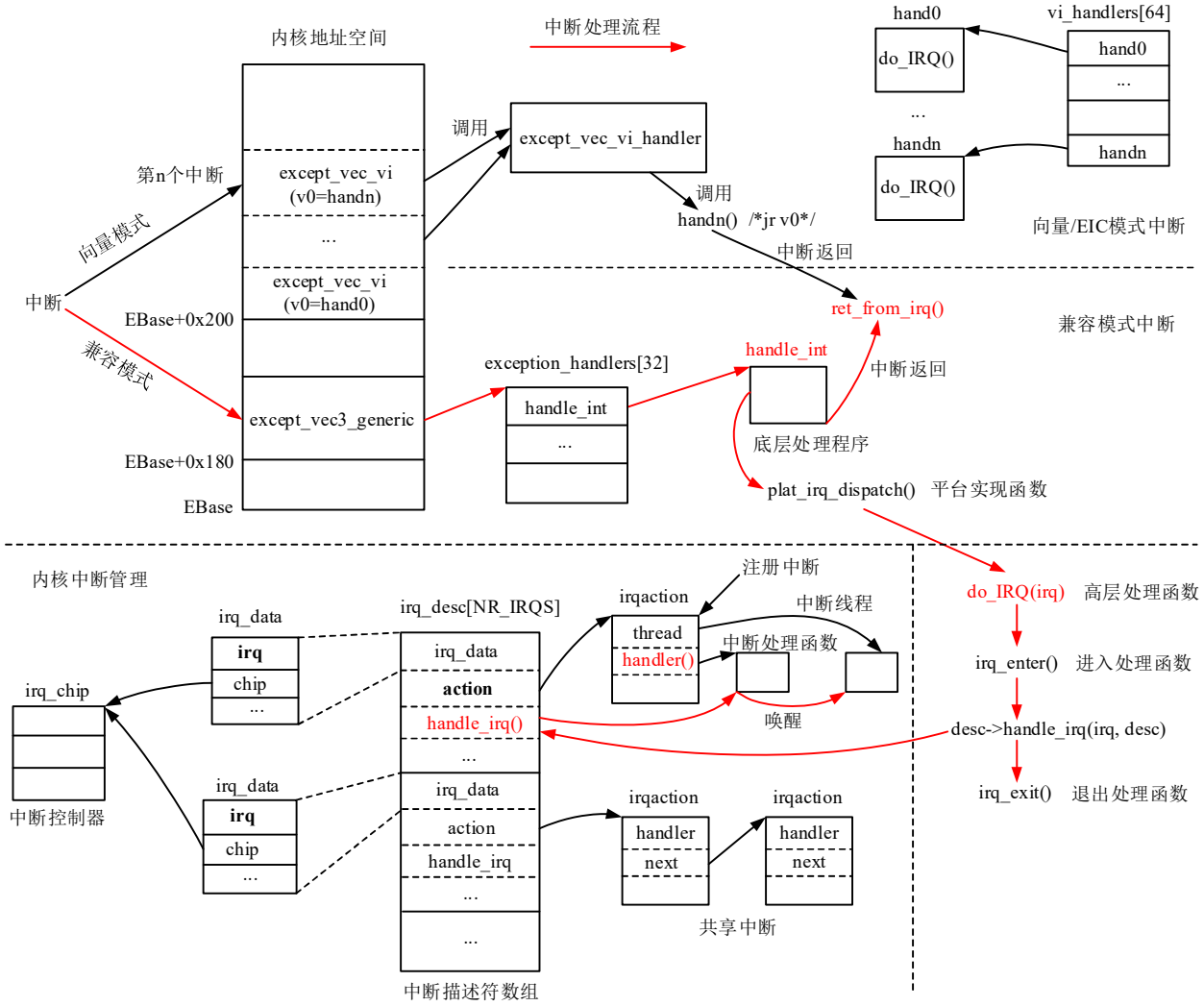
MIPS 处理器中断处理流程如下图所示。兼容模式中断处理程序由 except_vec3_generic 普通异常处理程序跳转至 handle_int 处代码处理。

handle_int 处理程序先保存 CPU 寄存器信息，设置内核态关中断，然后调用板级（平台）代码定义的

中断处理函数 plat_irq_dispatch()。plat_irq_dispatch()函数内检测硬件中断由哪个设备产生，并将硬件中断编号转为内核内部的中断编号 irq，然后调用通用的高层中断处理函数 do_IRQ(irq)。do_IRQ(irq)函数根据中断编号，找到对应的中断描述符，调用中断描述符中的通用函数，最终调用设备驱动程序注册的中断处理函数，并唤醒中断线程（如果需要）。高层函数 do_IRQ(irq)返回后，handle_int 恢复寄存器，中断返回。

在这里 handle_int 处理程序称它为底层中断处理程序，do_IRQ(irq)称它为高层中断处理函数，中断描述符中的处理函数称它为通用中断处理函数。

向量模式中断（含外部控制器模式），中断向量基地址为 EBase+0x200，每个中断向量包含若干字节的空间。内核定义了 vi_handlers[]数组保存平台定义的各中断处理函数，类似兼容模式的 plat_irq_dispatch()函数。接口函数 set_vi_handler(n, handn)负责将 handn 函数指针赋予 vi_handlers[n]数组项，将 except_vec_vi()函数体复制到 n 对应的中断向量内，并修改其中的 2 条指令，这 2 条指令的功能是将 handn 值写入 v0 寄存器。except_vec_vi()函数将会跳转至 except_vec_vi_handler()函数内执行，后者将执行 jr v0 指令，即调用 handn()函数，最后中断返回。这里的 handn()函数内也将调用 do_IRQ(irq)函数。



内核中每个中断由中断描述符 irq_desc 结构体表示，内核静态定义了 irq_desc[NR_IRQS]中断描述符实例数组（没有选择 SPARSE_IRQ 配置选项），数组项数 NR_IRQS 由平台相关代码定义。irq_chip 结构体表示中断控制器，结构体中主要包含操作中断的函数指针，如：关闭/打开某个中断等，它是一个需要由平台相关代码实现的数据结构。

高层中断处理函数 do_IRQ(irq)调用中断描述符 irq_desc 结构体 handle_irq 成员指示的函数，这是通用中断处理函数（同一中断控制器），此函数进而调用中断描述符关联中断响应 irqaction 实例中的 handler()

函数，并唤醒中断线程（如果需要）。

注册中断时，会创建中断响应 irqaction 结构体实例，其中 handler 函数指针成员指向驱动程序注册的中断处理函数，thread 指向创建的中断处理线程（内核线程，如果需要）。注册中断时传递的 thread_fn()函数就是在中断处理线程中将调用的函数。

在 handler()函数中将执行中断处理中上半部工作，在中断线程中执行下半部工作（thread_fn()函数）。另外，除了中断处理线程，下半部工作还可以由软中断、工作队列等完成，详见本章下文。

这里还要说明一下，在 handler()函数中不能进入睡眠，因为此函数是在关中断的状态下调用的。进程睡眠后会发生进程调度，调度运行下一个进程后，中断将继续保持关闭状态，此时周期性调度器失效（没有周期时钟中断了），如果下一进程不打开中断，也不主动放弃 CPU，系统将不再能执行进程调度。

6.4.2 底层处理程序

MIPS 处理器中断处理程序根据模式的不同，具有不同的入口地址（异常向量）。兼容模式中断其处理程序入口地址为 EBase+0x180，即与普通异常共用处理程序。向量模式（含外部中断控制器模式）中断具有专门的处理程序入口地址，并且每个中断都具有独立的入口地址（中断向量）。

中断模式由 CPU 协处理器 0 中相关寄存器确定，请读者查阅 MIPS 体系结构手册及具体处理器手册，因为有的处理器可能不支持向量模式。

本小节介绍两种模式下中断处理程序的实现。

1 兼容模式中断处理

MIPS 处理器兼容模式中断处理程序入口为 except_vec3_generic，它根据中断异常的编号（0x00），跳转至 handle_int 处执行中断处理。

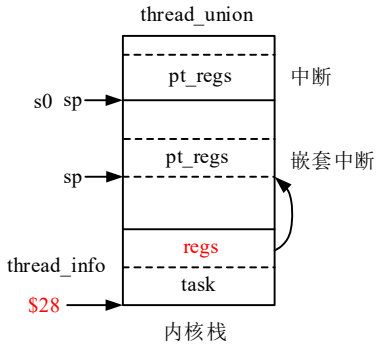
handle_int()函数定义在/arch/mips/kernel/genex.S 文件内，代码如下：

```
    .align 5
    BUILD_ROLLBACK_PROLOGUE handle_int
    NESTED(handle_int, PT_SIZE, sp)
#ifdef CONFIG_TRACE_IRQFLAGS
     ...
#endif              /*CONFIG_TRACE_IRQFLAGS 结束*/

     SAVE_ALL     /*保存 CPU 寄存器信息至内核栈 pt_regs 实例*/
     CLI          /*确保进入核心态，关闭中断，/arch/mips/include/asm/stackframe.h*/
                  /*Status：CU0=1，KSU=00（内核态），EXL=0，IE=0（关中断）*/

     TRACE_IRQS_OFF

     LONG_L   s0, TI_REGS($28)     /*thread_info.regs 值写入 s0（初始化时为 0）*/
     LONG_S   sp, TI_REGS($28)     /*保存当前 pt_regs 实例地址至 thread_info.regs*/
     PTR_LA   ra, ret_from_irq          /*plat_irq_dispatch()函数返回后，执行中断返回*/
     PTR_LA   v0, plat_irq_dispatch     /*平台定义的中断处理函数*/
     jr   v0              /*调用 plat_irq_dispatch()函数，返回地址为 ret_from_irq*/
    #ifdef CONFIG_CPU_MICROMIPS
```

```
    nop
#endif
    END(handle_int)
```

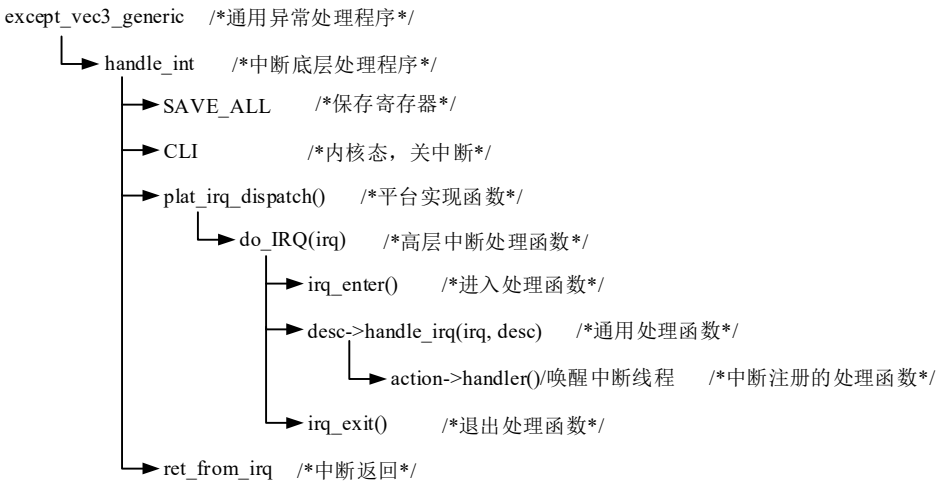
handle_int()函数比较简单，首先是在内核栈中预留 pt_regs 实例（sp 指向实例），保存 CPU 寄存器信息至此实例（SAVE_ALL 宏代码实现），修改状态寄存器使 CPU 处于内核态（KSU=0b00，EXL=0）并关中断（IE=0），然后保存原 thread_info.regs 值至 s0，保存当前 pt_regs 实例地址（sp 值）至 thread_info.regs，调用平台定义的 plat_irq_dispatch()函数处理中断，函数返回 ret_from_irq 处，中断返回。

如果存在中断嵌套，thread_info.regs 指向最近一次中断处理程序使用的 pt_regs 实例，中断返回前恢复为上一次中断处理程序使用的 pt_regs 实例地址，如下图所示。



■中断处理流程

下面先整体上来看一下中断处理流程，函数调用关系如下图所示，具体函数定义后面再介绍：



平台定义的 plat_irq_dispatch()函数需要确定中断是由谁产生的，找到对应的内核中断编号 irq，依此调用高层中断处理函数 do_IRQ(irq)，函数返回后跳转至 ret_from_irq 处，中断返回。

do_IRQ(irq)首先调用 irq_enter()函数表示 CPU 进入了中断处理中，主要是增加进程 preempt_count 抢占计数中 HARDIRQ 位域值等；然后调用 irq 对应中断描述符中的 handle_irq()函数，这个函数进而调用驱动程序注册中断时注册的中断处理函数，并唤醒中断线程（如果有的话）；最后调用 irq_exit()函数表示 CPU 退出中断处理，主要工作是减小 HARDIRQ 位域值，执行软中断（见下节）等。

为了尽量少产生中断嵌套，中断处理程序通常是在关中断的状态下执行，因此中断处理程序应当尽快地执行完，使 CPU 退出中断状态（开中断，以响应其它的中断）。

为此，驱动程序通常将中断处理工作划分成上半部和下半部，上半部是比较紧急需要马上处理的工作，

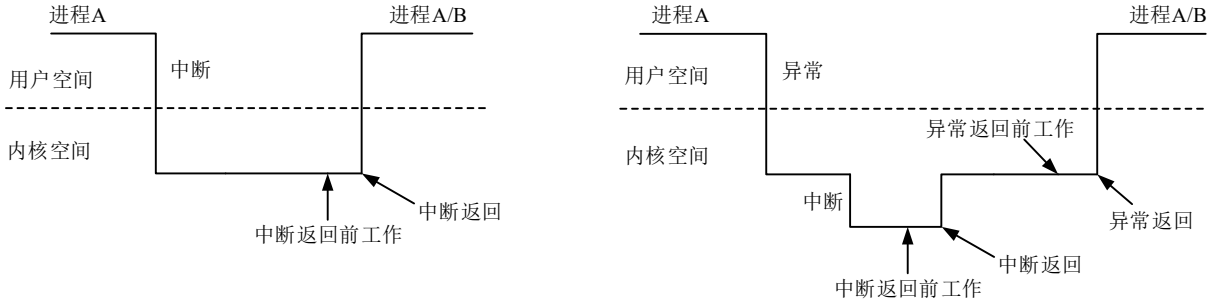
在中断处理程序中执行（关中断），不那么紧急的工作放在下半部中进行（退出中断处理后进行，开中断）。注册中断时可同时传入中断处理函数和中断线程中的执行函数，注册时将创建中断线程。中断处理函数中执行的是上半部的工作，中断线程中的执行函数执行的是下半部的工作。中断处理中执行完中断处理函数后，只需唤醒中断线程即可退出中断状态，由中断线程完成下半部工作。

除中断线程外，常见的下半部机制还有软中断、tasklet（不推荐使用）、工作队列等，本章后面将详细介绍。

中断处理程序最后跳转至 ret_from_irq 处从中断返回。

■中断返回

中断可能在发生在 CPU 用户态，也可能发生在内核态。因此中断返回可能返回用户空间，也可能返回内核空间，如下图所示：



底层中断处理程序在 ret_from_irq 处执行中断返回，代码如下（/arch/mips/kernel/entry.S）

```
FEXPORT(ret_from_irq)
    LONG_S  s0, TI_REGS($28) /*恢复原 thread_info.regs 值，上一次中断使用的 pt_regs 实例地址*/
FEXPORT(__ret_from_irq)
resume_userspace_check:                 /*判断是返回用户空间还是内核空间*/
    LONG_L  t0, PT_STATUS(sp)           /*刚进中断时的状态寄存器值（EXL=1）*/
    andi    t0, t0, KU_USER             /*KSU 为 0b00 表示内核态，0b10 表示用户态*/
    beqz    t0, resume_kernel           /*返回内核空间，跳转至 resume_kernel*/

resume_userspace:                       /*返回用户空间执行以下代码*/
    local_irq_disable                   /*关本地中断*/
    ...
    LONG_L    a2, TI_FLAGS($28)         /*进程是否有挂起的工作*/
    andi      t0, a2, _TIF_WORK_MASK
    bnez      t0, work_pending          / *处理挂起的工作，见本章上文，/arch/mips/kernel/entry.S*/
    j       restore_all                 /*没有挂起的工作，恢复寄存器，中断返回*/
```

中断返回前执行的代码与前面介绍的执行异常返回前工作的代码相同，不再讲解了。

2 向量模式中断处理

MIPS 处理器向量模式中断中，每个中断具有独立的处理程序入口地址，EBase+0x200 为中断向量基地址，每个中断向量占用若干字节空间（由 CP0 寄存器确定）。假设每个中断向量占用字节数为 size，则第 n 个中断向量地址为 EBase+0x200+n*size。

内核在/arch/mips/kernel/traps.c 文件内定义了整型数组，用于保存最多 64 个中断的处理函数入口地址：

unsigned long vi_handlers[64];

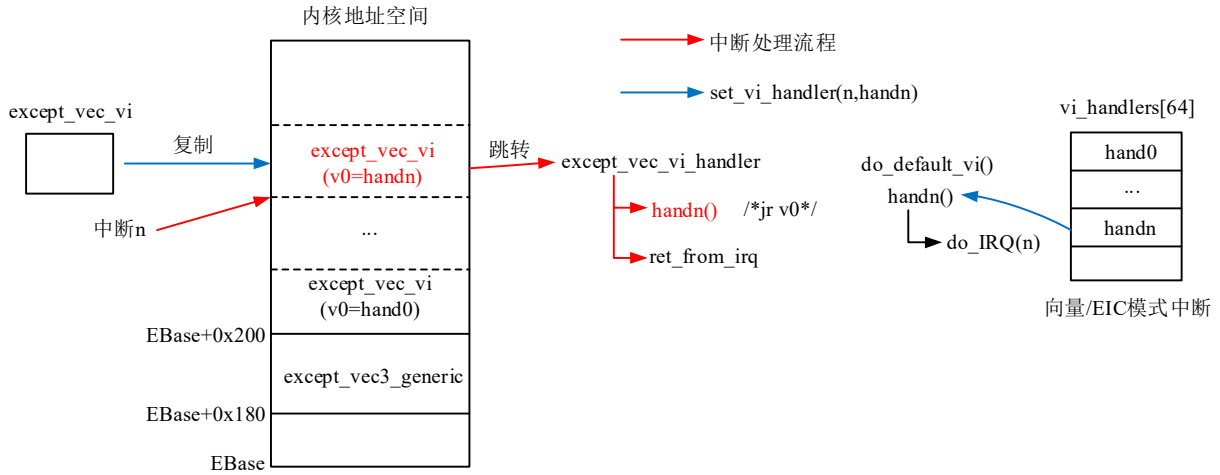
数组项中保存的是 vi_handler_t 类型函数指针，定义如下（/arch/mips/include/asm/setup.h）：

typedef void (*vi_handler_t)(void);

vi_handlers[]数组为什么是 64 项呢？前面介绍了 MIPS 处理器支持 8 个外部中断源，有 2 个是软件中断源，6 个是硬件中断源。在向量模式下，6 个硬件中断源被视为一个整体（位域），不再是单个的中断源了，6 个中断源表示 6 个二进制位，可表示 2^6=64 个数，即硬件中断编号，因此 vi_handlers[]数组是 64 项。

向量中断处理流程如下图所示，中断向量中写入的是 except_vec_vi 处代码，但在写入后会修改 2 条指令，这 2 条指令的作用是将中断处理函数 handn()地址写入 v0 寄存器，因为每个中断向量的 handn()函数可能不同，因此需要修改指令。

中断产生后，CPU 根据硬件中断号从相应中断向量处取指执行，即执行 except_vec_vi 代码，此代码最后跳转至 except_vec_vi_handler 处，此处在保存寄存器后调用 v0 保存的函数，即 handn()函数，最后函数返回至 ret_from_irq 处，从中断返回。



set_vi_handler(int **n**, vi_handler_t **addr**)函数将 addr 表示的中断处理函数地址（类似 plat_irq_dispatch()函数）写入 vi_handlers[n]数组项，复制 except_vec_vi 处代码至中断向量，并修改其中 2 条指令，这 2 条指令将 addr 值写入 v0 寄存器（即中断处理函数入口地址）。

内核在初始化异常向量的 trap_init()函数中将初始化向量模式的中断向量（如果处理器支持），代码如下：

```
void __init trap_init(void)
{
    ...
    if (cpu_has_veic || cpu_has_vint) {        /*初始化中断向量*/
        int nvec = cpu_has_veic ? 64 : 8;
        for (i = 0; i < nvec; i++)
            set_vi_handler(i, NULL);   /*填充各中断向量，默认处理函数为 do_default_vi()*/
    }
    ...
}
```

初始化函数中调用 set_vi_handler(i, NULL)函数，这里中断处理函数指针为 NULL，内核会默认将中断处理函数设为 **do_default_vi()**，此函数内只是输出信息，什么也不做。

平台（处理器）相关代码可再次调用 set_vi_handler(int **n**, vi_handler_t **addr**)函数为中断设置中断处理函数。

下面看一下 except_vec_vi 和 except_vec_vi_handler 标号处代码，如下所示（/arch/mips/kernel/genex.S）：

```
NESTED(except_vec_vi, 0, sp)
    SAVE_SOME       /*保存寄存器*/
    SAVE_AT
    .set  push
    .set  noreorder
    PTR_LA v1, except_vec_vi_handler
FEXPORT(except_vec_vi_lui)
    lui   v0, 0           /*set_vi_handler()函数会修改标红的指令，将中断处理函数地址写入 v0*/
    jr    v1              /*跳转至 except_vec_vi_handler 处执行*/
FEXPORT(except_vec_vi_ori)
    ori   v0, 0
    .set  pop
    END(except_vec_vi)
EXPORT(except_vec_vi_end)

NESTED(except_vec_vi_handler, 0, sp)          /*v0 保存了中断处理函数地址*/
    SAVE_TEMP       /*保存寄存器*/
    SAVE_STATIC
    CLI             /*内核态，关中断*/
#ifdef CONFIG_TRACE_IRQFLAGS
    ...
#endif
    LONG_L s0, TI_REGS($28)       /*thread_info.regs 值写入 s0*/
    LONG_S sp, TI_REGS($28)       /*sp 值写入 thread_info.regs，pt_regs 实例地址*/
    PTR_LA ra, ret_from_irq       /*中断返回地址写入 ra*/
    jr    v0                      /*调用 v0 保存的函数，即中断处理函数 handn()*/
                                  /*函数返回地址为 ret_from_irq，从中断返回*/
    END(except_vec_vi_handler)
```

6.4.3 内核中断管理

在底层中断处理程序中，确定了中断的内核编号后，将由平台实现的函数调用 do_IRQ(irq)函数进行高层的中断处理，主要是根据中断编号找到中断描述符，调用其中的处理函数。

内核对每个中断都赋予一个编号，这其实是由平台代码确定的。内核中断编号并不一定和硬件中断编号相同，硬件中断编号与其在内核中的编号有一个映射关系，由平台代码确定。

每个中断编号在内核中对应一个中断描述符，表示了中断的软硬件信息和状态，硬件信息由平台代码注册，软件信息就是驱动程序注册的中断处理函数或中断线程等。

本小节主要介绍内核中断编号、中断描述符等数据结构的管理和初始化。

1 数据结构

内核中断管理相关的数据结构主要有中断描述符 irq_desc、中断控制器 irq_chip 和中断响应 irqaction 结构体等。irq_desc 结构体表示中断的整体信息，关联 irq_chip 和 irqaction 结构体。irq_chip 表示中断控制器的信息，如怎么打开/关闭一个中断等。irqaction 结构体表示发生该中断后的动作（中断响应），主要包含驱动程序注册的中断处理函数和中断线程等。

内核还定义了中断域 irq_domain 结构体，它用于实现硬件中断编号至内核中断编号的转换，需要内核配置支持设备树，本节暂不涉及中断域。

■中断描述符

每个中断编号在内核中对应一个中断描述符 irq_desc 结构体实例，用于表示中断的整体信息，irq_desc 结构体定义在/include/linux/irqdesc.h 头文件，简列如下：

```
struct irq_desc {
    struct irq_common_data   irq_common_data;     /*中断数据，标记传递给中断控制器的信息*/
    struct irq_data      irq_data;        /*中断数据结构*/
    unsigned int __percpu   *kstat_irqs;     /*irq 状态，percpu 变量*/
    irq_flow_handler_t handle_irq;          /*通用中断处理函数*/
#ifdef CONFIG_IRQ_PREFLOW_FASTEOI
    irq_preflow_handler_t   preflow_handler;
#endif
    struct irqaction     *action;           /*中断响应（如果是共享中断，则为 irqaction 实例链表）*/
    unsigned int         status_use_accessors;      /*状态信息*/
    unsigned int         core_internal_state__do_not_mess_with_it;  /*别名 istate，内部状态信息*/
    unsigned int         depth;          /*禁止深度，irq_disable()函数使用*/
    unsigned int         wake_depth;   /*使能深度*/
    unsigned int         irq_count;      /* For detecting broken IRQs */
    unsigned long        last_unhandled;     /* Aging timer for unhandled count */
    unsigned int         irqs_unhandled;
    atomic_t         threads_handled;
    int              threads_handled_last;
    raw_spinlock_t           lock;
    struct cpumask           *percpu_enabled;    /*CPU 核亲和性，中断能由哪些 CPU 核响应*/
#ifdef CONFIG_SMP
    const struct cpumask     *affinity_hint;        /*中断的 CPU 亲和性*/
    struct irq_affinity_notify *affinity_notify;
  #ifdef CONFIG_GENERIC_PENDING_IRQ
    cpumask_var_t pending_mask;
  #endif
#endif
    unsigned long   threads_oneshot;
```

```
                                    /*位图，共享 ONESHOT 中断，标记正在处理中断的 irqaction 实现*/
    atomic_t    threads_active;         /*共享 ONESHOT 中断，当前还在运行中断线程数量*/
    wait_queue_head_t     wait_for_threads;         /*等待本中断所有中断线程执行完的进程*/
#ifdef CONFIG_PM_SLEEP
    unsigned int    nr_actions;         /*action 链表中实例数量*/
    unsigned int    no_suspend_depth;
    unsigned int    cond_suspend_depth;
    unsigned int    force_resume_depth;
#endif
#ifdef CONFIG_PROC_FS
    struct proc_dir_entry    *dir;
#endif
    int    parent_irq;
    struct module  *owner;
    const char    *name;            /*名称*/
} ____cacheline_internodealigned_in_smp;
```

irq_desc 结构体主要成员简介如下：

●**irq_data**：irq_data 结构体成员，结构体定义在/include/linux/irq.h 头文件内，主要表示中断物理信息：

```
struct irq_data {
    u32                mask;
    unsigned int           irq;          /*内核中断号*/
    unsigned long          hwirq;     /*硬件中断号*/
    unsigned int           node;
    struct irq_common_data    *common;       /*指向 irq_desc.irq_common_data 结构体成员*/
    struct irq_chip *chip;                 /*中断控制器结构体指针*/
    struct irq_domain    *domain;     /*硬件中断号与内核中断号转换*/
#ifdef     CONFIG_IRQ_DOMAIN_HIERARCHY
    struct irq_data        *parent_data;
#endif
    void               *handler_data;
    void               *chip_data;          /*中断控制器私有数据*/
    struct msi_desc        *msi_desc;
    cpumask_var_t          affinity;         /*CPU 亲和性*/
};
```

●**irq_common_data**：irq_common_data 结构体成员，结构体定义在/include/linux/irq.h 头文件内：

```
struct irq_common_data {
    unsigned int    state_use_accessors;        /*中断控制器共享的信息（物理信息），标记位*/
};
```

state_use_accessors 成员表示中断对于不同中断控制器共享的信息，传递给中断控制器中的函数，通常用于描述中断底层的状态，取值如下（/include/linux/irq.h）：

```
enum {
    IRQD_TRIGGER_MASK          = 0xf,       /*低 4 位触发类型掩码*/
    IRQD_SETAFFINITY_PENDING   = (1 << 8),     /*设置 CPU 亲和性操作正被挂起*/
    IRQD_NO_BALANCING          = (1 << 10),    /*禁止中为在 CPU 间平衡*/
    IRQD_PER_CPU               = (1 << 11),      /*percpu 中断*/
    IRQD_AFFINITY_SET          = (1 << 12),      /*已经设置了 CPU 亲和性*/
    IRQD_LEVEL                 = (1 << 13),      /*电平触发*/
    IRQD_WAKEUP_STATE          = (1 << 14),    /*中断被配置成唤醒系统的中断*/
    IRQD_MOVE_PCNTXT           = (1 << 15),    /*中断可在进程上下文被移动？？*/
    IRQD_IRQ_DISABLED          = (1 << 16),    /*中断处于关闭状态，初始化时设置此位*/
    IRQD_IRQ_MASKED            = (1 << 17),    /*中断被屏蔽*/
    IRQD_IRQ_INPROGRESS        = (1 << 18),    /*正在处理中断*/
    IRQD_WAKEUP_ARMED          = (1 << 19),    /*Wakeup mode armed*/
};
```

前面 irq_data 结构体成员的 common 成员指向这里的 irq_common_data 成员（初始化中断描述符时设置），因此内核通过 irq_data 结构体成员访问 state_use_accessors 值，如下所示：

```
#define   __irqd_to_state(d)      ((d)->common->state_use_accessors)    /*d 指向 irq_data 实例*/
```

●status_use_accessors：中断触发方式，中断属性等信息，取值定义如下（/include/linux/irq.h）：

```
enum {
    IRQ_TYPE_NONE      = 0x00000000,
    IRQ_TYPE_EDGE_RISING = 0x00000001,
    IRQ_TYPE_EDGE_FALLING    = 0x00000002,
    IRQ_TYPE_EDGE_BOTH   = (IRQ_TYPE_EDGE_FALLING | IRQ_TYPE_EDGE_RISING),
    IRQ_TYPE_LEVEL_HIGH  = 0x00000004,
    IRQ_TYPE_LEVEL_LOW   = 0x00000008,
    IRQ_TYPE_LEVEL_MASK = (IRQ_TYPE_LEVEL_LOW | IRQ_TYPE_LEVEL_HIGH),
    IRQ_TYPE_SENSE_MASK = 0x0000000f,
    IRQ_TYPE_DEFAULT = IRQ_TYPE_SENSE_MASK,       /*低 4 位表示中断触发类型*/
    IRQ_TYPE_PROBE             = 0x00000010,
                                          /*低 8 位标记同 irqaction.flags 成员低 8 位*/
    IRQ_LEVEL        = (1 << 8),
    IRQ_PER_CPU          = (1 << 9),     /*percpu 中断*/
    IRQ_NOPROBE          = (1 << 10),
    IRQ_NOREQUEST      = (1 << 11),      /*系统预留的中断，外设不可用*/
    IRQ_NOAUTOEN       = (1 << 12),      /*置位表示注册中断时不使能中断，否则使能*/
    IRQ_NO_BALANCING       = (1 << 13),    /*不可负载均衡*/
    IRQ_MOVE_PCNTXT        = (1 << 14),
    IRQ_NESTED_THREAD    = (1 << 15),    /*嵌套中断*/
    IRQ_NOTHREAD       = (1 << 16),      /*中断处理不可线程化*/
    IRQ_PER_CPU_DEVID   = (1 << 17),    /*预留给 IRQF_PERCPU 类型的中断*/
```

```
    IRQ_IS_POLLED        = (1 << 18),      /*轮询*/
};
```

内核在/kernel/irq/settings.h 头文件定义了设置/清零/检测 status_use_accessors 成员中标记位的函数，例如：

```
static inline bool irq_settings_is_level(struct irq_desc *desc)   /*检测 IRQ_LEVEL 标记位*/
static inline void irq_settings_clr_level(struct irq_desc *desc)    /*清零 IRQ_LEVEL 标记位*/
static inline void irq_settings_set_level(struct irq_desc *desc)   /*设置 IRQ_LEVEL 标记位*/
```

初始化中断描述符时，status_use_accessors 成员初始化为全 0。

●**core_internal_state__do_not_mess_with_it**：在/kernel/irq/internals.h 头文件内定义别名为 **istate**，中断状态的内核内部表示，取值（标记位）如下：

```
enum {
    IRQS_AUTODETECT        = 0x00000001,        /*描述符处于自动侦测状态*/
    IRQS_SPURIOUS_DISABLED   = 0x00000002,    /*视为伪中断，并禁用*/
    IRQS_POLL_INPROGRESS= 0x00000008,    /*描述符处于轮询 action*/
    IRQS_ONESHOT         = 0x00000020,     /*只执行一次*/
    IRQS_REPLAY          = 0x00000040,     /*重新发一次中断*/
    IRQS_WAITING         = 0x00000080,     /*等待状态*/
    IRQS_PENDING         = 0x00000200,     /*中断被挂起，未执行高层处理函数*/
    IRQS_SUSPENDED     = 0x00000800,     /*中断被暂停*/
};
```

●**handle_irq**：irq_flow_handler_t 类型函数指针（/include/linux/irqhandler.h），此函数称它为通用中断处理函数，也被称为电流处理函数，由高层处理函数 do_IRQ(irq)调用。

```
typedef    void (*irq_flow_handler_t)(unsigned int irq, struct irq_desc *desc);
```

●**action**：中断响应 irqaction 结构体指针，实际指向 irqaction 实例链表（如果是共享中断），表示对中断的响应，其中包括中断处理函数，标记 flags 等成员，见下文。

■中断控制器

irq_chip 结构体是对中断控制器的描述。irq_chip 结构体中主要是一些函数指针，用于实现对中断的控制和操作，结构体实例由平台（处理器）相关代码实现，类似于设备驱动程序。

irq_chip 结构体定义如下（/include/linux/irq.h）：

```
struct irq_chip {
    const char      *name;                  /*中断控制器名称*/
    unsigned int    (*irq_startup)(struct irq_data *data); /*使能（初始化）中断，一般转向 irq_enable()*/
    void        (*irq_shutdown)(struct irq_data *data);   /*禁止中断，一般转向 disable()*/
    void        (*irq_enable)(struct irq_data *data);      /*使能中断*/
    void        (*irq_disable)(struct irq_data *data);     /*禁止中断*/
    void        (*irq_ack)(struct irq_data *data);         /*应答中断，清中断*/
```

```
    void        (*irq_mask)(struct irq_data *data);       /*屏蔽中断*/
    void        (*irq_mask_ack)(struct irq_data *data);   /*应答（清中断）并屏蔽中断*/
    void        (*irq_unmask)(struct irq_data *data);     /*撤销中断屏蔽*/
    void        (*irq_eoi)(struct irq_data *data);        /*发送 EOI 信号给中断控制器，表示中断处理完成*/
    int         (*irq_set_affinity)(struct irq_data *data, const struct cpumask *dest, bool force);
                                                          /*绑定中断至某个 CPU 核，SMP*/
    int         (*irq_retrigger)(struct irq_data *data);          /*重新向 CPU 发送中断信号*/
    int         (*irq_set_type)(struct irq_data *data, unsigned int flow_type);  /*设置中断触发类型*/
    int         (*irq_set_wake)(struct irq_data *data, unsigned int on);
                                                          /*使能/关闭中断在电源管理中的唤醒功能*/
    void        (*irq_bus_lock)(struct irq_data *data);           /*对低速总线上中断控制器操作上锁*/
    void        (*irq_bus_sync_unlock)(struct irq_data *data);    /*同步或解锁低速总线中断控制器*/
    void        (*irq_cpu_online)(struct irq_data *data);   /*中断源配置到另一个 CPU 核*/
    void        (*irq_cpu_offline)(struct irq_data *data);
    void        (*irq_suspend)(struct irq_data *data);     /*暂停*/
    void        (*irq_resume)(struct irq_data *data);      /*重启*/
    void        (*irq_pm_shutdown)(struct irq_data *data);   /*关闭控制器时调用*/
    void        (*irq_calc_mask)(struct irq_data *data)
    void        (*irq_print_chip)(struct irq_data *data, struct seq_file *p);
    int         (*irq_request_resources)(struct irq_data *data);       /*请求资源*/
    void        (*irq_release_resources)(struct irq_data *data);       /*释放资源*/
    void        (*irq_compose_msi_msg)(struct irq_data *data, struct msi_msg *msg);
    void        (*irq_write_msi_msg)(struct irq_data *data, struct msi_msg *msg);
    int         (*irq_get_irqchip_state)(struct irq_data *data, enum irqchip_irq_state which, bool *state);
                                                          /*返回中断内部状态（istate）*/
    int         (*irq_set_irqchip_state)(struct irq_data *data, enum irqchip_irq_state which, bool state);
                                                          /*设置中断内部状态（istate）*/
    int         (*irq_set_vcpu_affinity)(struct irq_data *data, void *vcpu_info);
    unsigned long   flags;       /*标记，/include/linux/irq.h*/
};
```

irq_chip 结构体标记成员取值定义如下：

```
enum {
    IRQCHIP_SET_TYPE_MASKED      = (1 << 0),     /*调用 chip.irq_set_type()之前被屏蔽*/
    IRQCHIP_EOI_IF_HANDLED       = (1 << 1),
    IRQCHIP_MASK_ON_SUSPEND      = (1 << 2),
    IRQCHIP_ONOFFLINE_ENABLED    = (1 << 3),
    IRQCHIP_SKIP_SET_WAKE     = (1 << 4),
    IRQCHIP_ONESHOT_SAFE      = (1 << 5), /*中断控制器不支持中断嵌套，只支持 ONESHOT*/
    IRQCHIP_EOI_THREADED      = (1 << 6),
};
```

中断控制器可视为外部设备，irq_chip 结构体实例由平台（处理器）相关代码实现，通常并不需要定

义结构体中的所有函数，只需要实现一部分。

■中断响应

irqaction 结构体表示对中断的响应，称它为中断响应描述符，结构体定义在/include/linux/interrupt.h 头文件：

```
struct irqaction {
    irq_handler_t  handler;      /*设备驱动程序注册的中断处理函数*/
    void   *dev_id;              /*设备指针，用于标识设备*/
    void __percpu   *percpu_dev_id;      /*用于标识设备，percpu 中断*/
    struct   irqaction    *next;         /*共享中断，指向下一个 irqaction 实例*/
    irq_handler_t         thread_fn;     /*中断处理线程内的执行函数*/
    struct task_struct    *thread;       /*指向中断处理线程 task_struct 实例*/
    unsigned int          irq;           /*内核中断号*/
    unsigned int          flags;         /*注册中断时传递的标记，/include/linux/interrupt.h*/
    unsigned long         thread_flags;      /*中断线程标记，如 IRQTF_FORCED_THREAD*/
    unsigned long         thread_mask;       /*共享中断标记位，设置描述符 threads_oneshot 位图*/
    const char            *name;         /*设备名称*/
    struct proc_dir_entry     *dir;      /*导出到 proc 文件系统*/
} ____cacheline_internodealigned_in_smp;
```

irqaction 结构体主要成员简介如下：

●**handler**：设备驱动程序注册的中断处理函数，函数类型定义如下（/include/linux/interrupt.h）：

```
typedef   irqreturn_t (*irq_handler_t)(int, void *);     /*第一个参数为中断号，第二个为设备数据指针*/
```

此函数返回值为枚举类型 irqreturn_t，定义在/include/linux/irqreturn.h 头文件：

```
enum irqreturn {
    IRQ_NONE          = (0 << 0),          /*没有产生中断*/
    IRQ_HANDLED       = (1 << 0),          /*中断已得到处理*/
    IRQ_WAKE_THREAD = (1 << 1),          /*需唤醒中断线程*/
};
typedef   enum   irqreturn   irqreturn_t;
```

●**thread**：指向中断处理线程 task_struct 实例。

●**thread_fn**：中断处理线程内要执行的函数。

●**thread_flags**：中断线程标记（内部使用标记），取值定义如下（/kernel/irq/internals.h）：

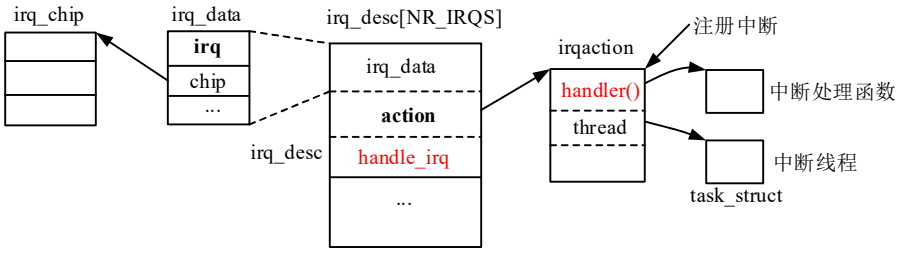
```
enum {
    IRQTF_RUNTHREAD,      /*应当运行中断线程*/
    IRQTF_WARNED,
    IRQTF_AFFINITY,       /*请求调整线程的 CPU 亲和性*/
    IRQTF_FORCED_THREAD,     /*被强制执行的线程化*/
};
```

●**next**：共享中断时，用于链接同一中断号下的多个中断响应 irqaction 实例。

●**flags**：注册中断时设置的标记，标记中断触发和处理的属性，取值如下（/include/linux/interrupt.h）：

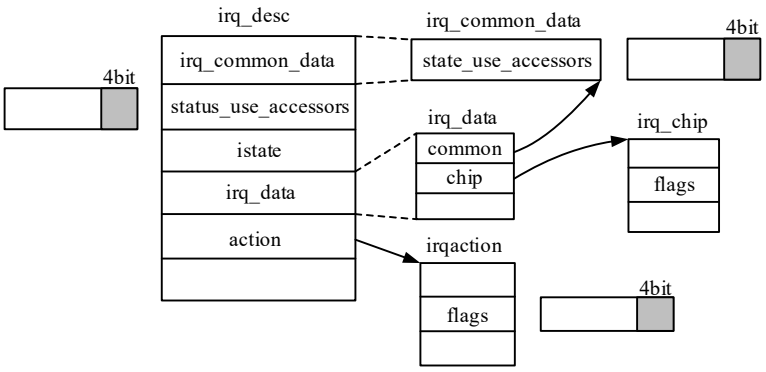
```
#define   IRQF_TRIGGER_NONE          0x00000000
#define   IRQF_TRIGGER_RISING        0x00000001    /*上升沿触发中断*/
#define   IRQF_TRIGGER_FALLING       0x00000002    /*下降沿触发中断*/
#define   IRQF_TRIGGER_HIGH          0x00000004    /*高电平触发中断*/
#define   IRQF_TRIGGER_LOW           0x00000008    /*低电平触发中断*/
#define   IRQF_TRIGGER_MASK    (IRQF_TRIGGER_HIGH | IRQF_TRIGGER_LOW | \
                  IRQF_TRIGGER_RISING | IRQF_TRIGGER_FALLING) /*低 4 位表示触发类型*/
#define   IRQF_TRIGGER_PROBE   0x00000010

#define   IRQF_SHARED          0x00000080     /*允许多个设备共享同一个中断号*/
#define   IRQF_PROBE_SHARED    0x00000100
#define   __IRQF_TIMER         0x00000200    /*定时器中断*/
#define   IRQF_PERCPU          0x00000400    /*特定于某个 CPU 的中断*/
#define   IRQF_NOBALANCING     0x00000800    /*不允许该中断在处理器之间中断均衡*/
#define   IRQF_IRQPOLL         0x00001000    /*中断被用作轮询*/
#define   IRQF_ONESHOT         0x00002000    /*单次触发中断，中断不能嵌套*/
#define   IRQF_NO_SUSPEND      0x00004000    /*系统睡眠时不要关闭本中断*/
#define   IRQF_FORCE_RESUME    0x00008000    /*系统唤醒时必须强制打开本中断*/
#define   IRQF_NO_THREAD       0x00010000    /*中断处理不可线程化*/
#define   IRQF_EARLY_RESUME    0x00020000
#define   IRQF_COND_SUSPEND    0x00040000
#define   IRQF_TIMER       (__IRQF_TIMER | IRQF_NO_SUSPEND | IRQF_NO_THREAD)
```

设备驱动程序在注册中断时，将创建并设置 irqaction 实例，添加到中断描述符 **irq_desc.action** 链表。以上数据结构组织关系如下图所示：



■标记说明

这里还需要对 irq_desc、irqaction 和 irq_chip 结构体中的标记成员说明一下，如下图所示：



irq_desc 结构体中有三个标记成员，如下所示：

●**state_use_accessors**：表示中断底层物理上的信息，通过 irq_data.common 成员访问，低 4 位表示中断触发类型，标记形如 IRQD_XXX，irqd_set_xxx()和 irqd_clr_xxx()函数分别用于设置和清零标记位。在注册中断时会设置/清零标记位

●**status_use_accessors**：中断触发方式和状态属性等，通过 irq_settings_xxx()函数（/kernel/irq/settings.h）操作，低 4 位表示触发类型，标记形式为 IRQ_XXX。在注册中断时会设置/清零标记位

●**istate**：中断状态的内部表示，标记形式为 IRQS_XXX，注册中断时设置。

●**irqaction.flags**：注册中断处理程序时传递的中断信息，标记形式为 IRQF_XXX，低 4 位表示触发类型。

●**irq_chip.flags**：中断控制器标记，形如 IRQCHIP_XXX，定义中断控制器实例时设置，表示中断控制器物理属性。

对于 ONESHOT 类型的中断还要说明一下，ONESHOT 类型中断表示中断不能嵌套，触发一次中断后需要处理完成之后，才能触发下一中断，要求如下：

（1）在硬件中断处理完成之后才能打开中断。

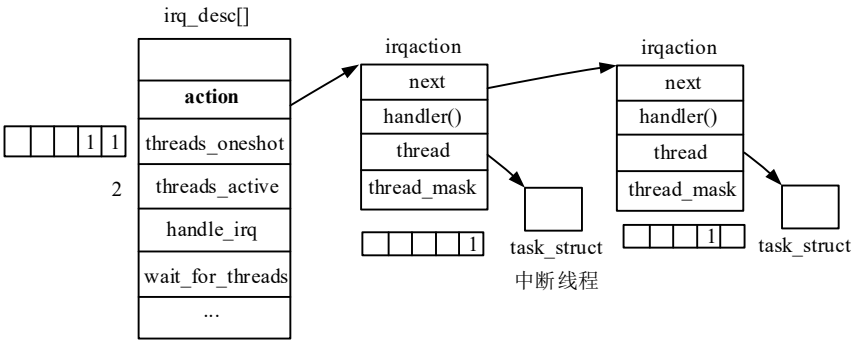
（2）在中断线程化中保持中断关闭状态，直到该中断源上所有 thread_fn()函数完成之后才能打开中断。

（3）如果注册中断时没有指定 handler()函数（为 NULL），且中断控制器不支持硬件 ONESHOT 功能，则需要注册中断时显式设置 IRQF_ONESHOT 标记位。

如下图所示，假设共享中断 irq_desc.action 链表中有两个 irqaction 实例，irqaction 实例中 thread_mask 成员将作为一个位图，设置其中 1 个标记位，标记位的位置表示是链表中第几个 irqaction 实例（最后注册的实例在链表末尾）。

如果共享中断是 ONESHOT 类型，并且注册了中断线程，则只有当所有中断线程结束后才能重新使能该中断。唤醒中断线程时，各 irqaction 实例中 thread_mask 标记位将写入 irq_desc.threads_oneshot 成员（标记所有正在运行中断线程的 irqaction 实例），threads_active 表示运行中断线程数量。中断线程结束时将清零 irq_desc.threads_oneshot 标记位，threads_active 值减 1，当这两个成员值都为 0 时，表示中断线程都结束了，中断处理函数可以使能该中断了。

irq_desc.wait_for_threads 等待队列中是等待 irq 中断处理完成的进程，在使能中断后，唤醒此等待队列中进程。



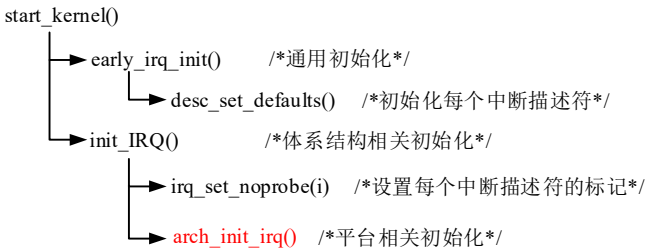
2 中断描述符初始化

内核在/kernel/irq/irqdesc.c 文件内定义了 irq_desc 结构体数组 irq_desc[NR_IRQS]，数组项数 NR_IRQS 由平台（处理器）相关代码定义，因为不同的处理器有不同的中断数量。

```
struct irq_desc irq_desc[NR_IRQS] __cacheline_aligned_in_smp = {      /*没有选择 SPARSE_IRQ 选项*/
        [0 ... NR_IRQS-1] = {
            .handle_irq    = handle_bad_irq,      /*默认通用处理函数，/kernel/irq/handle.c*/
            .depth         = 1,
            .lock          = __RAW_SPIN_LOCK_UNLOCKED(irq_desc->lock),
        }
};
```

■通用初始化

内核中断管理初始化函数调用关系简列如下图所示：



early_irq_init()是所有体系结构通用的初始化函数，负责初始化 irq_desc[NR_IRQS]数组成员。
init_IRQ()是体系结构相关的初始化函数，负责注册中断描述符关联的中断控制器、通用处理函数等。

early_irq_init()函数定义在/kernel/irq/irqdesc.c 文件内：

```
int __init early_irq_init(void)          /*没有选择 SPARSE_IRQ 配置选项*/
{
    int count, i, node = first_online_node;
    struct irq_desc *desc;

    init_irq_default_affinity();
    /*为 irq_default_affinity 分配位图，并全置 1，表示 irq 默认的 CPU 亲和性，/kernel/irq/irqdesc.c*/
```

```
    printk(KERN_INFO "NR_IRQS:%d\n", NR_IRQS);

    desc = irq_desc;       /*指向中断描述符实例数组*/
    count = ARRAY_SIZE(irq_desc);     /*实例数组项数*/

    for (i = 0; i < count; i++) {
        desc[i].kstat_irqs = alloc_percpu(unsigned int);
        alloc_masks(&desc[i], GFP_KERNEL, node);     /*为 desc->irq_data.affinity 分配位图*/
        raw_spin_lock_init(&desc[i].lock);
        lockdep_set_class(&desc[i].lock, &irq_desc_lock_class);
        desc_set_defaults(i, &desc[i], node, NULL);
                                /*初始化 irq_desc[NR_IRQS]数组成员，/kernel/irq/irqdesc.c*/
    }
    return    arch_early_irq_init();   /*空操作，直接返回 0，/kernel/softirq.c*/
}
```

early_irq_init()函数为每个 irq_desc[NR_IRQS]数组成员，分配 CPU 亲和性位图，调用 desc_set_defaults()函数初始化 irq_desc 实例各成员，源代码请读者自行阅读。

init_IRQ()函数定义在/arch/mips/kernel/irq.c 文件内，函数定义如下：

```
void __init init_IRQ(void)
{
    int i;

    for (i = 0; i < NR_IRQS; i++)
        irq_set_noprobe(i);
            /*设置中断描述符状态 irq_common_data.state_use_accessors，/include/linux/irq.h*/
    arch_init_irq();          /*平台（处理器）实现的初始化函数*/
}
```

arch_init_irq()函数由平台（处理器）相关代码实现，主要是为 irq_desc[NR_IRQS]中断描述符设置关联的中断控制器和通用处理函数。后面将介绍龙芯 1B 相关代码实现的此函数。

■设置中断描述符

内核定义了设置（初始化）irq_desc 实例的函数，例如（/include/linux/irq.h）：

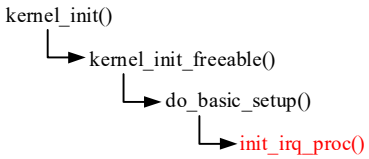
●void **irq_set_handler**(unsigned int **irq**, irq_flow_handler_t **handle**)：设置 irq 号中断对应 irq_desc 实例的 handle_irq 成员值为 handle()，即通用处理函数为 handle()。

●void **irq_set_chip_and_handler**(unsigned int irq, struct irq_chip ***chip**, irq_flow_handler_t **handle**)：设置 irq 号中断对应 irq_desc 实例关联的中断控制器为 chip，通用中断处理函数为 handle()。

在 irq_set_handler()和 irq_set_chip_and_handler()函数内都会调用__irq_set_handler()函数设置中断描述符中的通用函数，源代码请读者自行阅读。

■用户接口

内核通过 proc 文件系统将中断描述符导出到用户空间，使用户能对中断进行查看和操作，相关初始化函数为 init_irq_proc()，函数调用关系如下图所示：



init_irq_proc()函数定义在/kernel/irq/proc.c 文件内，代码如下：

```
void init_irq_proc(void)
{
    unsigned int irq;
    struct irq_desc *desc;

    root_irq_dir = proc_mkdir("irq", NULL);        /*创建/proc/irq/目录*/
    if (!root_irq_dir)
        return;

    register_default_affinity_proc();
                                /*创建/proc/irq/default_smp_affinity 文件，用于设置默认的 CPU 亲和性*/
    for_each_irq_desc(irq, desc) {          /*遍历中断描述符*/
        if (!desc)
            continue;
        register_irq_proc(irq, desc);       /*为每个 irq 在/proc/irq/目录下创建一个目录*/
    }
}
```

init_irq_proc()函数对每个中断描述符调用 register_irq_proc()函数，在/proc/irq/目录下创建一个目录，函数定义如下（/kernel/irq/proc.c）：

```
void register_irq_proc(unsigned int irq, struct irq_desc *desc)
{
    static DEFINE_MUTEX(register_lock);
    char name [MAX_NAMELEN];

    if (!root_irq_dir || (desc->irq_data.chip == &no_irq_chip))
        return;

    mutex_lock(&register_lock);

    if (desc->dir)
        goto out_unlock;
```

```
        memset(name, 0, MAX_NAMELEN);
        sprintf(name, "%d", irq);          /*内部中断号设为目录名称*/

        desc->dir = proc_mkdir(name, root_irq_dir);      /*创建/proc/irq/xxx/目录，xxx 为内核中断号*/
        if (!desc->dir)
            goto out_unlock;

    #ifdef CONFIG_SMP
        /*创建/proc/irq/<irq>/smp_affinity 文件，位掩码*/
        proc_create_data("smp_affinity", 0644, desc->dir, &irq_affinity_proc_fops, (void *)(long)irq);

        /*创建/proc/irq/<irq>/affinity_hint 文件，处理器列表*/
        proc_create_data("affinity_hint", 0444, desc->dir,&irq_affinity_hint_proc_fops, (void *)(long)irq);

        /*创建/proc/irq/<irq>/smp_affinity_list 文件*/
        proc_create_data("smp_affinity_list", 0644, desc->dir,&irq_affinity_list_proc_fops, (void *)(long)irq);

        proc_create_data("node", 0444, desc->dir, &irq_node_proc_fops, (void *)(long)irq);
    #endif
        /*创建/proc/irq/<irq>/spurious 文件，只读文件*/
        proc_create_data("spurious", 0444, desc->dir,&irq_spurious_proc_fops, (void *)(long)irq);

    out_unlock:
        mutex_unlock(&register_lock);
    }
```

3 龙芯 1B 中断管理

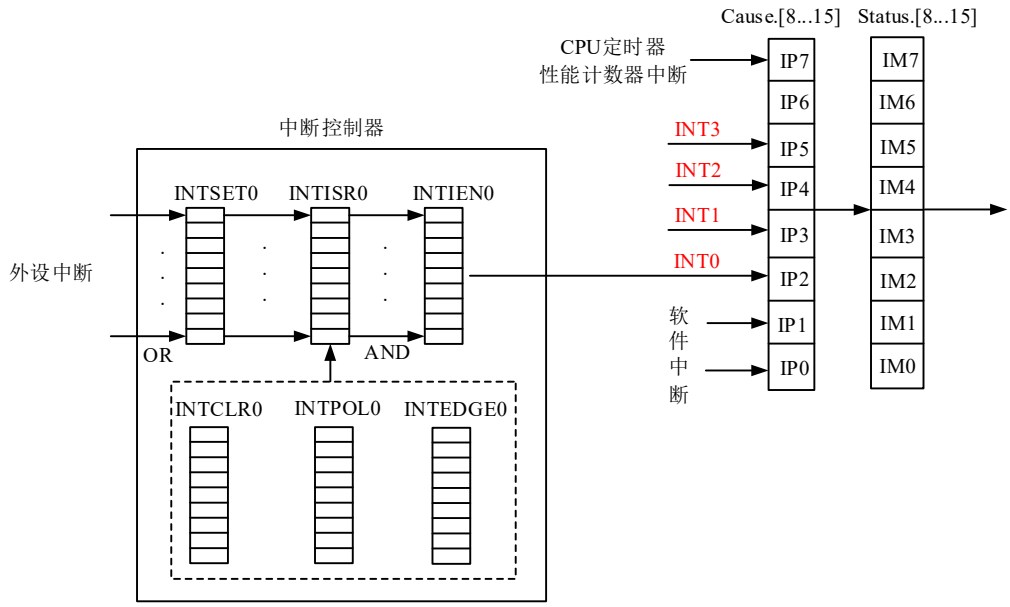
龙芯 1B 芯片（Soc）内置了简单、灵活的中断控制器。中断控制器除了管理 GPIO 输入的中断信号外，还处理外部设备事件引发的中断，支持 64 个 GPIO 中断和 64 个设备中断。

下面介绍龙芯 1B 中断控制器的实现，以及平台代码如何实现对中断的管理和初始化。

■中断控制器

龙芯 1B 处理器内置的中断控制器，支持 64 个 GPIO 中断和 64 个外设中断。中断控制器共输出 4 路中断信号，分别对应 CPU 硬件中断 INT0、INT1、INT2 和 INT3，这 4 个信号分别对应原因寄存器（Cause）IP2-IP5 中断挂起标记位和状态寄存器（Status）IM2-IM5 中断屏蔽位，如下图所示。

每个中断控制器输出信号管理 32 个外设或 GPIO 中断，每 32 个中断由一组中断控制寄存器控制，因此共有 4 组中断控制寄存器。寄存器中每一位对应一个中断。第 0 和第 1 组中断控制寄存器控制 64 个外设中断，第 2 和第 3 组中断控制寄存器控制 64 个 GPIO 中断。下图只画出了第 0 组中断控制寄存器，对应输出中断信号 INT0。



每组中断控制器包含 6 个寄存器，依次为 INTISR、INTIEN、INTSET、INTCLR、INTPOL 和 INTEDGE。

●**INTISR**：中断状态寄存器，每一位指示有无相应的中断，只读。

●**INTIEN**：使能中断寄存器，置位表示使能相应中断，系统复位时默认不使能。

●**INTSET**：中断置位寄存器，边沿触发方式时，写相应位强制置位中断控制状态寄存器中相应位。

●**INTCLR**：中断清空寄存器，边沿触发方式时，写相应位清除中断。

●**INTPOL**：高电平触发中断使能寄存器。

●**INTEDGE**：边沿触发中断使能寄存器。

对于 GPIO 中断，还需要通过 GPIO 配置寄存器，设置 GPIO 属性（如配置成输入引脚），详情请参考处理器手册。

由中断控制器结构可知，当外设产生中断信号时将置位 INTISR 寄存器中相应的位，如果 INTIEN 内相应位也置位，那么中断控制器将产生中断信号 INT。INT 信号可能是由 32 个外设（或 GPIO）中断中的任意一个触发的，此时需要读取 INTISR 和 INTIEN 寄存器值判断是那个中断产生的中断信号。如果设置的是边沿触发方式，中断处理程序中需写 INTCLR 寄存器，清除相应中断。另外，还可通过写 INTSET 寄存器强制产生中断。

在平台相关的/arch/mips/loongson32/common/irq.c 文件内定义了下列宏，用于寻址中断控制寄存器：

```
#define   LS1X_INTC_REG(n, x) \
          ((void __iomem *)KSEG1ADDR(LS1X_INTC_BASE + (n * 0x18) + (x)))
                                                    /*LS1X_INTC_BASE=0x1fd01040*/
#define   LS1X_INTC_INTISR(n)     LS1X_INTC_REG(n, 0x0)     /*第 n 组 INTISR 寄存器地址*/
#define   LS1X_INTC_INTIEN(n)     LS1X_INTC_REG(n, 0x4)     /*第 n 组 INTIEN 寄存器地址*/
#define   LS1X_INTC_INTSET(n)     LS1X_INTC_REG(n, 0x8)     /*第 n 组 INTSET 寄存器地址*/
#define   LS1X_INTC_INTCLR(n)     LS1X_INTC_REG(n, 0xc)     /*第 n 组 INTCLR 寄存器地址*/
#define   LS1X_INTC_INTPOL(n)     LS1X_INTC_REG(n, 0x10)    /*第 n 组 INTPOL 寄存器地址*/
#define   LS1X_INTC_INTEDGE(n)    LS1X_INTC_REG(n, 0x14)    /*第 n 组 INTEDGE 寄存器地址*/
```

■平台初始化

龙芯 1B 外部中断号要转换成内核中断编号才能用于索引 irq_desc[NR_IRQS]数组项。处理器内核中断编号定义在/arch/mips/include/asm/mach-loongson32/irq.h 头文件：

```
#define   MIPS_CPU_IRQ_BASE      0                              /*起始编号为 0*/
#define   MIPS_CPU_IRQ(x)      (MIPS_CPU_IRQ_BASE + (x))     /*x 为偏移量，即内核中断号*/

#define   SOFTINT0_IRQ    MIPS_CPU_IRQ(0)     /*0 为软件中断 0*/
#define   SOFTINT1_IRQ    MIPS_CPU_IRQ(1)     /*1 为软件中断 1*/
#define   INT0_IRQ        MIPS_CPU_IRQ(2)
#define   INT1_IRQ        MIPS_CPU_IRQ(3)
#define   INT2_IRQ        MIPS_CPU_IRQ(4)
#define   INT3_IRQ        MIPS_CPU_IRQ(5)
#define   INT4_IRQ        MIPS_CPU_IRQ(6)
#define   TIMER_IRQ       MIPS_CPU_IRQ(7)   /*CPU 核定时器中断，/arch/mips/kernel/cevt-r4k.c*/
          /*以上是原始的 MIPS 中断编号（0-7），通过设置 Status.[8...15]对应位来关闭/打开中断*/

#define   MIPS_CPU_IRQS   (MIPS_CPU_IRQ(7) + 1 - MIPS_CPU_IRQ_BASE)

#define   LS1X_IRQ_BASE       MIPS_CPU_IRQS     /*中断控制器管理中断的起始内核编号，8*/
#define   LS1X_IRQ(n, x)      (LS1X_IRQ_BASE + (n << 5) + (x))   /*第 n 组 x 号中断的内核编号*/

/*各外设、GPIO 中断内核编号，在设备驱动程序注册中断时使用*/
#define   LS1X_UART0_IRQ              LS1X_IRQ(0, 2)
#define   LS1X_UART1_IRQ              LS1X_IRQ(0, 3)
#define   LS1X_UART2_IRQ              LS1X_IRQ(0, 4)
#define   LS1X_UART3_IRQ              LS1X_IRQ(0, 5)
...
#define   LS1X_IRQS   (LS1X_IRQ(4, 31) + 1 - LS1X_IRQ_BASE)   /*外设和 GPIO 中断总的数量*/

#define   NR_IRQS     (MIPS_CPU_IRQS + LS1X_IRQS)   /*总中断数量，irq_desc[]数组项数*/
```

在平台相关的代码中需要实现中断控制器 irq_chip 结构体实例，并在初始化函数 arch_init_irq()中将其赋予 irq_desc[NR_IRQS]数组中每个中断描述符实例，设置实例关联的通用处理函数指针。

龙芯平台在/arch/mips/loongson32/common/irq.c 文件内定义了中断控制器 irq_chip 实例：

```
static struct irq_chip ls1x_irq_chip = {
    .name           = "LS1X-INTC",
    .irq_ack    = ls1x_irq_ack,             /*写 INTCLR 寄存器相应位，清中断*/
    .irq_mask= ls1x_irq_mask,           /*清零 INTIEN 寄存器相应位，禁止中断*/
    .irq_mask_ack= ls1x_irq_mask_ack,   /*清除中断，并禁止中断*/
    .irq_unmask    = ls1x_irq_unmask,       /*置位 INTIEN 寄存器相应位，使能中断*/
```

```
};
```

irq_chip 实例各成员函数都比较简单，就是对控制寄存器相应位的操作，源代码请读者自行阅读。

平台定义的中断初始化函数 **arch_init_irq()**在/arch/mips/loongson32/common/irq.c 文件内，代码如下：

```
void __init arch_init_irq(void)
{
    mips_cpu_irq_init();          /*清除并屏蔽中断等，/drivers/irqchip/irq-mips-cpu.c*/
    ls1x_irq_init(LS1X_IRQ_BASE);  /*平台相关初始化函数，/arch/mips/loongson32/common/irq.c*/
}
```

龙芯 1B 配置文件中选择了 IRQ_MIPS_CPU 配置选项，因此将编译/drivers/irqchip/irq-mips-cpu.c 文件内，mips_cpu_irq_init()函数在这个文件内定义。

mips_cpu_irq_init()函数会清零 Cause.[8...15]和 Status.[8...15]标记位，清除和屏蔽所在外部中断，注册一个中断域实例，在注册中断域时会设置 SOFTINT0_IRQ 至 INT4_IRQ 中断号（原始的中断号）对应的中断描述符，其关联的 irq_chip 实例为 mips_cpu_irq_controller，通用处理函数为 handle_percpu_irq()，此时所有中断都是屏蔽的。

ls1x_irq_init(LS1X_IRQ_BASE)函数在/arch/mips/loongson32/common/irq.c 文件内实现，代码如下：

```
static void __init ls1x_irq_init(int base)
{
    int n;
    /*禁止所有中断，清除所有中断，中断设为高电平触发*/
    for (n = 0; n < 4; n++)
    {
        __raw_writel(0x0, LS1X_INTC_INTIEN(n));
        __raw_writel(0xffffffff, LS1X_INTC_INTCLR(n));
        __raw_writel(0xffffffff, LS1X_INTC_INTPOL(n));     /*所有中断设为高电平触发*/

        /*设置 DMA0, DMA1 和 DMA2 为边沿触发*/
        __raw_writel(n ? 0x0 : 0xe000, LS1X_INTC_INTEDGE(n));
    }

    for (n = base; n < LS1X_IRQS; n++) {
        irq_set_chip_and_handler(n, &ls1x_irq_chip,handle_level_irq);
                /*初始化中断描述符关联 ls1x_irq_chip 实例，通用处理函数设为 handle_level_irq()*/
    }

    setup_irq(INT0_IRQ, &cascade_irqaction); /*为中断注册 irqaction 实例，同注册中断，使能中断*/
    setup_irq(INT1_IRQ, &cascade_irqaction);
    setup_irq(INT2_IRQ, &cascade_irqaction);
    setup_irq(INT3_IRQ, &cascade_irqaction);
}
```

ls1x_irq_init()函数内清除并关闭所有外设和 GPIO 中断，设置所有中断（除 DMA 中断外）为高电平

触发方式，随后初始化 irq_desc[NR_IRQS]数组实例，设置各中断描述符关联 irq_chip 实例为 **ls1x_irq_chip**（从 LS1X_IRQ_BASE 中断开始），通用处理函数设为 **handle_level_irq()**，即电平触发处理函数，这是内核提供的通用函数，后面将详细介绍。

ls1x_irq_init()函数随后还将为 INT0_IRQ 至 INT3_IRQ 中断注册 irqaction 实例，这个动作等同于后面介绍的注册中断，在注册过程中会使能 INT0_IRQ 至 INT3_IRQ 中断，即 Status.[10...13]位域置 1，使能中断。这个动作很重要，因为在 mips_cpu_irq_init()函数中屏蔽了所有中断，在这里需要打开。

■平台处理函数

在前面介绍的底层中断处理函数中将调用平台定义的 plat_irq_dispatch()函数，查找外部中断对应的内核中断编号，以找到中断对应的中断描述符，调用描述符中的通用处理函数。

龙芯平台定义的中断处理函数 plat_irq_dispatch()如下所示（/arch/mips/loongson32/common/irq.c）：

```
asmlinkage void plat_irq_dispatch(void)
{
    unsigned int pending;

    pending = read_c0_cause() & read_c0_status() & ST0_IM;     /*读取处理器状态及原因寄存器值*/

    if (pending & CAUSEF_IP7)          /*Cause.IP7 置位，表示定时器或性能计数器中断*/
        do_IRQ(TIMER_IRQ);       /*中断号为 TIMER_IRQ，处理函数在/arch/mips/kernel/cevt-r4k.c*/
    else if (pending & CAUSEF_IP2)     /*中断控制器输入的中断，转内核编号，调用 do_IRQ(irq)*/
        ls1x_irq_dispatch(0);          /* INT0 */
    else if (pending & CAUSEF_IP3)
        ls1x_irq_dispatch(1);          /* INT1 */
    else if (pending & CAUSEF_IP4)
        ls1x_irq_dispatch(2);          /* INT2 */
    else if (pending & CAUSEF_IP5)
        ls1x_irq_dispatch(3);          /* INT3 */
    else if (pending & CAUSEF_IP6)
        ls1x_irq_dispatch(4);          /* INT4 */
    else
        spurious_interrupt();        /*增加一次错误中断的次数，/arch/mips/kernel/irq.c*/
}
```

plat_irq_dispatch()函数首先检测 Cause.[8...15]和 Status.[8...15]中同时置位的位，如果最高位置位，则表示产生了定时器或性能计数器中断，要优先处理。

定时器中断处理函数由 MIPS 平台实现，在/arch/mips/kernel/cevt-r4k.c 文件内初始化函数中将为定时器中断描述符关联 irqaction 实例 c0_compare_irqaction，内含中断处理函数，后面再介绍。

plat_irq_dispatch()函数随后依次检测 Cause.IP2 至 Cause.IP6 位，查找第一个置位的位，检测的顺序其实反映的就是中断优先级。如果有置位的位，则调用 ls1x_irq_dispatch(n)函数，n 表示中断控制器中第几组中断。

ls1x_irq_dispatch(int n)函数定义如下（/arch/mips/loongson32/common/irq.c）：

```
static void ls1x_irq_dispatch(int n)
{
    u32 int_status, irq;
    int_status = __raw_readl(LS1X_INTC_INTISR(n)) &__raw_readl(LS1X_INTC_INTIEN(n));
                                                    /*中断状态、使能寄存器*/
    if (int_status) {
        irq = LS1X_IRQ(n, __ffs(int_status));    /*中断控制寄存器第一个置位的中断内核编号*/
        do_IRQ(irq);              /*高层中断处理函数，见下文*/
    }
}
```

ls1x_irq_dispatch(int n)函数参数 n 表示第几组的中断，函数再检测组内中断状态和使能寄存器，找到第一个同时都置位的位（中断），由此转换成内核中断编号 irq，调用高层处理函数 **do_IRQ(irq)**。

中断寄存器内置位位检测的顺序反映了组内中断的优先级。高层处理函数 do_IRQ(irq)依内核中断编号查找中断描述符，调用其中的通用处理函数处理中断，后面再介绍。

4 注册中断

中断初始化完成后，内核中已经有了中断描述符数组和中断控制器，但是中断响应 irqaction 实例还没有，也就是说还没有注册中断的行为，即使这时发生了中断也不会有什么动作。我们知道，中断响应由设备驱动程序注册，俗称注册中断，实际是注册中断响应。注册中断就是创建并设置 irqaction 实例，将其添加到中断描述符 irq_desc.action 链表。

中断响应 irqaction 结构体定义简列如下：

```
struct irqaction {
    irq_handler_t       handler;      /*设备驱动程序注册的中断处理函数*/
    void                *dev_id;      /*设备指针，用于标识设备*/
    void __percpu       *percpu_dev_id;     /*用于标识设备，percpu 变量*/
    struct  irqaction   *next;        /*共享中断，指向下一个 irqaction 实例*/
    irq_handler_t       thread_fn;    /*中断处理线程的执行函数*/
    struct task_struct  *thread;      /*指向中断处理线程 task_struct 实例*/
    unsigned int        irq;          /*内核中断号*/
    unsigned int        flags;        /*注册中断时传递的标记，/include/linux/interrupt.h*/
    unsigned long       thread_flags;       /*中断线程标记，如 IRQTF_FORCED_THREAD*/
    unsigned long       thread_mask;        /*共享中断标记位，设置描述符 threads_oneshot 位图*/
    ...
} ____cacheline_internodealigned_in_smp;
```

irqaction 结构体中 handler 成员是注册的中断处理函数指针，用于执行上半部工作，thread_fn 成员是中断线程中执行函数指针，用于执行下半部工作。这两个函数的原型定义如下（/include/linux/interrupt.h）：

```
typedef   irqreturn_t (*irq_handler_t)(int, void *);  /*第一个参数为中断编号，第二个为设备数据指针*/
```

irq_handler_t()函数返回值为枚举类型 irqreturn_t，定义在/include/linux/irqreturn.h 头文件：

```
enum irqreturn {
```

```
    IRQ_NONE            = (0 << 0),              /*没有产生中断*/
    IRQ_HANDLED         = (1 << 0),              /*中断已得到处理（处理完毕）*/
    IRQ_WAKE_THREAD     = (1 << 1),   /*需唤醒中断处理线程继续处理中断*/
};
```

在中断描述符通用处理函数中将先调用 handler()函数，若其返回 IRQ_WAKE_THREAD，将唤醒中断线程继续处理中断，否则不唤醒。

■注册函数

注册中断的接口函数为 request_irq()和 **request_threaded_irq()**等，前者是较老的接口，后者是新式的接口。

request_irq()函数定义在/include/linux/interrupt.h 头文件：

```
static inline int __must_check    request_irq(unsigned int irq, irq_handler_t handler, unsigned long flags, \
                                                const char *name, void *dev)
/*
*irq：中断内核编号，handler：中断处理函数指针，
*flags：中断标记，如 IRQF_SHARED，传递给 action->flags 成员，name：名称，dev：私有数据指针。
*/
{
    return request_threaded_irq(irq, handler, NULL, flags, name, dev);     /*/kernel/irq/manage.c*/
}
```

request_irq()函数调用 request_threaded_irq()函数实现，它只能注册中断处理函数，不能创建中断线程。

request_threaded_irq()函数定义在/kernel/irq/manage.c 文件内，可用于注册中断处理函数和创建中断线程，函数代码简列如下：

```
int request_threaded_irq(unsigned int irq, irq_handler_t handler,     \
                         irq_handler_t thread_fn, unsigned long irqflags,const char *devname, void *dev_id)
 /*thread_fn：中断线程执行函数指针，其它参数同 request_irq()函数*/
{
    struct irqaction *action;
    struct irq_desc *desc;
    int retval;
    ...
    desc = irq_to_desc(irq);     /*由中断编号获取 irq_desc 实例指针*/
   ...
   if (!handler) {                /*参数 handler 为 NULL 时，设为默认的 irq_default_primary_handler()*/
        if (!thread_fn)       /*handler 为 NULL 时，thread_fn 不能为 NULL*/
            return -EINVAL;
        handler = irq_default_primary_handler;
                              /*函数直接返回 IRQ_WAKE_THREAD，表示要唤醒中断线程*/
    }
    action = kzalloc(sizeof(struct irqaction), GFP_KERNEL);   /*分配 irqaction 实例*/
```

```
    ...
    /*设置 irqaction 实例*/
    action->handler = handler;          /*中断处理函数*/
    action->thread_fn = thread_fn;      /*中断线程执行函数*/
    action->flags = irqflags;           /*传递标记成员，取值定义在/include/linux/interrupt.h 头文件*/
    action->name = devname;
    action->dev_id = dev_id;

    chip_bus_lock(desc);
    retval = __setup_irq(irq, desc, action);
                        /*建立 irqaction 实例与 irq_desc 实例的关联，并创建中断线程，/kernel/irq/manage.c*/
    chip_bus_sync_unlock(desc);
    ...
    return retval;          /*注册成功返回 0*/
}
```

由 request_threaded_irq()函数可知，handler 参数为和 thread_fn 参数不能同时为 NULL，可以有一个为 NULL。当 thread_fn 为 NULL 时，中断只由 handler()函数处理，不需要中断处理线程。当 handler 为 NULL 时，中断将由中断线程处理，线程执行函数为 thread_fn()。

request_threaded_irq()函数为中断描述符创建 irqaction 实例并初始化，然后调用__setup_irq()函数建立 irqaction 实例与中断描述符 irq_desc 实例之间的关联，并创建中断线程等。

内核还定义了注册 percpu 中断接口函数，如下所示（/kernel/irq/manage.c）：

```
int request_percpu_irq(unsigned int irq, irq_handler_t handler,const char *devname, \
                                                        void __percpu *dev_id)
/*dev_id：传递给 irqaction.percpu_dev_id 成员*/
{
    struct irqaction *action;
    struct irq_desc *desc;
    int retval;

    if (!dev_id)
        return -EINVAL;

    desc = irq_to_desc(irq);
    if (!desc || !irq_settings_can_request(desc) ||!irq_settings_is_per_cpu_devid(desc))
        return -EINVAL;

    action = kzalloc(sizeof(struct irqaction), GFP_KERNEL);
    ...
    action->handler = handler;
    action->flags = IRQF_PERCPU | IRQF_NO_SUSPEND;
    action->name = devname;
```

```
    action->percpu_dev_id = dev_id;     /*每个 CPU 有自己的数据*/

    chip_bus_lock(desc);
    retval = __setup_irq(irq, desc, action);          /*设置中断*/
    chip_bus_sync_unlock(desc);
    ...
    return retval;
}
```

percpu 中断注册后并没有使能中断，需要各 CPU 核调用 enable_percpu_irq(irq, type)函数使能中断。

void **free_irq**(unsigned int irq, void *dev_id)函数用于释放注册的中断。

■设置中断

前面介绍的注册中断函数在创建并初始化 irqaction 实例后，都还需要调用**__setup_irq()**函数设置实例，创建中断线程，并关联到 irq_desc 实例等。

下面简略看一下__setup_irq()函数的代码（/kernel/irq/manage.c）：

```
static int __setup_irq(unsigned int irq, struct irq_desc *desc, struct irqaction *new)
{
    struct irqaction *old, **old_ptr;
    unsigned long flags, thread_mask = 0;
    int ret, nested, shared = 0;
    cpumask_var_t mask;
    ...
    nested = irq_settings_is_nested_thread(desc);       /*嵌套中断*/
                                /*desc->status_use_accessors & _IRQ_NESTED_THREAD*/
    if (nested) {
        ...
    } else {
        if (irq_settings_can_thread(desc))
                                /*desc->status_use_accessors 没有设置_IRQ_NOTHREAD 标记位*/
            irq_setup_forced_threading(new);        /*是否执行强制中断处理线程化*/
    }

    if (new->thread_fn && !nested) {        /*创建中断线程*/
        struct task_struct *t;
        static const struct sched_param param = {
            .sched_priority = MAX_USER_RT_PRIO/2,     /*实时优先级*/
        };

        t = kthread_create(irq_thread, new, "irq/%d-%s", irq,new->name);
                                /*创建中断线程，执行函数为 irq_thread()，下一小节介绍*/
```

```
        ...
        sched_setscheduler_nocheck(t, SCHED_FIFO, &param);        /*设为实时进程！！！*/

        get_task_struct(t);
        new->thread = t;            /*指向中断线程 task_struct 实例*/
        set_bit(IRQTF_AFFINITY, &new->thread_flags);        /*后面需要设置 CPU 亲和性*/
}
if (!alloc_cpumask_var(&mask, GFP_KERNEL)) {        /*分配 CPU 核位图 mask*/
        ...
}
if (desc->irq_data.chip->flags & IRQCHIP_ONESHOT_SAFE)
                                                    /*中断控制器只支持 ONESHOT 中断*/
        new->flags &= ~IRQF_ONESHOT;        /*清标记位*/

raw_spin_lock_irqsave(&desc->lock, flags);
old_ptr = &desc->action;
old = *old_ptr;
if (old) {
        ...  /*处理中断描述符已经关联了 irqaction 实例的情形（含共享中断，新实例加到末尾）*/
}

if (new->flags & IRQF_ONESHOT) {
        ...             /*一个共享中断不能超过 32/64 个中断源*/
        new->thread_mask = 1 << ffz(thread_mask);     /*设置 irqaction.thread_mask 中 1 个标记位*/

} else if (new->handler == irq_default_primary_handler &&
                !(desc->irq_data.chip->flags & IRQCHIP_ONESHOT_SAFE)) {
        ...       /*报错*/
        /*若中断控制器没有设置 IRQCHIP_ONESHOT_SAFE 标记位，必须传递 handler()函数*/
}

if (!shared) {        /*不是共享中断*/
        ret = irq_request_resources(desc);       /*调用 irq_chip->irq_request_resources()请求资源*/
        ...
        init_waitqueue_head(&desc->wait_for_threads);     /*初始化中断描述符上等待队列头*/
        if (new->flags & IRQF_TRIGGER_MASK) {        /*设置中断触发类型*/
                ret = __irq_set_trigger(desc, irq,new->flags & IRQF_TRIGGER_MASK);
                ...
        }

        desc->istate &= ~(IRQS_AUTODETECT | IRQS_SPURIOUS_DISABLED | \
                                        IRQS_ONESHOT | IRQS_WAITING);
```

```
        irqd_clear(&desc->irq_data, IRQD_IRQ_INPROGRESS);       /*清标记位*/

        if (new->flags & IRQF_PERCPU) {         /*设置标记位*/
                irqd_set(&desc->irq_data, IRQD_PER_CPU);
                irq_settings_set_per_cpu(desc);
        }

        if (new->flags & IRQF_ONESHOT)
                desc->istate |= IRQS_ONESHOT;

        if (irq_settings_can_autoenable(desc))
                        /*desc->status_use_accessors 没有设置_IRQ_NOAUTOEN 标记位，返回 1*/
                irq_startup(desc, true);        /*使能中断，/kernel/irq/chip.c*/
        else
                desc->depth = 1;

        if (new->flags & IRQF_NOBALANCING) {
                irq_settings_set_no_balancing(desc);
                irqd_set(&desc->irq_data, IRQD_NO_BALANCING);
        }
        setup_affinity(irq, desc, mask);        /*设置中断的 CPU 亲和性，默认设为所有在线 CPU 核*/

} else if (new->flags & IRQF_TRIGGER_MASK) {        /*共享中断*/
        ...
}
new->irq = irq;         /*中断号*/
*old_ptr = new;         /*desc->action=new*/
irq_pm_install_action(desc, new);       /*电源管理相关设置，/kernel/irq/pm.c*/

desc->irq_count = 0;
desc->irqs_unhandled = 0;
if (shared && (desc->istate & IRQS_SPURIOUS_DISABLED)) {     /*共享中断*/
        desc->istate &= ~IRQS_SPURIOUS_DISABLED;
        __enable_irq(desc, irq);        /*使能中断*/
}

raw_spin_unlock_irqrestore(&desc->lock, flags);
if (new->thread)
        wake_up_process(new->thread);         /*唤醒中断线程*/

register_irq_proc(irq, desc);
new->dir = NULL;
```

```
        register_handler_proc(irq, new);      /*创建/proc/irq/<irq>/<new->name>/目录，/kernel/irq/proc.c*/
        free_cpumask_var(mask);
        return 0;
        ...
    }
```

以上函数代码中添加了简要的注释，请读者自行阅读，这里有几个要点要说明一下。

（1）强制中断处理线程化

强制中断处理线程化就是对只注册了 handler()中断处理函数，而没有注册中断线程内执行函数 thread_fn()的中断，强制为其创建中断线程，把 handler()函数作为 thread_fn()函数，放在中断线程内执行，即把原上半部的工作移到下半部。当前对中断处理强制线程化是有条件的。

内核定义了 force_irqthreads 全局变量，表示是否启用强制中断处理线程化，默认值为 0（不启用）。如果要启用该机制，内核配置需选择 IRQ_FORCED_THREADING 配置选项，并在命令行参数中传递参数 "threadirqs"，此参数处理函数会将 force_irqthreads 值设为 true。

在__setup_irq()函数中将调用 irq_settings_can_thread(desc))函数检查 desc->status_use_accessors 是否设置了_IRQ_NOTHREAD 标记位（不允许线程化），如果没有设置，则调用 irq_setup_forced_threading(new)函数检查是否能将中断处理线程化，若能则执行线程化。

irq_setup_forced_threading(new)定义如下（/kernel/irq/manage.c）：

```
static void irq_setup_forced_threading(struct irqaction *new)
{
    if (!force_irqthreads)
        return;
    if (new->flags & (IRQF_NO_THREAD | IRQF_PERCPU | IRQF_ONESHOT))
        return;          /*注册中断标记中若含以上任一个标记位，则不能线程化*/

    new->flags |= IRQF_ONESHOT;

    if (!new->thread_fn) {          /*thread_fn 还需为 NULL*/
        set_bit(IRQTF_FORCED_THREAD, &new->thread_flags);      /*设置标记*/
        new->thread_fn = new->handler;          /*handler 设为 thread_fn*/
        new->handler = irq_default_primary_handler;     /*此函数直接返回 IRQ_WAKE_THREAD*/
    }
}
```

（2）中断处理线程为实时线程，实时优先级为 50，__setup_irq()函数最后会将中断处理线程唤醒。

（3）若中断控制器没有设置 IRQCHIP_ONESHOT_SAFE 标记位，必须传递 handler()函数。电平触发中断也必须设置 handler()函数，在此函数内清中断。

（4）共享中断后注册的 irqaction 实例添加到 irq_desc.action 链表末尾。

（5）如果 irq.desc->status_use_accessors 没有设置_IRQ_NOAUTOEN 标记位，将使能中断。

内核在/kernel/irq/manage.c 文件内还定义了 int **setup_irq**(unsigned int irq, struct irqaction ***act**)函数，用于将一个静态定义的 irqaction 实例注册到 irq 对应的中断描述符，函数内调用__setup_irq()函数，源代码请读者自行阅读。

■带资源的注册函数

内核在/kernel/irq/devres.c 文件内定义了带中断资源的注册中断函数，函数内为设备 device 实例创建表示中断资源的 devres 实例（设备资源详见第 8 章）。

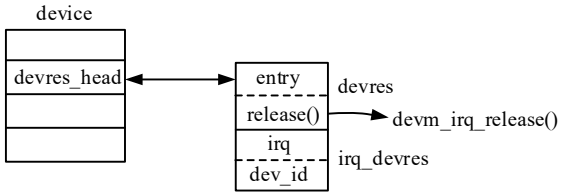
中断资源由 irq_devres 结构体表示，定义如下（/kernel/irq/devres.c）：

```
struct irq_devres {
    unsigned int irq;
    void *dev_id;
};
```

为设备注册带中断资源的注册函数 **devm_request_threaded_irq**()定义如下（/kernel/irq/manage.c）：

```
int devm_request_threaded_irq(struct device *dev, unsigned int irq, irq_handler_t handler,    \
                irq_handler_t thread_fn, unsigned long irqflags, const char *devname, void *dev_id)
/*dev：指向 device 实例，devname：设备名称，dev_id：设备数据*/
{
    struct irq_devres *dr;
    int rc;
    dr = devres_alloc(devm_irq_release, sizeof(struct irq_devres),GFP_KERNEL);
                /*创建 devres 实例，后接 irq_devres 实例，中断资源释放函数为 devm_irq_release()*/
    if (!dr)
        return -ENOMEM;
    rc = request_threaded_irq(irq, handler, thread_fn, irqflags, devname, dev_id);
                                                        /*注册中断*/
    ...
    dr->irq = irq;
    dr->dev_id = dev_id;
    devres_add(dev, dr);      /*添加 devres 实例至 device 实例资源链表*/

    return 0;
}
```

devm_request_threaded_irq()函数创建的数据结构实例如下图所示，通用函数 devm_irq_release()用于释放设备占用的中断资源。



void devm_free_irq(struct device *dev, unsigned int irq, void *dev_id)为释放注册中断的函数。

■中断控制接口

内核在/kernel/irq/manage.c 文件内定义了控制中断的接口函数，供驱动程序（模块）调用，例如：

●void **enable_irq**(unsigned int irq)：使能 irq 中断（声明在/include/linux/interrupt.h，下同）。

●void **disable_irq**(unsigned int irq)：禁止 irq 中断。

●void **irq_wake_thread**(unsigned int irq, void *dev_id)：唤醒中断线程。

●void **synchronize_irq**(unsigned int irq)：当前进程在中断描述符 desc->wait_for_threads 队列等待挂起的 irq 中断处理完成。

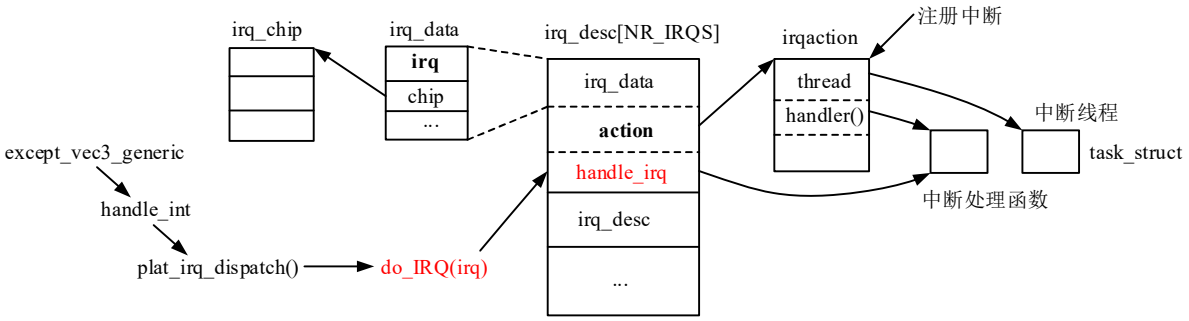
●int **irq_set_affinity**(unsigned int irq, const struct cpumask *cpumask)：设置中断 CPU 亲和性。

6.4.4 高层处理函数

到这里为止，内核中断管理的数据结构已经准备好了，中断处理函数也注册了，可以响应硬件中断了。

在前面介绍的底层中断处理程序 handle_int 中将调用 plat_irq_dispatch()函数，此函数确定中断内核编号后调用高层处理函数 do_IRQ(irq)处理中断，如下图所示。

do_IRQ(irq)函数又将调用中断描述符中指定的 **handle_irq()**，此函数最终调用 irqaction 实例中注册的 handler()函数处理中断，如有需要再唤醒中断线程。



1 do_IRQ()

中断高层处理函数 do_IRQ()定义在/arch/mips/kernel/irq.c 文件内，代码如下：

```
void __irq_entry do_IRQ(unsigned int irq)
{
    irq_enter();       /*进入中断处理前操作，/kernel/softirq.c*/
    check_stack_overflow();       /*检查栈溢出，没有选择 DEBUG_STACKOVERFLOW 为空操作*/
    generic_handle_irq(irq);     /*调用通用中断处理函数，/kernel/irq/irqdesc.c*/
    irq_exit();       /*退出中断处理前操作，/kernel/softirq.c*/
}
```

do_IRQ()函数内调用 generic_handle_irq(irq)函数处理中断，irq_enter()和 irq_exit()函数处理进入和退出中断处理前的工作。

先看一下 generic_handle_irq(irq)函数，定义如下（/kernel/irq/irqdesc.c）：

```
int generic_handle_irq(unsigned int irq)
{
    struct irq_desc *desc = irq_to_desc(irq);     /*由 irq 获取 irq_desc 实例*/

    if (!desc)
        return -EINVAL;
    generic_handle_irq_desc(irq, desc);    /*/include/linux/irqdesc.h*/
    return 0;
```

```
}
```

generic_handle_irq_desc(irq, desc)函数定义在/include/linux/irqdesc.h 头文件：

```
static inline void generic_handle_irq_desc(unsigned int irq, struct irq_desc *desc)
{
    desc->handle_irq(irq, desc);     /*调用 irq_desc 实例中 handle_irq()函数*/
}
```

由以上函数调用可知，中断的处理实际上就是直接调用中断描述符 irq_desc.handle_irq 成员指向的函数。在前面龙芯 1B 中断初始化函数中，中断描述符的 handle_irq 成员赋予 **handle_level_irq()**函数指针，后面将详细介绍此函数实现。

现在先看一下 irq_enter()和 irq_exit()函数的实现，这两个函数都定义在/kernel/softirq.c 文件内。

■进入中断处理

irq_enter()函数表示当前 CPU 将要进入中断处理函数，函数定义如下：

```
void irq_enter(void)
{
    rcu_irq_enter();     /*通知 RCU 机制当前 CPU 正在进入中断处理流程，/kernel/rcu/tree.c*/
    if (is_idle_task(current) && !in_interrupt()) {
                        /*当前进程是否是空闲进程，且不处于硬中断、软中断、不可屏蔽中断内*/
        local_bh_disable();      /*禁止软中断*/
        tick_irq_enter();      /*处理启用动态时钟的情形，见本章下文，/kernel/time/tick-sched.c*/
        _local_bh_enable();  /*开启软中断*/
    }
    __irq_enter();     /*增加硬件中断抢占计数值，/include/linux/hardirq.h*/
}
```

进入高层中断处理函数后，如果当前进程是 idle 进程且不处于硬中断、软中断、不可屏蔽中断内，则调用 tick_irq_enter()函数处理启用了动态时钟的情形（否则不调用），然后调用__irq_enter()函数增加硬中断抢占计数（当前进程 preempt_count 值）。

```
#define __irq_enter()                \
    do {                             \
        account_irq_enter_time(current);  \
        preempt_count_add(HARDIRQ_OFFSET);    \
                            /*抢占计数 HARDIRQ_MASK 位域加 1，标记处于硬中断内*/
        trace_hardirq_enter();               \
    } while (0)
```

■退出中断处理

高层中断处理函数 do_IRQ()返回前将调用 irq_exit()函数，定义如下：

```
void irq_exit(void)          /*调用 irq_exit()函数时须关闭中断*/
{
  #ifndef __ARCH_IRQ_EXIT_IRQS_DISABLED
     local_irq_disable();      /*关闭本地中断*/
```

```
#else
    WARN_ON_ONCE(!irqs_disabled());
#endif

    account_irq_exit_time(current);
    preempt_count_sub(HARDIRQ_OFFSET);
                        /*减小抢占计数 HARDIRQ_MASK 位域值，退出硬中断（非嵌套）*/
    if (!in_interrupt() && local_softirq_pending())
            /*如果有挂起的软中断，且当前 CPU 不处于硬中断、软中断、不可屏蔽中断内*/
        invoke_softirq();      /*执行软中断，见本章下文，/kernel/softirq.c*/

    tick_irq_exit();    /*处理启用动态时钟的情况，见本章下文，/kernel/softirq.c*/
    rcu_irq_exit();      /*处理 RCU 机制*/
    trace_hardirq_exit();   /* must be last! */
}
```

在 irq_exit()函数内，如果当前进程有挂起软中断，且当前 CPU 不处于硬中断、软中断、不可屏蔽中断内，则调用 invoke_softirq()函数执行软中断，详见本章下文。

2 通用处理函数

龙芯 1B 中断初始化函数中，内核 irq_desc[NR_IRQS]数组实例 handle_irq 成员赋予 **handle_level_irq()** 函数指针，在 do_IRQ(irq)函数中将调用此函数处理中断。

handle_level_irq()是内核提供的处理电平触发中断的通用函数，定义在/kernel/irq/chip.c 文件内：

```
void handle_level_irq(unsigned int irq, struct irq_desc *desc)
{
    raw_spin_lock(&desc->lock);
    mask_ack_irq(desc);      /*清除并屏蔽本中断，/kernel/irq/chip.c*/

    if (!irq_may_run(desc))    /*中断处理程序是否可运行，/kernel/irq/chip.c*/
        goto out_unlock;

    desc->istate &= ~(IRQS_REPLAY | IRQS_WAITING);
    kstat_incr_irqs_this_cpu(irq, desc);

    if (unlikely(!desc->action || irqd_irq_disabled(&desc->irq_data))) {
        desc->istate |= IRQS_PENDING;      /*挂起中断*/
        goto out_unlock;
    }

    handle_irq_event(desc);   /*调用中断注册的处理函数，/kernel/irq/handle.c*/
    cond_unmask_irq(desc);    /*打开本中断（需 irq_desc->threads_oneshot=0），/kernel/irq/chip.c*/
```

```
    out_unlock:
        raw_spin_unlock(&desc->lock);
}
```

handle_level_irq()函数首先屏蔽本中断，然后调用 handle_irq_event()函数继续执行中断处理，最后打开中断。这里打开中断要求 irq_desc->threads_oneshot=0，也就是所有中断线程都执行完毕。

handle_irq_event()函数定义在/kernel/irq/handle.c 文件内，代码如下：

```
irqreturn_t handle_irq_event(struct irq_desc *desc)
{
    struct irqaction *action = desc->action;
    irqreturn_t ret;

    desc->istate &= ~IRQS_PENDING;   /*清掉挂起标记位*/
    irqd_set(&desc->irq_data, IRQD_IRQ_INPROGRESS);      /*设置正在处理标记位*/
    raw_spin_unlock(&desc->lock);
    ret = handle_irq_event_percpu(desc, action);
                        /*遍历 desc->action 实例链表，响应中断，/kernel/irq/handle.c*/
    raw_spin_lock(&desc->lock);
    irqd_clear(&desc->irq_data, IRQD_IRQ_INPROGRESS);      /*清零正在处理标记位*/
    return ret;
}
```

handle_irq_event_percpu(desc, action)函数遍历 desc->action 链表中 irqaction 实例，调用实例中 handler()函数，并唤醒中断线程（/kernel/irq/handle.c）。

```
irqreturn_t handle_irq_event_percpu(struct irq_desc *desc, struct irqaction *action)
{
    irqreturn_t retval = IRQ_NONE;
    unsigned int flags = 0, irq = desc->irq_data.irq;

    do {        /*遍历 desc->action 链表中 irqaction 实例，共享中断有多个 irqaction 实例*/
        irqreturn_t res;

        trace_irq_handler_entry(irq, action);
        res = action->handler(irq, action->dev_id);   /*调用 irqaction 实例中处理函数，注意返回值*/
        trace_irq_handler_exit(irq, action, res);

        if (WARN_ONCE(!irqs_disabled(),"irq %u handler %pF enabled interrupts\n", \
                                                        irq, action->handler))
            local_irq_disable();

        switch (res) {              /*action->handler()函数返回值*/
        case IRQ_WAKE_THREAD:       /*需要唤醒中断线程*/
```

```
                if (unlikely(!action->thread_fn)) {
                    warn_no_thread(irq, action);
                    break;
                }
                __irq_wake_thread(desc, action);   /*唤醒中断线程，/kernel/irq/handle.c*/

            case IRQ_HANDLED:        /*中断已被处理，可以返回了*/
                flags |= action->flags;
                break;

            default:
                break;
            }
            retval |= res;
            action = action->next;
        } while (action);     /*遍历 desc->action 链表中 irqaction 实例结束*/

        add_interrupt_randomness(irq, flags);

        if (!noirqdebug)
            note_interrupt(irq, desc, retval);
        return retval;       /*返回值为 irqreturn 枚举类型*/
    }
```

handle_level_irq()函数内遍历 desc->action 链表中 irqaction 实例，对每个实例调用 action->handler(irq, action->dev_id)函数处理中断，如果此函数返回值为 IRQ_HANDLED，则表示中处理完成，处理函数返回。如果 action->handler()函数返回 IRQ_WAKE_THREAD，则表示需要调用**__irq_wake_thread(desc, action)**函数唤醒中断处理线程继续处理。

中断处理函数 action->handler()和处理线程 action->thread 在注册中断时赋值/创建，注册中断在设备驱动程中执行，注册中断的操作见上文。

■唤醒中断处理线程

唤醒中断线程的__irq_wake_thread()函数定义如下（/kernel/irq/handle.c）：

```
void __irq_wake_thread(struct irq_desc *desc, struct irqaction *action)
{
    if (action->thread->flags & PF_EXITING)          /*中断线程正在退出*/
        return;

    if (test_and_set_bit(IRQTF_RUNTHREAD, &action->thread_flags))
        return;          /*action->thread_flags 中 IRQTF_RUNTHREAD 标记位须为 0，并设为 1*/

    desc->threads_oneshot |= action->thread_mask;     /*添加 irqaction 实例标记位至 irq_desc 实例*/
```

atomic_inc(&desc->threads_active); /*增加 threads_active 计数（运行线程数量）*/

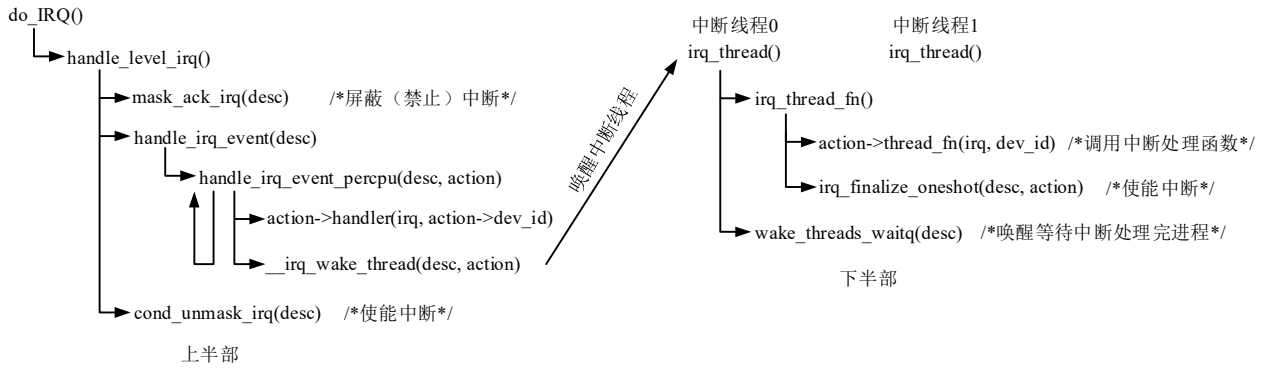
wake_up_process(action->thread); /*唤醒中断线程*/

}

唤醒操作先检测 action->thread_flags 中 IRQTF_RUNTHREAD 标记位，此位为 0 表示中断线程没有运行，然后置位此位，唤醒中断线程。这里还要将 irqaction.thread_mask 添加至 irq_desc.threads_oneshot，在中断线程执行完后会清此标记位。handle_level_irq()函数最后要检测 irq_desc.threads_oneshot 为 0 才会打开中断。

■中断处理线程

在介绍中断处理线程前，先看一下中断处理线程与高层中断处理函数的关系，如下图所示：



左侧表示的是高层中断处理函数，执行中断上半部工作，右侧是中断处理线程，执行中断下半部工作。中断处理线程被唤醒后，中断处理线程与高层中断处理函数是并行执行的（进程并行）。若是共享中断，可能会唤醒多个中断处理线程。

在唤醒线程__irq_wake_thread()函数中将检测 irqaction->thread_flags 中 IRQTF_RUNTHREAD 标记位是否为 0，为 0 则将此位置 1，并唤醒中断处理线程。线程唤醒执行后，会清 0 此标记位。

在前面注册中断调用的__setup_irq()函数中将创建中断线程，其执行函数为 **irq_thread()**，函数定义如下（/kernel/irq/manage.c）：

```
static int irq_thread(void *data)
/*data：指向 irqaction 实例*/
{
    struct callback_head on_exit_work;      /*回调工作*/
    struct irqaction *action = data;
    struct irq_desc *desc = irq_to_desc(action->irq);
    irqreturn_t (*handler_fn)(struct irq_desc *desc,struct irqaction *action);

    if (force_irqthreads && test_bit(IRQTF_FORCED_THREAD,&action->thread_flags))
        handler_fn = irq_forced_thread_fn;      /*强制线程化*/
    else
        handler_fn = irq_thread_fn;         /*将调用 action->thread_fn()函数*/

    init_task_work(&on_exit_work, irq_thread_dtor);
```

```
    task_work_add(current, &on_exit_work, false);     /*on_exit_work 添加到 task->task_works 链表*/

    irq_thread_check_affinity(desc, action);
    /*以上是中断线程第一次运行时才执行的代码*/

    while (!irq_wait_for_interrupt(action)) {       /*可中断睡眠等待，/kernel/irq/manage.c*/
        irqreturn_t action_ret;

        irq_thread_check_affinity(desc, action);
        action_ret = handler_fn(desc, action);     /*调用 irq_thread_fn()函数*/
        if (action_ret == IRQ_HANDLED)
            atomic_inc(&desc->threads_handled);

        wake_threads_waitq(desc);          /*唤醒在中断描述符 desc->wait_for_threads 上等待的进程*/
    }

    task_work_cancel(current, irq_thread_dtor);       /*运行至此表示线程要终止了*/
    return 0;
}
```

中断处理线程首次运行时，将在 task->task_works 链表中添加 on_exit_work 回调函数实例，其回调函数为 irq_thread_dtor()（执行的工作与下面介绍的 irq_finalize_oneshot()函数类似），然后中断处理线程调用 irq_wait_for_interrupt(action)函数，进入可中断睡眠等待。

唤醒后 irq_wait_for_interrupt(action)函数检测标记成员 irqaction->thread_flags 中 IRQTF_RUNTHREAD 标记位是否为 1（唤醒操作中置位），是则清 0 此标记位，函数返回 0。中断线程进入 while()循环内执行，调用函数 handler_fn()函数，这里为 **irq_thread_fn()**函数（假设没有启用强制中断处理线程化），在此函数内将调用中断注册的中断处理线程内的执行函数。

irq_thread_fn()函数定义如下（/kernel/irq/manage.c）：

```
static irqreturn_t irq_thread_fn(struct irq_desc *desc,struct irqaction *action)
{
    irqreturn_t ret;
    ret = action->thread_fn(action->irq, action->dev_id);   /*注册中断时的线程执行函数*/
    irq_finalize_oneshot(desc, action);     /*开中断等*/
    return ret;
}
```

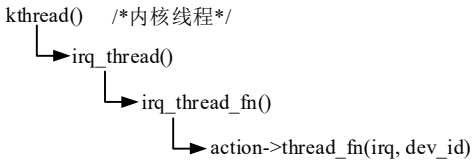
irq_finalize_oneshot()函数定义在/kernel/irq/manage.c 文件内，代码如下：

```
static void irq_finalize_oneshot(struct irq_desc *desc,struct irqaction *action)
{
    if (!(desc->istate & IRQS_ONESHOT))       /*非 ONESHOT 中断，直接返回*/
        return;
again:
    chip_bus_lock(desc);
```

```
        raw_spin_lock_irq(&desc->lock);
        if (unlikely(irqd_irq_inprogress(&desc->irq_data))) {
            raw_spin_unlock_irq(&desc->lock);
            chip_bus_sync_unlock(desc);
            cpu_relax();
            goto again;
        }

        if (test_bit(IRQTF_RUNTHREAD, &action->thread_flags))
            goto out_unlock;

        desc->threads_oneshot &= ~action->thread_mask;        /*清标记位*/

        if (!desc->threads_oneshot && !irqd_irq_disabled(&desc->irq_data) &&
                                            irqd_irq_masked(&desc->irq_data))
            unmask_threaded_irq(desc);          /*使能中断*/

    out_unlock:
        raw_spin_unlock_irq(&desc->lock);
        chip_bus_sync_unlock(desc);
    }
```

irq_finalize_oneshot()函数对于非 ONESHOT 类型中断，直接返回。对于 ONESHOT 类型中断要等到所有中断处理线程都执行完后（irq_desc->threads_oneshot=0），才会去掉中断的屏蔽，使能中断。

在前面设置中断的__setup_irq()函数中调用 kthread_create()函数创建中断处理线程，即内核线程。线程的执行函数其实也不是 irq_thread()函数，而是 kthread()函数，函数调用关系如下（见第 5 章）：



设备驱动程序可通过第 5 章介绍的 kthread_stop(k)函数使中断处理线程 k 退出，kthread_park(k)函数可暂停中断处理线程 k，kthread_unpark(k)函数可恢复中断处理线程 k 的运行。

6.4.5 中断工作队列

中断工作队列机制（需选择 IRQ_WORK 配置选项）提供了在中断处理程序中执行工作的机制。中断工作队列中每项工作由 irq_work 结构体实例表示，结构体定义在/include/linux/irq_work.h 头文件：

```
struct irq_work {
    unsigned long    flags;          /*标记*/
    struct llist_node    llnode;       /*单链表成员*/
    void    (*func)(struct irq_work *);   /*执行函数*/
};
```

irq_work 结构体 flags 标记成员取值定义如下：

```
#define   IRQ_WORK_PENDING     1UL
#define   IRQ_WORK_BUSY        2UL
#define   IRQ_WORK_FLAGS       3UL
#define   IRQ_WORK_LAZY        4UL
```

内核在/include/linux/irq_work.h 头文件定义了声明并初始化 irq_work 实例的宏以及初始化实例函数：

```
#define DEFINE_IRQ_WORK(name, _f)    struct irq_work name = { .func = (_f), }

static inline void init_irq_work(struct irq_work *work, void (*func)(struct irq_work *))
{
    work->flags = 0;
    work->func = func;
}
```

内核在/kernel/irq_work.c 文件内为每个 CPU 核定义了全局单链表头，用于管理注册的 irq_work 实例：

```
static DEFINE_PER_CPU(struct llist_head, raised_list);
static DEFINE_PER_CPU(struct llist_head, lazy_list);   /*链接在周期时钟中断处理的实例*/
```

●bool **irq_work_queue**(struct irq_work *work)：将 irq_work 实例注册到当前 CPU 的中断工作队列。如果实例 flags 标记成员设置成 IRQ_WORK_LAZY 标记，则将其注册到 lazy_list 链表，否则将实例注册到 raised_list 链表。

●bool **irq_work_queue_on**(struct irq_work *work, int cpu)：用于将 irq_work 实例注册到指定 CPU 的 raised_list 中断工作链表。

●**irq_work_run(void)/irq_work_tick(void)**：这两个函数分别扫描当前 CPU 的 raised_list 和 lazy_list 链表，对每个 irq_work 实例调用其中的 func()函数执行工作。

irq_work_run(void)函数在 flush_smp_call_function_queue()函数内调用，**irq_work_tick()**函数在内核周期时钟（节拍）中断处理函数 update_process_times()内调用执行。

6.5 软中断

软中断是延期（异步）执行内核工作的机制，可用于执行硬中断处理的下半部。这里的软中断并不是前面提到的软件中断，前面的软件中断是通过软件写寄存器中的中断挂起标记位来触发中断，其处理流程与硬中断一样。

软中断是完全由软件实现的中断，称之为 softIRQ。软中断数量由内核定义，用户不能动态添加，每个软中断对应一个软中断处理函数（软中断向量），向软中断向量注册处理函数就会启用此软中断。

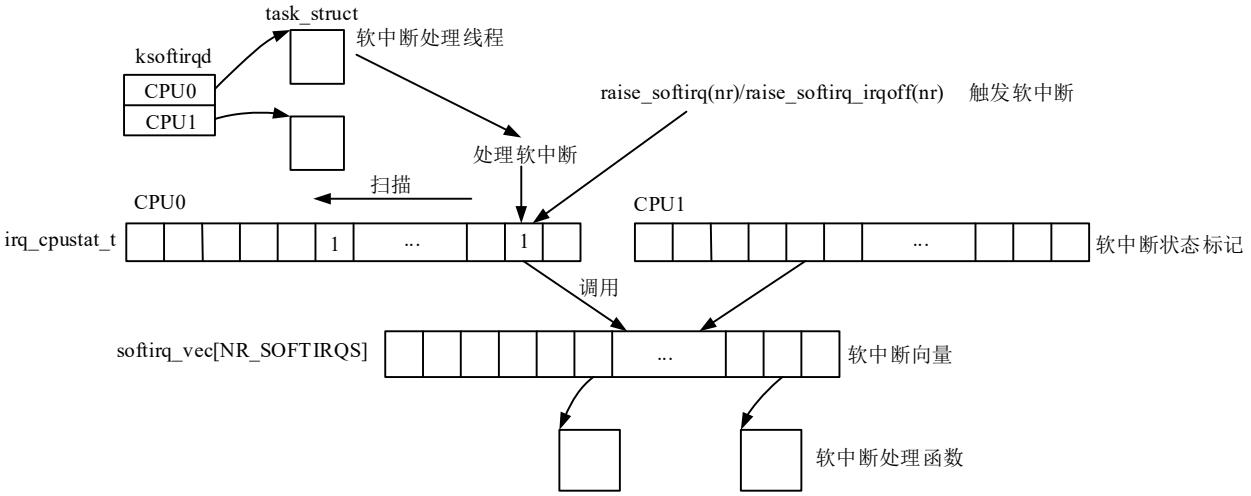
内核为每个 CPU 核定义了软中断状态标记。触发软中断就是置 1 软中断标记位。处理软中断就是扫描软中断标记中置 1 的位，调用对应的处理函数。处理软中断的时机有三个：（1）软中断处理线程中，（2）在高层中断处理函数中调用的 irq_exit()函数内，（3）使能软中断时。

软中断相关代码主要位于/kernel/softirq.c 文件内。

6.5.1 软中断

软中断运行机制如下图所示，内核定义了固定数量的软中断，每个软中断对应一个软中断向量，即软中断处理函数指针。软中断向量由 softirq_vec[]数组表示，数组项只包含处理函数指针成员。

内核为每个 CPU 核定义了一个软中断状态标记结构（实际为无符号整型数）。触发软中断就是置位 CPU 核软中断状态标记中的对应位。处理软中断就是扫描 CPU 核中断状态标记，对置位的软中断调用对应的软中断处理函数。内核在多个时机会检测 CPU 核的软中断状态标记，处理软中断。



由上图可知，每个 CPU 核都有自己的软中断状态标记，但是都对应同一个软中断向量，也就是说不同 CPU 核可以同时处理同一个软中断，因此软中断处理函数必须是可重入的。

1 初始化

软中断的初始化主要包括各 CPU 核软中断标记的定义，软中断向量的注册（注册处理函数）和软中断处理线程的创建等。

■开启软中断

内核软中断类型（编号）定义在/include/linux/interrupt.h 头文件：

```
enum
{
    HI_SOFTIRQ=0,        /*高优先级 tasklet，硬中断的下半部工作*/
    TIMER_SOFTIRQ,       /*处理低分辨率定时器软中断*/
    NET_TX_SOFTIRQ,      /*网络发送数据软中断*/
    NET_RX_SOFTIRQ,      /*网络接收数据软中断*/
    BLOCK_SOFTIRQ,       /*块设备软中断*/
    BLOCK_IOPOLL_SOFTIRQ,
    TASKLET_SOFTIRQ,     /*普通优先级 tasklet，硬中断的下半部工作*/
    SCHED_SOFTIRQ,       /*CFS 调度器类负载均衡软中断*/
    HRTIMER_SOFTIRQ,     /*不再使用，高分辨率定时器直接在中断处理程序中处理*/
    RCU_SOFTIRQ,         /*RCU 软中断*/

    NR_SOFTIRQS          /*软中断数量*/
```

};
软中断编号表示了软中断的优先级，编号越小，优先级越高。内核按优先级从高到低依次处理软中断。

内核在/kernel/softirq.c 文件内定义了各 CPU 核的软中断状态标记，置位的位表示有挂起的软中断：
#ifndef __ARCH_IRQ_STAT
  irq_cpustat_t irq_stat[NR_CPUS] ____cacheline_aligned;
  EXPORT_SYMBOL(irq_stat);
#endif

irq_cpustat_t 结构体定义在/include/asm-generic/hardirq.h 头文件：
typedef struct {
    unsigned int **__softirq_pending**; /*无符号整型数*/
} ____cacheline_aligned **irq_cpustat_t**;

__softirq_pending 成员每一位用于标记是否有相应的软中断挂起等待需要处理。由此可知对于 32 位系统，最多可支持 32 个软中断。

软中断向量定义在/kernel/softirq.c 文件内（全局的向量，不是特定于 CPU 核的）：
static struct softirq_action **softirq_vec[NR_SOFTIRQS]** __cacheline_aligned_in_smp;

软中断向量是 softirq_action 结构体数组，结构体定义如下（/include/linux/interrupt.h）：
struct softirq_action
{
    void (*action) (struct softirq_action *); /*软中断处理函数指针*/
};
软中断向量中只包含相应软中断处理函数指针成员。

内核需要开启某一软中断时，需向软中断向量注册处理函数，注册函数定义如下：
void **open_softirq**(int nr, void (*action)(struct softirq_action *))
/*nr：软中断编号，action：软中断处理函数指针*/
{
    softirq_vec[nr].action = action;
}

内核启动函数 start_kernel()内将调用 softirq_init()函数初始化软中断机制（/kernel/softirq.c）：
void __init softirq_init(void)
{
    int cpu;

    for_each_possible_cpu(cpu) {
        per_cpu(tasklet_vec, cpu).tail =&per_cpu(tasklet_vec, cpu).head;
        per_cpu(tasklet_hi_vec, cpu).tail =&per_cpu(tasklet_hi_vec, cpu).head;

```
    }

    open_softirq(TASKLET_SOFTIRQ, tasklet_action);     /*开启 TASKLET_SOFTIRQ 软中断*/
    open_softirq(HI_SOFTIRQ, tasklet_hi_action);       /*开启 HI_SOFTIRQ（高优先级）软中断*/
}
```

初始化函数内主要完成 TASKLET_SOFTIRQ 和 HI_SOFTIRQ 软中断的开启及相关数据结构（全局链表）的初始化。

■软中断处理线程

本节开头提到软中断可能在三个时机被处理，有一个就是软中断处理线程，因此在初始化时要为每个 CPU 核创建软中断处理线程。

全局指针 ksoftirqd（percpu 变量）指向各 CPU 核软中断处理线程 task_struct 实例（/kernel/softirq.c）：

```
DEFINE_PER_CPU(struct task_struct *, ksoftirqd);
```

内核在启动后期初始化子系统时调用 spawn_ksoftirqd()函数创建软中断处理线程（/kernel/softirq.c）：

```
static __init int spawn_ksoftirqd(void)
{
    register_cpu_notifier(&cpu_nfb);

    BUG_ON(smpboot_register_percpu_thread(&softirq_threads));
                                                /*为每个 CPU 核创建内核线程，/kernel/smpboot.c*/
    return 0;
}
early_initcall(spawn_ksoftirqd);
```

smpboot_register_percpu_thread()函数根据参数 softirq_threads 实例创建内核线程（详见第 5 章创建内核线程），softirq_threads 实例定义如下：

```
static struct smp_hotplug_thread softirq_threads = {
    .store                  = &ksoftirqd,
    .thread_should_run = ksoftirqd_should_run,
    .thread_fn              = run_ksoftirqd,     /*内核线程中调用的函数*/
    .thread_comm        = "ksoftirqd/%u",    /*线程名称*/
};
```

由 softirq_threads 实例定义可知创建的内核线程中将调用 **run_ksoftirqd()**函数，函数定义如下：

```
static void run_ksoftirqd(unsigned int cpu)          /*/kernel/softirq.c*/
{
    local_irq_disable();        /*关本地中断*/
    if (local_softirq_pending()) {      /*检测软中断状态标记是否非零，非零表示有挂起的软中断*/
        __do_softirq();             /*处理软中断，通用函数，/kernel/softirq.c*/
        local_irq_enable();         /*开本地中断*/
```

```
            cond_resched_rcu_qs();
            return;
        }
        local_irq_enable();     /*开本地中断*/
    }
```

run_ksoftirqd()函数内调用__do_softirq()函数处理软中断，这是一个通用函数，后面再详细介绍。

2 触发软中断

内核代码中通过调用 raise_softirq(unsigned int nr)/raise_softirq_irqoff(unsigned int nr)函数来触发某个软中断。

raise_softirq()函数定义如下（/kernel/softirq.c）：

```
void raise_softirq(unsigned int nr)
/*nr：触发的软中断编号*/
{
    unsigned long flags;
    local_irq_save(flags);        /*保存当前本地中断状态，关中断*/
    raise_softirq_irqoff(nr);     /*关中断状态*/
    local_irq_restore(flags);     /*恢复本地之前中断状态*/
}
```

raise_softirq_irqoff()函数定义如下（/kernel/softirq.c）：

```
inline void raise_softirq_irqoff(unsigned int nr)
{
    __raise_softirq_irqoff(nr);     /*当前 CPU 核软中断状态 nr 标记位置 1，/kernel/softirq.c*/

    if (!in_interrupt())
                /*CPU 不在硬件中断、软中断（没有禁止软中断）和不可屏蔽中断处理上下文*/
        wakeup_softirqd();      /*唤醒软中断处理线程*/
}
```

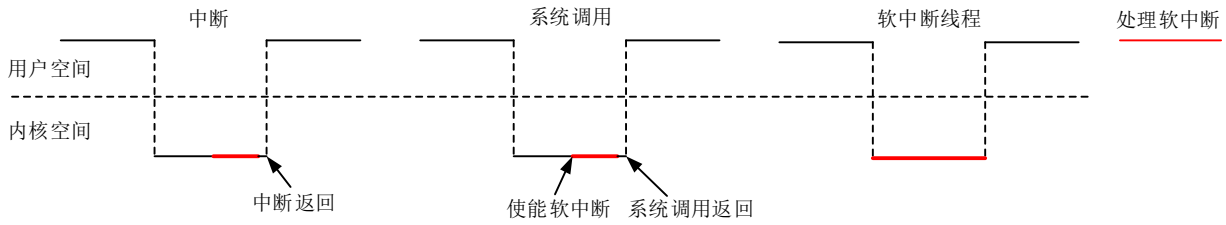
内核中如果直接调用 raise_softirq_irqoff()函数，需在关本地中断的情况下调用。raise_softirq_irqoff()函数设置将当前 CPU 核软中断状态 nr 标记位置 1，然后判断当前 CPU 是不是在硬中断、软中断或不可屏蔽中断处理上下文中，若是则函数返回，不唤醒软中断处理线程，否则唤醒软中断处量线程。

__raise_softirq_irqoff(nr)函数定义如下：

```
void __raise_softirq_irqoff(unsigned int nr)
{
    trace_softirq_raise(nr);
    or_softirq_pending(1UL << nr);          /*置位软中断标记位*/
}
```

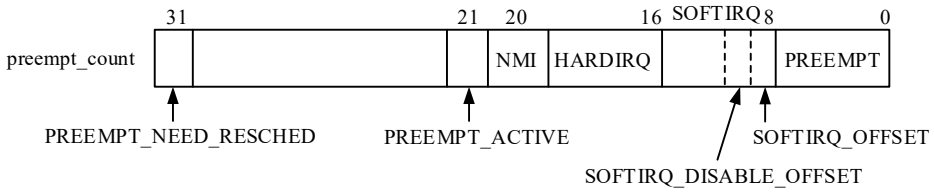
3 处理软中断

处理软中断就是检测 CPU 核软中断标记，对置位的位调用对应软中断向量的处理函数。软中断的处理有三个时机：一是软中断线程，二是硬中断处理函数中，三是使能软中断时。如下图所示，三个时机都是调用相同的软中断处理函数__do_softirq()：



下面先看一下抢占计数中与软中断相关的位域，然后介绍处理软中断的三个时机。

进程抢占计数 thread_info.preempt_count 中有与软中断相关的位域，如下图所示：

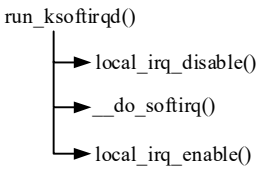


SOFTIRQ 位域对应软中断，表示 CPU 当前正在处理软中断或禁止（屏蔽）软中断。CPU 在处理软中断时会将 SOFTIRQ 位域加 1，处理完成时减 1。当禁止/屏蔽软中断时将对该位域加 2。使能软中断时，先对此位域减 1，然后检测有没有挂起的软中断，如果有则处理，处理完成后，再对 SOFTIRQ 位域减 1。

当 SOFTIRQ 位域非零时，表示禁止内核抢占，也就是说软中断的处理需要尽快的执行完，处理过程中不能发生进程调度（不能睡眠，被其它进程抢占）。软中断设计的目的就是快速地执行中断下半部工作。

■线程处理

在内核启动阶段将为各 CPU 核创建软中断处理线程，线程内调用 **run_ksoftirqd**()函数，函数调用关系如下图所示：



__do_softirq()函数是处理软中断的通用函数，函数内扫描软中断状态标记各比特位，对置 1 的位，调用对应软中断向量 softirq_vec[].action()函数处理该软中断。

__do_softirq()函数代码简列如下（/kernel/softirq.c）：

```
asmlinkage __visible void __do_softirq(void)
{
    unsigned long end = jiffies + MAX_SOFTIRQ_TIME;        /*2 毫秒*/
    unsigned long old_flags = current->flags;              /*当前进程标记*/
    int max_restart = MAX_SOFTIRQ_RESTART;         /*10*/
    struct softirq_action *h;
    bool in_hardirq;
    __u32 pending;
```

```
    int softirq_bit;

    current->flags &= ~PF_MEMALLOC;

    pending = local_softirq_pending();     /*保存当前 CPU 核软中断状态标记*/
    account_irq_enter_time(current);

    __local_bh_disable_ip(_RET_IP_, SOFTIRQ_OFFSET);
                                /*抢占计数 SOFTIRQ 位域加 1，禁止抢占，/kernel/softirq.c*/
    in_hardirq = lockdep_softirq_start();          /*没有选择 TRACE_IRQFLAGS 为空操作*/

restart:
    set_softirq_pending(0);       /*当前 CPU 软中断标记清零*/
    local_irq_enable();           /*开本地中断*/

    h = softirq_vec;     /*软中断向量*/

    while ((softirq_bit = ffs(pending))) {       /*依次查找置 1 的位*/
        unsigned int vec_nr;
        int prev_count;
        h += softirq_bit - 1;         /*软中断向量*/

        vec_nr = h - softirq_vec;
        prev_count = preempt_count();       /*当前进程（处理线程）抢占计数值*/
        kstat_incr_softirqs_this_cpu(vec_nr);
        trace_softirq_entry(vec_nr);
        h->action(h);         /*调用软中断处理函数*/
        trace_softirq_exit(vec_nr);
        if (unlikely(prev_count != preempt_count())) {     /*若处理软中断时有硬中断，有可能会修改*/
            ...         /*输出信息*/
            preempt_count_set(prev_count);         /*设置为原抢占计数，不可修改*/
        }
        h++;       /*下一个软中断向量*/
        pending >>= softirq_bit;       /*软中断状态标记右移*/
    }      /*遍历软中断状态标记比特位结束*/

    rcu_bh_qs();     /*记录加速版不可抢占 RCU 静止状态*/
    local_irq_disable();       /*关闭本地中断*/

    pending = local_softirq_pending();       /*再次读取软中断状态标记*/
    if (pending) {       /*仍有挂起软中断，唤醒软中断处理线程*/
        if (time_before(jiffies, end) && !need_resched() &&--max_restart)
```

```
                goto restart;       /*若处理时长未超过 2 毫秒，且不需要重调度，且重复次数未超过 10*/

            wakeup_softirqd();      /*这里是在软中断处理线程中调用__do_softirq()函数，不需要唤醒*/
        }

        lockdep_softirq_end(in_hardirq);
        account_irq_exit_time(current);
        __local_bh_enable(SOFTIRQ_OFFSET);    /*抢占计数 SOFTIRQ 位域减 1，/kernel/softirq.c*/
        WARN_ON_ONCE(in_interrupt());
        tsk_restore_flags(current, old_flags, PF_MEMALLOC);
    }
```

__do_softirq()函数比较容易理解，其流程简述如下：

（1）读取当前 CPU 核软中断状态标记值（存入局部变量），抢占计数 SOFTIRQ 位域加 1，清零软中断状态标记，开本地中断。

（2）遍历软中断状态标记各比特位，对置位的位调用对应软中断向量中处理函数，处理软中断。

（3）关闭本地中断，再次检测软中断状态标记，若为 0，执行步骤（4）。若不为 0（有挂起软中断），且本次软中断处理时长未超过 2 毫秒，且不需要重调度，且循环次数未超过 10，则跳转至步骤（1）继续处理软中断（循环次数加 1），否则唤醒软中断处理线程（这里不需要唤醒），执行步骤（4）。

（4）抢占计数 SOFTIRQ 位域减 1，处理完成，函数返回。

■硬中断内处理

在前面介绍的硬中断高层处理函数中，在函数返回前会调用 irq_exit()函数，在此函数中也将处理软中断，相关代码如下：

```
void irq_exit(void)
{
    ...
    preempt_count_sub(HARDIRQ_OFFSET);      /*减小硬中断计数值*/
    if (!in_interrupt() && local_softirq_pending())     /*处理软中断条件*/
        invoke_softirq();       /*处理软中断，/kernel/softirq.c*/
    ...
}
```

由以上代码可知，在 irq_exit()函数中处理软中断需要满足两个条件：

（1）当前 CPU 不是处于硬中断、软中断或不可屏蔽中断上下文中。在这之前减小了 HARDIRQ 位域值（抢占计数），因此可以保证不在硬中断处理上下文（嵌套中断除外）。

在软中断处理上下文中有两种情况，一是当前硬中断发生在软中断处理线程中，此时 SOFTIRQ 位域值不为 0，二是当前禁止软中断，若禁止软中断会增加 SOFTIRQ 位域值（详见下文）。

（2）当前 CPU 核有挂起的软中断。

invoke_softirq()函数在这里用于处理软中断，函数定义如下（/kernel/softirq.c）：

```
static inline void invoke_softirq(void)
{
```

```
    if (!force_irqthreads) {        /*没有强制线程化*/
      #ifdef CONFIG_HAVE_IRQ_EXIT_ON_IRQ_STACK
          __do_softirq();
      #else
          do_softirq_own_stack();          /*/include/linux/interrupt.h*/
                /*若没有定义__ARCH_HAS_DO_SOFTIRQ，直接调用__do_softirq()函数*/
      #endif
    } else {        /*强制线程化，唤醒软中断处理线程*/
        wakeup_softirqd();
    }
}
```

invoke_softirq()函数中若没有启用强制中断处理线程化，则调用__do_softirq()函数处理软中断，否则唤醒软中断处理线程。

■禁止/使能软中断

内核在/include/linux/bottom_half.h 头文件内定义了禁止/使能下半部机制的接口函数，其实就是禁止和使能软中断的接口函数。

●禁止软中断

禁止软中断的 local_bh_disable()函数定义如下：

```
static inline void local_bh_disable(void)
{
    __local_bh_disable_ip(_THIS_IP_, SOFTIRQ_DISABLE_OFFSET);
                                    /*SOFTIRQ 位域加 2，/include/linux/bottom_half.h*/
}
```

若内核配置没有选择 TRACE_IRQFLAGS 选项，则__local_bh_disable_ip()函数定义如下：

```
static __always_inline void __local_bh_disable_ip(unsigned long ip, unsigned int cnt)
{
    preempt_count_add(cnt);        /*抢占计数加 cnt*/
    barrier();
}
```

●使能软中断

使能软中断的 local_bh_enable()函数定义如下：

```
static inline void local_bh_enable(void)
{
    __local_bh_enable_ip(_THIS_IP_, SOFTIRQ_DISABLE_OFFSET);
}
```

__local_bh_enable_ip()函数定义如下（/kernel/softirq.c）：

```
void __local_bh_enable_ip(unsigned long ip, unsigned int cnt)
{
```

```
    WARN_ON_ONCE(in_irq() || irqs_disabled());        /*不可在硬中断上下文中，不可关闭硬中断*/
#ifdef CONFIG_TRACE_IRQFLAGS
        local_irq_disable();
#endif

    if (softirq_count() == SOFTIRQ_DISABLE_OFFSET)
        trace_softirqs_on(ip);

    preempt_count_sub(cnt - 1);
                    /*SOFTIRQ 位域减 2，计数值再加 1（仍禁止内核抢占），后面再减 1*/

    if (unlikely(!in_interrupt() && local_softirq_pending())) {
        do_softirq();        /*调用__do_softirq()函数处理软中断*/
    }

    preempt_count_dec();        /*抢占计数减 1*/
#ifdef CONFIG_TRACE_IRQFLAGS
    local_irq_enable();
#endif
    preempt_check_resched();        /*处理重调度*/
}
```

local_bh_enable()函数不可在中断处理函数中调用（只能在开中断状态下调用）。local_bh_enable()函数内将检测当前 CPU 是不是不处于硬中断、软中断上下文中，若是且有挂起软中断，则处理软中断。

在处理软中断前，先对抢占计数 SOFTIRQ 位域减 2，再对抢占计数加 1，用于禁止内核抢占，然后处理软中断，处理完成后抢占计数减 1。

由上面的分析可知，禁止软中断 local_bh_disable()函数会对抢占计数 SOFTIRQ 位域加 2，在此函数之后，若触发软中断将只会设置相应的软中断状态标记位，而不会唤醒软中断处理线程。在随后的硬中断处理函数中也不会处理软中断。

使能软中断 local_bh_enable()函数会恢复禁止软中断之前的抢占计数，当前 CPU 不在硬、软中断上下文中，并且有挂起软中断时，将处理软中断。

local_bh_disable()/local_bh_enable()函数应成对出现，不可在硬中断处理函数中调用（没有必要），这两个函数可以嵌套。CPU 执行这两个函数之间的代码时，不处理软中断（可以设置软中断状态标记），不能进行进程切换，但是可以响应中断，在中断处理函数中不处理软中断。

4 小结

软中断是一种快速执行硬中断处理下半部的机制。触发软中断就是置位 CPU 核软中断状态标记中的相应位，并唤醒软中断处理线程（若当前 CPU 不在硬、软、不可屏蔽中断上下文中）。

软中断的处理有三个时机：（1）硬中断高层处理函数返回前（开中断），（2）使能软中断时，（3）软中断处理线程中。这三个时机最终都是调用__do_softirq()函数处理软中断，此函数在处理软中断达到一

定限制条件时，将唤醒软中断处理线程继续处理，函数返回（不执行过长的时间）。

内核代码中触发软中断后，将在以上三个时机中最近的一个时机处理软中断（如果允许处理）。内核保证一个 CPU 核上的软中断处理是串行进行的，不会嵌套。各 CPU 核的软中断可以并行。

内核通过以下措施来保证一个 CPU 核软中断处理的串行执行：

（1）处理软中断时是禁止内核抢占的，以保证一次处理不会被其它进程中断（中断处理线程禁止内核抢占）。

（2）在软中断处理线程中产生的硬中断，在其处理函数中就不会处理软中断。

（3）在硬中断处理函数中，软中断处理函数中，以及禁止软中断后触发的软中断，只会设置软中断状态标记位，不会唤醒软中断处理线程。

（4）使能软中断时，若 CPU 处于硬中断处理函数中，软中断处理函数中或仍处于禁止软中断状态，将不处理软中断。

由于各 CPU 核使用相同的软中断向量，因此各软中断处理函数必须是可重入的。

各软中断处理函数中不能进入睡眠，原因如下：

（1）处理软中断时，内核是按优先级从高到低依次处理，若某个处理函数睡眠了，其后的处理函数也不会调用了，耽误后面软中断的处理。

（2）软中断可能在硬中断处理上下文中处理，若处理函数睡眠了，中断将不能及时返回，影响进程的实时性。如果在使能软中断时处理软中断进入睡眠，也同样会影响进程实时性。

总之，软中断应是用来处理一些比较紧急，对时间要求较严，不怎么费时的下半部工作，普通的下半部工作应使用中断处理线程或工作队列，放在进程上下文中处理。

6.5.2 tasklet

tasklet（小任务）是利用软中断来执行设备中断下半部工作的一种机制。tasklet 包括高优先级和普通优先级 tasklet，分别对应 HI_SOFTIRQ 和 TASKLET_SOFTIRQ 软中断。

前面说过软中断会影响进程的实时性，其实内核并不推荐使用 tasklet 来执行下半部工作，而应当使用中断处理线程或工作队列。本小节简要介绍 tasklet 的实现。

1 概述

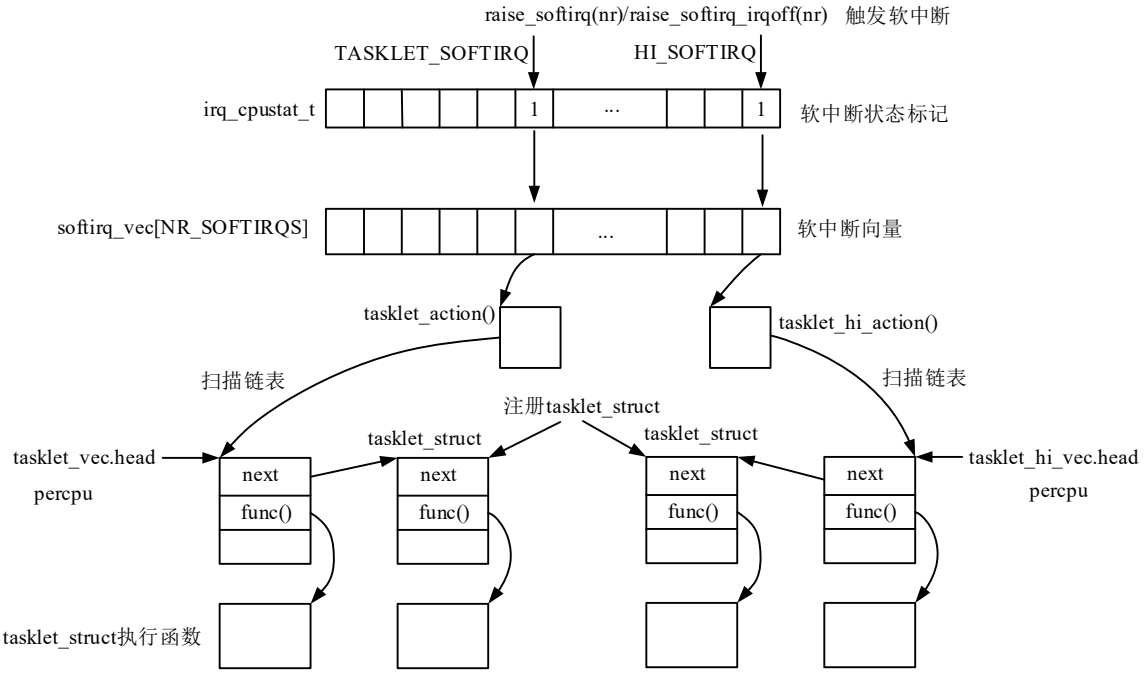
tasklet 框架如下图所示。每个 tasklet 由 task_struct 结构体实例表示，其中 func()函数表示要执行的工作，注意此函数不能进入睡眠，不能主动调度，也不能调用可能引起睡眠的内核函数。

内核为每个 CPU 核定义了两个链表，tasklet_hi_vec 和 tasklet_vec，分别管理高优先级和普通优先级的 task_struct 实例。

HI_SOFTIRQ 软中断处理函数为 tasklet_hi_action()，函数扫描 tasklet_hi_vec 链表中 task_struct 实例，调用其中的 func()函数执行各 tasklet。

TASKLET_SOFTIRQ 软中断处理函数为 tasklet_action()，函数扫描 tasklet_vec 链表中 task_struct 实例，调用其中的 func()函数执行各 tasklet。

设备驱动程序需要定义并注册 task_struct 实例。注册 task_struct 实例的函数为 **tasklet_hi_schedule(t)** 和 **tasklet_schedule(t)**，这两个函数分别将 t 指向的 task_struct 实例添加到 tasklet_hi_vec 和 tasklet_vec 链表，并触发相应软中断，在软中断处理函数中调用 task_struct 实例中的 func()函数。



2 注册 tasklet

tasklet_struct 结构体定义在/include/linux/interrupt.h 头文件：

```
struct tasklet_struct
{
    struct tasklet_struct *next;   /*指向下一实例*/
    unsigned long   state;         /*状态*/
    atomic_t   count;       /*引用计数，不为 0 的，不执行 func()函数*/
    void (*func)(unsigned long);   /*执行函数*/
    unsigned long   data;          /*func()函数参数*/
};
```

tasklet_struct 结构体成员语义如下：

●**next**：指向链表中下一个实例。

●**func**：tasklet 执行函数指针。

●**data**：func()函数参数。

●**count**：引用计数，tasklet_hi_vec 和 tasklet_vec 链表中引用计数非零的实例在执行时将跳过。

●**state**：tasklet 状态，取值定义如下：

```
enum
{
    TASKLET_STATE_SCHED,     /*bit0，置位表示 tasklet_struct 实例已经注册*/
    TASKLET_STATE_RUN        /*bit1，置位 tasklet 正在执行(SMP only) */
};
```

TASKLET_STATE_SCHED 标记位表示 tasklet_struct 实例已经注册，即在 tasklet_hi_vec 或 tasklet_vec 链表中时置此位。CPU 执行软中断时是开中断的，如果在执行本 tasklet_struct 实例时，同一个中断又发生了，在注册 tasklet_struct 实例时将检测 TASKLET_STATE_SCHED 标记位，如果置位将不会重复注册。

TASKLET_STATE_RUN 标记位置位时表示当前正在执行 tasklet_struct 实例，即调用 func()函数，函

数返回后清零 TASKLET_STATE_RUN 标记位。

内核定义并初始化 tasklet_struct 实例的宏如下（/include/linux/interrupt.h）：

```
#define DECLARE_TASKLET(name, func, data) \
   struct tasklet_struct name = { NULL, 0, ATOMIC_INIT(0), func, data }

#define DECLARE_TASKLET_DISABLED(name, func, data) \
   struct tasklet_struct name = { NULL, 0, ATOMIC_INIT(1), func, data }     /*不执行的实例*/
```

内核在/kernel/softirq.c 文件内定义了全局 percpu 链表用于管理注册的 tasklet_struct 实例：

```
struct tasklet_head {
      struct tasklet_struct *head;     /*单链表*/
      struct tasklet_struct **tail;
};
static DEFINE_PER_CPU(struct tasklet_head, tasklet_vec);       /*普通 tasklet*/
static DEFINE_PER_CPU(struct tasklet_head, tasklet_hi_vec);   /*高优先级 tasklet*/
```

tasklet_vec 链表管理普通优先级 tasklet_struct 实例，tasklet_hi_vec 链表管理高优先级 tasklet_struct 实例。

设备驱动程序中创建并初始化 tasklet_struct 实例后需调用注册函数向内核注册 tasklet_struct 实例，通常是在中断处理函数（上半部）中调用注册函数。

注册普通优先级 tasklet_struct 实例的 **tasklet_schedule(struct tasklet_struct *t)**函数定义如下：

```
static inline void tasklet_schedule(struct tasklet_struct *t)         /*/include/linux/interrupt.h*/
{
      if (!test_and_set_bit(TASKLET_STATE_SCHED, &t->state))   /*检测 bit0，需为 0，且置 1*/
            __tasklet_schedule(t);     /*/kernel/softirq.c*/
}
```

tasklet_schedule()函数检测 tasklet_struct 实例状态成员值，bit0 为 0 则对其置 1，并继续执行，否则函数返回。__tasklet_schedule(t)函数将 tasklet_struct 实例添加到**当前 CPU 核**的 tasklet_vec 链表末尾，并触发 TASKLET_SOFTIRQ 软中断（raise_softirq_irqoff(TASKLET_SOFTIRQ)），此处没有唤醒软中断处理线程。

注册高优先级 tasklet_struct 实例的函数为 void **tasklet_hi_schedule**(struct tasklet_struct *t)，它将实例添加到当前 CPU 核的 tasklet_hi_vec 链表末尾，触发 HI_SOFTIRQ 软中断，不唤醒软中断处理线程。

3 执行 tasklet

内核在软中断初始化函数 softirq_init()中开启 HI_SOFTIRQ 和 TASKLET_SOFTIRQ 软中断，相关代码如下：

```
void __init softirq_init(void)
{
      int cpu;
```

```
    for_each_possible_cpu(cpu) {          /*初始化链表*/
        per_cpu(tasklet_vec, cpu).tail =&per_cpu(tasklet_vec, cpu).head;
        per_cpu(tasklet_hi_vec, cpu).tail =&per_cpu(tasklet_hi_vec, cpu).head;
    }

    open_softirq(TASKLET_SOFTIRQ, tasklet_action);     /*开启 tasklet 软中断*/
    open_softirq(HI_SOFTIRQ, tasklet_hi_action);       /*开启 tasklet_hi（高优先级）软中断*/
}
```

HI_SOFTIRQ 和 TASKLET_SOFTIRQ 软中断处理函数分别为 tasklet_hi_action(a)和 tasklet_action(a)，分别用于执行高优先级和普通优先级 tasklet。

以上两个处理函数都定义在/kernel/softirq.c 文件内，下面以 tasklet_action(a)函数为例说明其实现：

```
static void tasklet_action(struct softirq_action *a)
{
    struct tasklet_struct *list;

    local_irq_disable();    /*关本地中断*/
    list = __this_cpu_read(tasklet_vec.head);          /*取出 CPU 核 tasklet_vec 链表*/
    __this_cpu_write(tasklet_vec.head, NULL);          /*清空链表指针*/
    __this_cpu_write(tasklet_vec.tail, this_cpu_ptr(&tasklet_vec.head));
    local_irq_enable();     /*开本地中断*/

    while (list) {     /*遍历 tasklet_struct 实例链表，调用每个实例的执行函数*/
        struct tasklet_struct *t = list;
        list = list->next;
        if (tasklet_trylock(t)) {    /*设置状态成员 TASKLET_STATE_RUN 标记位*/
            if (!atomic_read(&t->count)) {    /*t->count 需为 0，才调用 tasklet_struct 实例执行函数*/
                if (!test_and_clear_bit(TASKLET_STATE_SCHED,&t->state))
                    BUG();
                t->func(t->data);    /*调用执行函数，参数为 tasklet_struct 实例 data 成员*/
                tasklet_unlock(t);
                continue;
            }
            tasklet_unlock(t);       /*清零状态成员 TASKLET_STATE_RUN 标记位*/
        }

        /*tasklet_trylock(t)失败时执行以下代码，t 放回 tasklet_vec 链表末尾*/
        local_irq_disable();
        t->next = NULL;
        *__this_cpu_read(tasklet_vec.tail) = t;
        __this_cpu_write(tasklet_vec.tail, &(t->next));
        __raise_softirq_irqoff(TASKLET_SOFTIRQ);     /*触发软中断*/
        local_irq_enable();
```

```
    }    /*遍历链表结束*/
}
```

tasklet_action()函数比较简单，函数取出 tasklet_vec 链表中实例，依次调用 tasklet_struct 实例中注册的执行函数 func()。

HI_SOFTIRQ 软中断处理函数 tasklet_hi_action(a)与 tasklet_action(a)函数类似，处理 tasklet_hi_vec 链表中实例，源代码请读者自行阅读。

6.6 工作队列

工作队列是内核提供的一种延期（异步）执行内核工作的通用机制。工作队列的基本原理是将工作交由内核线程完成，在进程上下文中执行。因此对工作没有任何限制，可以睡眠，可以长时间运行等。工作队列是执行中断下半部比较好的方式。

工作队列会动态管理执行工作的内核线程，不致于使系统内线程太多，因此比用户（驱动程序）自己创建内核线程执行工作要好。

工作队列相关代码主要位于/kernel/workqueue.c 文件内。

6.6.1 概述

本小节先概述工作队列机制的框架，介绍其原理，以及初始化函数，后面小节将详细介绍工作队列机制的实现。

1 框架

工作队列分为绑定型工作队列和非绑定型工作队列。绑定型工作队列是指提交到本队列的工作只能由指定的 CPU 核完成，不能由其它 CPU 核完成。非绑定型工作队列是指提交的工作不指定由哪个 CPU 核完成，工作队列机制可以决定由哪个 CPU 核完成。

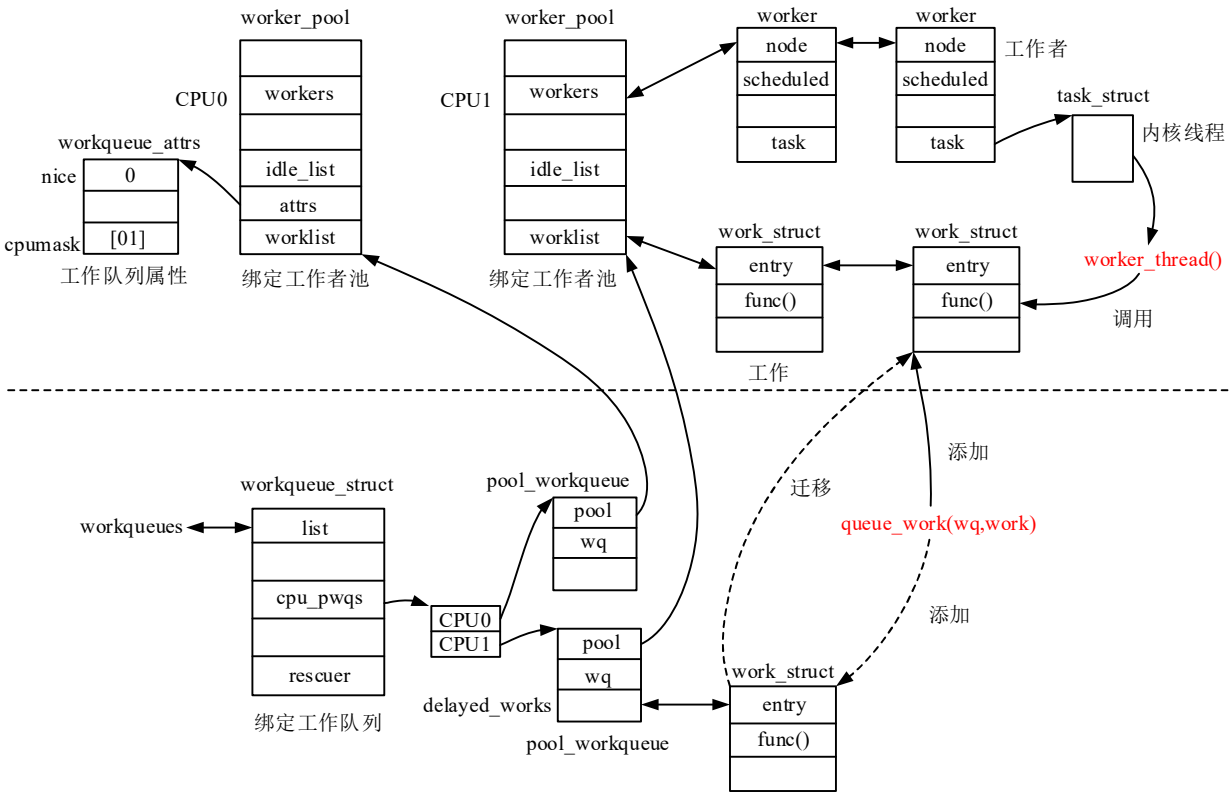
下面先看绑定型工作队列是如何实现的，然后再介绍非绑定型工作队列。如下图所示，工作队列由 workqueue_struct 结构体表示，这可看成向内核提交工作的入口。

工作者池用于管理执行工作的内核线程和提交的工作，由 worker_pool 结构体表示。绑定型工作者池用来处理绑定型工作队列提交的工作和工作者。工作者池中工作者由 worker 结构体表示，即内核线程，工作者池根据工作量动态管理工作者的数量。

工作队列通过 pool_workqueue 结构体建立与工作者池的关联。工作队列可对应多个工作者池，工作者池也可以处理多个工作队列提交的工作。

对于绑定型工作队列，具有一个 pool_workqueue 结构体指针成员，percpu 变量，即工作队列通过此变量指向实例关联到所有 CPU 核的绑定型工作者池。

通俗地说，工作者池相当于“包工头”（或“建筑公司”），管理着“工人”（工作者），工作队列相当于一个“中介公司”，pool_workqueue 是“中介公司”的员工，每个员工可以联系到一个“包工头”。客户发布工作时，先找到“中介公司”，中介公司再通过其员工找到“包工头”，“包工头”揽了活之后将工作分配给手下“工人”去完成。“包工头”可以根据工作量的多少雇用或解雇工人，以达到利益最大化。



用户（驱动程序）提交的工作由 work_struct 结构体表示，其 func()函数指针成员表示要执行的工作。提交工作是提交给工作队列，对于绑定型工作队列，由哪个 CPU 核提交的工作就转交给哪个 CPU 核的工作者池，由其安排工作者完成，也可以在提交工作时指定由哪个 CPU 核处理。

提交的工作将添加到 worker_pool.worklist 双链表，然后工作者池唤醒工作者扫描此链表，执行工作。如果一个工作队列提交的工作过多，工作将缓存在 pool_workqueue.delayed_works 双链表，当有工作处理完成后，再迁移到 worker_pool.worklist 双链表，由工作者执行。

每个 CPU 核具有两个绑定型工作者池用来处理所有绑定工作队列提交的工作，一个用于处理普通优先级工作队列提交的工作，另一个用于处理高优先级工作队列提交的工作。

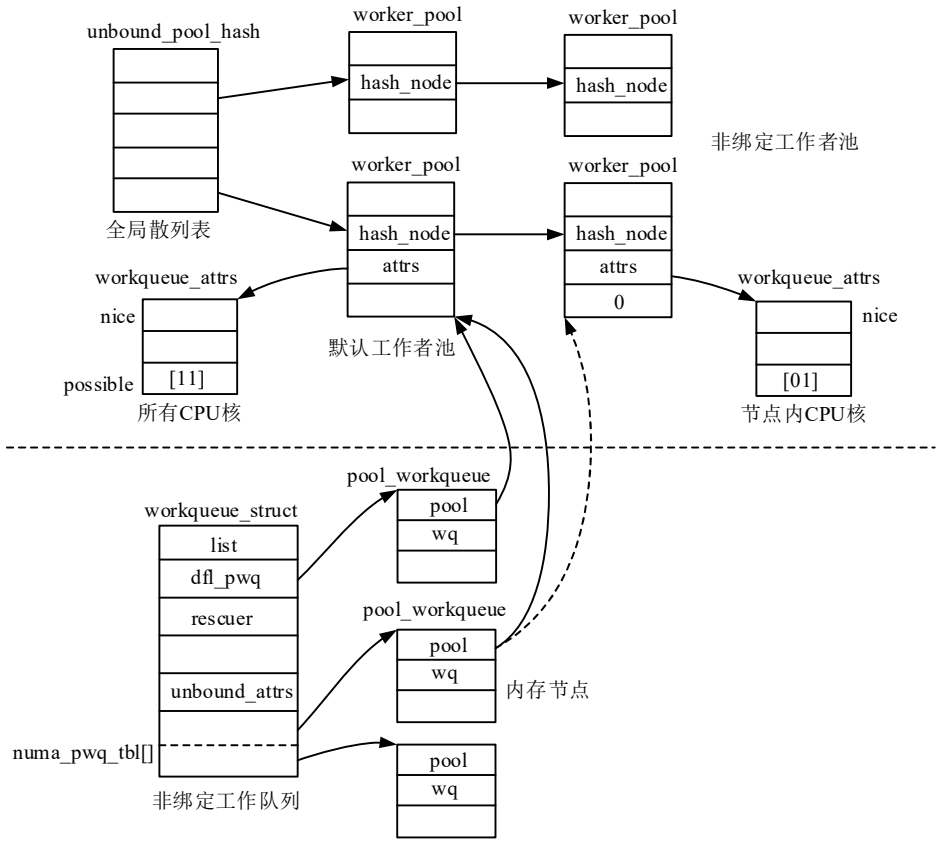
每个工作者池由工作队列属性 workqueue_attrs 标识，也就是说每个工作者池的工作队列属性都不同。工作队列属性中包含工作者池中工作者的 nice 值（进程优先级）、CPU 核亲和性等。对于绑定型工作者池其 CPU 亲和性只关联当前 CPU 核，即工作者只能在当前 CPU 核运行，不能迁移。

非绑定型工作队列如下图所示，它与绑定型工作队列相似。非绑定型工作队列提交的工作不指定由哪个 CPU 核执行。非绑定型工作队列关联到非绑定型工作者池。非绑定型工作者池由全局散列表管理，在创建非绑定型工作队列时动态创建。

非绑定型工作队列分又两种，分别是普通型和 ordered 型。普通型是指非绑定型工作队列除了关联一个默认的工作者池外（CPU 亲和性为所有 CPU 核），还关联具有内存节点亲和性的非绑定型工作者池，即对每个内存节点还有一个工作者池，其 CPU 亲和性为内存节点关联的 CPU 核。ordered 型非绑定工作队列禁止内存节点亲和性，工作队列只关联一个默认的工作者池。

非绑定型工作队列 workqueue_struct 结构体最后是一个 pool_workqueue 结构体指针数组，每个内存节点对应一个数组项。对于 ordered 型非绑定工作队列，所有数组项指向的 pool_workqueue 实例都关联到默认的工作者池，工作队列只关联一个工作者池。普通非绑定型工作队列，各内存节点可以有自己的工作者池或也可使用默认的工作者池，工作队列可以关联到多个工作者池。

两种类型的非绑定工作队列，也区分普通优先级和高优先级，因此共有 4 种非绑定型工作队列。



向非绑定型工作队列提交工作时，由当前 CPU 核或指定 CPU 核确定内存节点，然后由内存节点查找 pool_workqueue 实例，将工作提交给其关联的工作者池。工作的处理和绑定型工作队列相同，只不过工作者可以迁移，这里不再重复了。

内核为各 CPU 核静态定义了绑定型工作者池，不会动态增加，用于处理所有绑定型工作队列提交的工作。非绑定型工作者池在创建非绑定型工作队列时动态创建。

内核提供了创建工作队列的接口，并在初始化阶段创建了一些内核默认的工作队列。内核推荐用户将工作提交到这些默认的工作队列，而不是自己创建新的工作队列。

内核还提供了创建/初始化工作，提交工作，冲刷工作队列等操作的接口函数。如果在中断处理程序中使用工作队列，则一般在中断处理上半部中提交工作，工作中执行中断下半部工作，将由工作者内核线程执行下半部工作。

2 初始化

内核在启动阶段后期初始化各子系统时调用 init_workqueues()函数，初始化工作队列子系统，函数代码如下（/kernel/workqueue.c）：

```
static int __init init_workqueues(void)
{
    int std_nice[NR_STD_WORKER_POOLS] = { 0, HIGHPRI_NICE_LEVEL };
    int i, cpu;

    WARN_ON(__alignof__(struct pool_workqueue) < __alignof__(long long));

    BUG_ON(!alloc_cpumask_var(&wq_unbound_cpumask, GFP_KERNEL));
```

```
cpumask_copy(wq_unbound_cpumask, cpu_possible_mask);   /*wq_unbound_cpumask 分配位图*/

pwq_cache = KMEM_CACHE(pool_workqueue, SLAB_PANIC);
                                /*创建 pool_workqueue 结构体 slab 缓存，用于非绑定型工作队列*/
/*向 CPU 通知链注册通知*/
cpu_notifier(workqueue_cpu_up_callback, CPU_PRI_WORKQUEUE_UP);
hotcpu_notifier(workqueue_cpu_down_callback, CPU_PRI_WORKQUEUE_DOWN);

wq_numa_init();    /*NUMA 系统工作队列初始化（UMA 系统直接返回）*/

/*初始化静态定义的各 CPU 核绑定型工作者池*/
for_each_possible_cpu(cpu) {
      struct worker_pool *pool;

      i = 0;
      for_each_cpu_worker_pool(pool, cpu) {
          BUG_ON(init_worker_pool(pool));   /*初始化工作者池（含分配工作队列属性实例）*/
          pool->cpu = cpu;        /*工作者池绑定的 CPU 核*/
          cpumask_copy(pool->attrs->cpumask, cpumask_of(cpu));    /*只标记本 CPU 核*/
          pool->attrs->nice = std_nice[i++];    /*nice 值*/
          pool->node = cpu_to_node(cpu);      /*CPU 本地内存节点编号*/

          mutex_lock(&wq_pool_mutex);
          BUG_ON(worker_pool_assign_id(pool));    /*为工作者池分配 ID*/
          mutex_unlock(&wq_pool_mutex);
      }
}

/*为绑定工作者池创建初始工作者*/
for_each_online_cpu(cpu) {
    struct worker_pool *pool;
    for_each_cpu_worker_pool(pool, cpu) {
        pool->flags &= ~POOL_DISASSOCIATED;    /*清 POOL_DISASSOCIATED 标记位*/
        BUG_ON(!create_worker(pool));    /*为 worker_pool 实例创建初始工作者*/
    }
}

/*初始化非绑定型工作队列属性，这是所有非绑定型工作者池关联工作队列属性的蓝本*/
for (i = 0; i < NR_STD_WORKER_POOLS; i++) {
    struct workqueue_attrs *attrs;

    BUG_ON(!(attrs = alloc_workqueue_attrs(GFP_KERNEL)));
```

```
        attrs->nice = std_nice[i];
        unbound_std_wq_attrs[i] = attrs;

        BUG_ON(!(attrs = alloc_workqueue_attrs(GFP_KERNEL)));
        attrs->nice = std_nice[i];
        attrs->no_numa = true;          /*禁止内存节点亲和性*/
        ordered_wq_attrs[i] = attrs;
    }

    /*创建内核初始工作队列*/
    system_wq = alloc_workqueue("events", 0, 0);     /*普通优先级绑定型工作队列*/
    system_highpri_wq = alloc_workqueue("events_highpri", WQ_HIGHPRI, 0);
                                                        /*高优先级绑定型工作队列*/
    system_long_wq = alloc_workqueue("events_long", 0, 0);
    system_unbound_wq = alloc_workqueue("events_unbound", WQ_UNBOUND,
                                                    WQ_UNBOUND_MAX_ACTIVE);
                                                        /*普通非绑定型工作队列*/
    system_freezable_wq = alloc_workqueue("events_freezable",WQ_FREEZABLE, 0);
                                                    /*可冻结普通绑定型工作队列*/
    system_power_efficient_wq = alloc_workqueue("events_power_efficient",
                                                        WQ_POWER_EFFICIENT, 0);
                                                        /*节能非绑定型工作队列*/
    system_freezable_power_efficient_wq = alloc_workqueue("events_freezable_power_efficient",
                            WQ_FREEZABLE | WQ_POWER_EFFICIENT, 0);
                                                        /*节能、可冻结非绑定型工作队列*/
    ...
    return 0;
}
early_initcall(init_workqueues);
```

init_workqueues()函数执行的工作简述如下：

（1）为 wq_unbound_cpumask 位图分配空间，并设为 cpu_possible 位图，用于非绑定型工作者池确定 CPU 亲和性。

（2）创建 pool_workqueue 结构体 slab 缓存，向 CPU 通知链注册通知。

（3）NUMA 系统工作队列初始化，主要是为 wq_numa_possible_cpumask[]分配 CPU 核位图，每个内存节点对应一个数组项，即 CPU 核位图，表示内存节点关联到哪些 CPU 核（内存节点的 CPU 亲和性）。

（4）初始化各 CPU 核绑定型工作者池（分配并设置工作队列属性），为其创建初始工作者等。

（5）为非绑定工作者池创建并设置工作队列属性，unbound_std_wq_attrs[]和 ordered_wq_attrs[]数组项指向 workqueue_attrs 实例，分别用于创建普通非绑定型工作池和 ordered 非绑定型工作者池。

（6）创建内核初始工作队列，如 system_wq 等。

6.6.2 工作者池与工作者

工作者池管理着工作者，工作者实际就是内核线程。工作队列关联到工作者池，提交到工作队列的工作将交给关联的工作者池，由工作者池安排工作者处理。

内核为每个 CPU 核静态定义了两个绑定型工作者池，一个用于处理普通优先级工作，另一个用于处理高优先级工作。绑定型工作池就是其下工作者只能在本 CPU 核运行，不能迁移到其它 CPU 核。

内核在初始化函数 init_workqueues()中对各 CPU 核绑定型工作者池进行初始化并为其创建初始的工作者（内核线程）。

非绑定型工作者池是指其下工作者可以在 CPU 核之间迁移，不绑定到某个 CPU 核。非绑定型工作者池可以绑定到内存节点，即其下工作者进程结构实例都位于同一个内存节点中。

在创建非绑定型工作队列时，将根据传递的工作队列标志，动态创建非绑定型工作者池。非绑定型工作者池由全局散列表管理。

工作队列属性标识了工作者池，也就说一个工作队列属性对应一个工作者池，创建工作队列时，相同的工作队列属性共用非绑定型工作者池。

1 工作者池

工作者池由 worker_pool 结构体表示，定义在/kernel/workqueue.c 文件内：

```
struct worker_pool {
    spinlock_t      lock;               /*保护自旋锁*/
    int             cpu;                /*绑定的 CPU 核（绑定工作者池）*/
    int             node;               /*所属内存节点编号*/
    int             id;                 /*工作者池编号*/
    unsigned int        flags;          /*标记成员*/
    struct list_head    worklist;       /*提交工作 work_struct 实例添加到此双链表*/
    int             nr_workers;         /*工作者池中工作者的总数量*/
    int             nr_idle;            /*处于空闲状态工作者数量*/

    struct list_head    idle_list;      /*空闲工作者双链表（worker.entry）*/
    struct timer_list   idle_timer;     /*定时器，注销多余的空闲工作者*/
    struct timer_list   mayday_timer;       /* L:  急救定时器*/

    /*忙状态工作者散列表，工作者位于此散列表或空闲链表或 manager 指针*/
    DECLARE_HASHTABLE(busy_hash, BUSY_WORKER_HASH_ORDER);

    struct mutex        manager_arb;    /* manager arbitration */
    struct worker       *manager;       /* L: purely informational */
    struct mutex        attach_mutex;   /* attach/detach exclusion */
    struct list_head    workers;            /*工作者双链表，新创建工作者添加到此双链表末尾*/
    struct completion   *detach_completion;     /*完成量*/

    struct ida      worker_ida;                 /*管理其下工作者 ID*/
```

```
    struct workqueue_attrs   *attrs;          /*工作队列属性，工作者优先级、CPU 亲和性等*/
    struct hlist_node    hash_node;       /*非绑定工作者池，添加到全局散列表*/
    int              refcnt;              /*非绑定工作者池，引用计数*/

    atomic_t  nr_running    ____cacheline_aligned_in_smp;
                        /*正在运行工作者数量（有的运行工作者会被忽略），独占一个缓存行*/
    struct rcu_head      rcu;        /*回调函数*/
} ____cacheline_aligned_in_smp;
```

worker_pool 结构体主要成员简介如下：

●**worklist**：双链表头，链接提交的工作 work_struct 实例。

●**idle_list**：双链表头，链接空闲工作者（worker.entry）。

●**workers**：双链表头，链接工作者池中的所有工作者（worker.node）。

●**busy_hash**：工作者池中忙工作者散列表（worker.hentry）。

●**attrs**：指向工作队列属性 workqueue_attrs 实例，在创建工作者池时分配，用于标识工作者池。

workqueue_attrs 结构体定义如下（/include/linux/workqueue.h）

```
struct workqueue_attrs {
    int         nice;       /*工作者 nice 值*/
    cpumask_var_t       cpumask;        /*工作者 CPU 亲和性*/
    bool        no_numa;        /*是否禁止内存节点亲和性，用于非绑定型工作者池*/
};
```

●**nr_workers**：工作者池中工作者的总数量，worklist 双链表中工作者数量。

●**nr_idle**：处于空闲状态工作者数量，idle_list 双链表中工作者数量。此值为 0 将启动动态创建工作者操作。

●**nr_running**：正在运行工作者数量（有的运行工作者会被忽略）。在动态创建工作者时，若此值为 0，将创建新工作者。在提交工作时，若此值为 0 将唤醒新的空闲工作者。

●**idle_timer**：定时器，到期函数用于注销多余的空闲工作者。

●**mayday_timer**：定时器，用于处理创建工作者失败时的紧急情况。

●**flags**：工作者池标记成员，取值定义如下（/kernel/workqueue.c）：

```
enum {                  /*枚举类型中还定义了工作者标记和其它常数*/
    POOL_DISASSOCIATED    = 1 << 2,       /*置位表示非绑定型工作者池*/

    /*工作者标记*/
    WORKER_DIE          = 1 << 1,    /*工作者死亡*/
    WORKER_IDLE         = 1 << 2,    /*工作者空闲*/
    WORKER_PREP         = 1 << 3,    /*工作者正准备执行工作（刚唤醒）*/
    WORKER_CPU_INTENSIVE     = 1 << 6,  /*CPU 敏感型工作者（消耗 CPU）*/
    WORKER_UNBOUND      = 1 << 7,  /*工作者是非绑定的（可在 CPU 核间迁移）*/
    WORKER_REBOUND      = 1 << 8,  /*工作者是重新绑定的*/

    WORKER_NOT_RUNNING     = WORKER_PREP | WORKER_CPU_INTENSIVE |
                        WORKER_UNBOUND | WORKER_REBOUND,
```

```
                                                    /*工作者不是运行状态，不计入 nr_running 值*/
    NR_STD_WORKER_POOLS     = 2,        /*每种类型工作者池的数量，普通优先级和高优先级*/
    UNBOUND_POOL_HASH_ORDER  = 6,       /*非绑定工作者池散列表阶数*/
    BUSY_WORKER_HASH_ORDER  = 6,        /*忙工作者散列表阶数*/

    MAX_IDLE_WORKERS_RATIO      = 4,        /* 1/4 of busy can be idle */
    IDLE_WORKER_TIMEOUT     = 300 * HZ,   /*空闲工作者存活时间（5 分钟）*/

    MAYDAY_INITIAL_TIMEOUT   = HZ / 100 >= 2 ? HZ / 100 : 2,
    MAYDAY_INTERVAL     = HZ / 10,      /* and then every 100ms */
    CREATE_COOLDOWN     = HZ,           /* time to breath after fail */

    RESCUER_NICE_LEVEL   = MIN_NICE,
    HIGHPRI_NICE_LEVEL   = MIN_NICE,      /*高优先级工作者池中工作者 nice 值*/
    WQ_NAME_LEN         = 24,
};
```

■绑定型工作者池

内核在/kernel/workqueue.c 文件内为每个 CPU 核静态定义了绑定型工作者池，如下所示：

```
static DEFINE_PER_CPU_SHARED_ALIGNED(struct worker_pool [NR_STD_WORKER_POOLS], \
                                                    cpu_worker_pools);
```

NR_STD_WORKER_POOLS 值为 2，cpu_worker_pools 为指向 worker_pool[]数组的 percpu 变量。

两个绑定型工作者池分别表示普通优先级和高优先级工作者池，用于处理普通工作队列和高优先级工作队列提交的工作。

在前面介绍的 init_workqueues()初始化函数中将调用 init_worker_pool()函数初始化各 CPU 核的静态工作者池。

init_worker_pool()函数定义如下（/kernel/workqueue.c）：

```
static int init_worker_pool(struct worker_pool *pool)
{
    spin_lock_init(&pool->lock);
    pool->id = -1;
    pool->cpu = -1;
    pool->node = NUMA_NO_NODE;     /*-1，不绑定内存节点，后面会重设*/
    pool->flags |= POOL_DISASSOCIATED;       /*非绑定工作者池，后面会清此标记位*/
    INIT_LIST_HEAD(&pool->worklist);
    INIT_LIST_HEAD(&pool->idle_list);
    hash_init(pool->busy_hash);          /*初始化散列表*/

    init_timer_deferrable(&pool->idle_timer);
```

```
    pool->idle_timer.function = idle_worker_timeout;   /*定时器到期执行函数，注销空闲工作者*/
    pool->idle_timer.data = (unsigned long)pool;        /*worker_pool 实例地址*/

    setup_timer(&pool->mayday_timer, pool_mayday_timeout,(unsigned long)pool);
                                        /*设置定时器，创建工作者失败时紧急处理工作*/
    mutex_init(&pool->manager_arb);
    mutex_init(&pool->attach_mutex);
    INIT_LIST_HEAD(&pool->workers);

    ida_init(&pool->worker_ida);        /*初始化 ida 实例*/
    INIT_HLIST_NODE(&pool->hash_node);
    pool->refcnt = 1;

    pool->attrs = alloc_workqueue_attrs(GFP_KERNEL);
                            /*分配 workqueue_attrs 实例，CPU 核位图设为 possible CPU 核*/
    if (!pool->attrs)
        return -ENOMEM;
    return 0;
}
```

工作者池实例中 idle_timer 定时器到期执行函数为 idle_worker_timeout()，用于清理工作者池中的空闲工作者，后面再做介绍。

工作者池实例中 mayday_timer 定时器到期执行函数为 pool_mayday_timeout()，用于创建工作者失败时紧急处理工作者池中提交的工作，后面再做介绍。

init_workqueues()函数在调用 init_worker_pool()函数初始化绑定型工作者池实例后，还将设置关联的工作队列属性，清零各实例的 POOL_DISASSOCIATED 标记位，并调用 create_worker(pool)函数为工作者池创建初始的工作者，见上文。

■创建非绑定工作者池

非绑定型工作者池是动态创建的，并且由全局散列表 unbound_pool_hash 管理。在创建非绑定工作队列时，将动态创建非绑定型工作者池。

get_unbound_pool()函数用于根据工作队列属性查找或创建非绑定型工作者池，函数定义如下：

```
static struct worker_pool *get_unbound_pool(const struct workqueue_attrs *attrs)
/*attrs：指向 workqueue_attrs 实例*/
{
    u32 hash = wqattrs_hash(attrs);      /*由 workqueue_attrs 实例计算散列值*/
    struct worker_pool *pool;
    int node;

    lockdep_assert_held(&wq_pool_mutex);
```

```
    hash_for_each_possible(unbound_pool_hash, pool, hash_node, hash) {   /*查找散列表*/
        if (wqattrs_equal(pool->attrs, attrs)) { /*workqueue_attrs 实例相同，共用 worker_pool 实例*/
            pool->refcnt++;     /*引用计数加 1*/
            return pool;        /*返回 worker_pool 实例指针*/
        }
    }

    /*需创建新 worker_pool 实例*/
    pool = kzalloc(sizeof(*pool), GFP_KERNEL);
    if (!pool || init_worker_pool(pool) < 0)     /*初始化 worker_pool 实例，含分配工作队列属性*/
        goto fail;

    lockdep_set_subclass(&pool->lock, 1);       /* see put_pwq() */
    copy_workqueue_attrs(pool->attrs, attrs);      /*复制工作队列属性*/
    pool->attrs->no_numa = false;

    if (wq_numa_enabled) {       /*wq_numa_init()函数将此变量设为 true*/
        for_each_node(node) {
            if (cpumask_subset(pool->attrs->cpumask,wq_numa_possible_cpumask[node])) {
                pool->node = node;   /*内存节点 CPU 核位图包含了 pool->attrs->cpumask*/
                break;
            }
        }
    }

    if (worker_pool_assign_id(pool) < 0)       /*为工作者池分配 ID*/
        goto fail;

    if (!create_worker(pool))       /*创建初始工作者，见下文*/
        goto fail;

    hash_add(unbound_pool_hash, &pool->hash_node, hash);      /*worker_pool 实例添加到散列表*/

    return pool;      /*返回 worker_pool 实例指针*/
    ...
}
```

get_unbound_pool()函数根据 attrs 参数传递的工作队列属性 workqueue_attrs 实例，在全局散列表中查找是否有相同属性的 worker_pool 实例，若有则直接引用，返回 worker_pool 实例指针。若没有则创建并初始化 worker_pool 实例，复制 attrs 参数传递的工作队列属性至 worker_pool 实例，为其创建初始工作者，将 worker_pool 实例添加到全局散列表，最后返回新实例指针。

非绑定工作者池关联的 workqueue_attrs 实例，其 CPU 核位图是 attrs 参数传递 workqueue_attrs 实例中的位图，表示 CPU 核亲和性。

对于 NUMA 系统在 init_workqueues()初始化函数中将调用 wq_numa_init()函数初始化 NUMA 系统中工作队列，主要是为数组 wq_numa_possible_cpumask[]分配 CPU 核位图，每个内存节点对应一个数组项，即 CPU 核位图，表示此内存节点的 CPU 核亲和性，函数内还会设置 wq_numa_enabled 变量为 true。

在创建非绑定工作者池时，若工作队列属性中 CPU 核位图被包含于某个 wq_numa_possible_cpumask[]数组项位图，即工作者池亲和某个内存节点，则将此内存节点 node 编号赋予 pool->node 成员，表示工作者池绑定到内存节点 node。

2 工作者

工作者池中管理的工作者由 worker 结构体表示，定义如下（/kernel/workqueue_internal.h）：

```
struct worker {
    union {
        struct list_head  entry;        /*worker 添加到工作者池空闲工作者 idle_list 双链表*/
        struct hlist_node     hentry;   /*worker 添加到工作者池忙工作者散列表*/
    };

    struct work_struct  *current_work;   /*当前正在处理的工作*/
    work_func_t         current_func;    /*当前处理工作的执行函数*/
    struct pool_workqueue  *current_pwq;   /*指向 pool_workqueue 结构体，当前工作关联的实例*/
    bool                desc_valid;      /* desc[]成员是否有效*/
    struct list_head    scheduled;       /*正准备由本工作者执行的挂起工作 work_struct 实例双链表*/

    struct task_struct  *task;           /*工作者进程结构指针*/
    struct worker_pool  *pool;           /*所属工作者池*/
    struct list_head    node;            /*将实例添加到工作者池 worker_pool->workers 双链表*/

    unsigned long       last_active;     /*工作者最近一次活跃的时间戳*/
    unsigned int        flags;           /*工作者标记，见上文中 worker_pool 结构体 flags 成员介绍*/
    int         id;         /*工作者 ID*/

    char        desc[WORKER_DESC_LEN];      /*工作者描述*/
    struct workqueue_struct *rescue_wq;     /*用于急救工作者，指向工作队列*/
};
```

worker 结构体主要成员简介如下：

●**entry/hentry**：将工作者实例添加到工作者池空闲工作者双链表或忙工作者散列表。

●**current_work**：指向当前正在执行的工作，内核提交的工作由 work_struct 结构体表示（见下文）。

●**current_func**：当前工作执行函数指针。

●**scheduled**：链接等待调度执行的工作 work_struct 实例。当提交的多个工作 work_struct 实例有关联关系时，将这些 work_struct 实例添加到此链表，由工作者逐个处理。处理单个工作时，直接处理不添加到此双链表。

●**task**：工作者 task_struct 实例指针。

●**node**：将工作者实例链接到工作者池 worker_pool->workers 双链表。

工作者池中的工作者是动态管理的，在初始化函数中将会创建初始工作者，运行过程中将动态的创建和注销工作者。

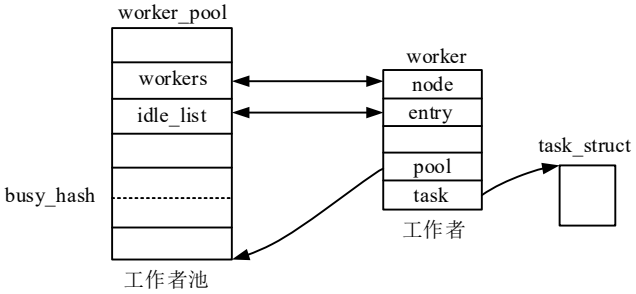
■创建工作者

创建工作者的 create_worker()函数定义如下：

```
static struct worker *create_worker(struct worker_pool *pool)
/*pool：归属的工作者池*/
{
    struct worker *worker = NULL;
    int id = -1;
    char id_buf[16];

    /*工作者分配编号*/
    id = ida_simple_get(&pool->worker_ida, 0, 0, GFP_KERNEL);
    ...    /*错误处理*/
    worker = alloc_worker(pool->node);
                        /*分配并初始化 worker 实例，设置 WORKER_PREP 标记位*/
    ...
    worker->pool = pool;
    worker->id = id;      /*工作者编号*/
    ...  /*输出信息*/

    worker->task = kthread_create_on_node(worker_thread, worker, pool->node,"kworker/%s", id_buf);
                                    /*创建内核线程，调用函数为 worker_thread()，见下文*/
    ...
    set_user_nice(worker->task, pool->attrs->nice);      /*设置工作者 nice 值（调度优先级）*/

    worker->task->flags |= PF_NO_SETAFFINITY;       /*不可由用户更改 CPU 亲和性*/

    worker_attach_to_pool(worker, pool);
        /*设置工作者 CPU 亲和性为 pool->attrs->cpumask，处理 WORKER_UNBOUND 标记位*/
        /*通过 worker->node 成员将工作者添加到 pool->workers 链表末尾*/

    spin_lock_irq(&pool->lock);
    worker->pool->nr_workers++;      /*工作者池中工作者数量加 1*/
    worker_enter_idle(worker);    /*设置工作者 WORKER_IDLE 标记位，pool->nr_idle++，
                                /*工作者添加到 pool->idle_list 链表头部等*/
    wake_up_process(worker->task);    /*唤醒工作者*/
    spin_unlock_irq(&pool->lock);
    return worker;       /*返回工作者指针*/
    ...
```

}

创建工作者的函数比较简单，函数内创建 worker 实例并初始化，为其创建内核线程，线程内调用函数为 worker_thread()（后面再介绍），建立工作者与工作者池的关联，工作者添加到工作者池中工作者双链表和空闲工作者双链表，如下图所示，工作者标记为空闲，最后唤醒工作者。



工作者的动态创建在工作者线程调用 worker_thread()函数中实现，也就是说由工作者创建工作者，详见下文工作者线程中的介绍。

■注销工作者

工作者池中嵌入的 idle_timer 定时器，其到期的执行函数为 **idle_worker_timeout()**，函数内检测工作者池中空闲工作者数量，如果过多则注销一些，函数定义如下。

```
static void idle_worker_timeout(unsigned long __pool)
{
    struct worker_pool *pool = (void *)__pool;
    spin_lock_irq(&pool->lock);

    while (too_many_workers(pool)) {     /*空闲工作者过多（空闲工作者比忙工作者多很多）*/
        struct worker *worker;
        unsigned long expires;

        worker = list_entry(pool->idle_list.prev, struct worker, entry);       /*从链表末尾扫描工作者*/
        expires = worker->last_active + IDLE_WORKER_TIMEOUT;    /*下次定时期到期时间*/

        if (time_before(jiffies, expires)) {       /*如果间隔时间不够，返回*/
            mod_timer(&pool->idle_timer, expires);
            break;
        }
        destroy_worker(worker);    /*设置工作者 WORKER_DIE 标记位，唤醒线程*/
    }
    spin_unlock_irq(&pool->lock);
}
```

idle_worker_timeout()函数判断是否有过多的空闲工作者，如果是则从工作者池空闲工作者双链表末尾取出一个工作者（新空闲工作者添加到头部，末尾的就是空闲时间最久的工作者），调用 destroy_worker()

函数注销工作者。重复以上动作，直到没有过多的空闲工作者。

destroy_worker(worker)函数内设置工作者的 WORKER_DIE 标记位，唤醒工作者线程，线程调用函数 worker_thread()中将检测 WORKER_DIE 标记位，如果为 1，则 worker_thread()函数返回，此函数返回内核线程就会退出，详见第 5 章创建内核线程。

6.6.3 创建工作队列

工作队列是提交工作的通道，对工作的提交取控制作用，其实它本身并不管理工作。工作队列由结构体 workqueue_struct 表示，内核提供了创建工作队列的接口函数。

1 数据结构

工作队列由 workqueue_struct 结构体表示，定义如下（/kernel/workqueue.c）：

```
struct workqueue_struct {
    struct list_head    pwqs;      /*双链表头，管理其下所有 pool_workqueue 实例*/
    struct list_head    list;      /*将实例添加到全局双链表 workqueues*/

    struct mutex        mutex;         /* protects this wq */
    int             work_color;        /* WQ: current work color  当前颜色值，初始值为 0*/
    int             flush_color;       /* WQ: current flush color，当前冲刷颜色值，初始值为 0*/
    atomic_t        nr_pwqs_to_flush;  /*尚未完成本次冲刷的 pool_workqueue 实例数量*/
    struct wq_flusher   *first_flusher;     /* WQ: first flusher 指向第一个冲刷者*/
    struct list_head    flusher_queue;     /* WQ: flush waiters  挂起的冲刷操作 wq_flusher 实例链表*/
    struct list_head    flusher_overflow;  /* WQ: flush overflow list，溢出的冲刷操作链表*/

    struct list_head    maydays;    /*链接需要急救的 pool_workqueue 实例*/
    struct worker       *rescuer;
            /*如果创建工作队列时设置了 WQ_MEM_RECLAIM 标记位，指向急救工作者*/

    int   nr_drainers;   /* WQ: drain in progress */
    int    saved_max_active;   /* WQ: saved pwq max_active */

    struct workqueue_attrs   *unbound_attrs;   /*工作队列属性，非绑定工作队列使用*/
    struct pool_workqueue   *dfl_pwq;          /*用于关联非绑定工作队列默认的 worker_pool 实例*/

  #ifdef CONFIG_SYSFS
    struct wq_device    *wq_dev;    /*sysfs  文件系统接口*/
  #endif
  #ifdef CONFIG_LOCKDEP
    struct lockdep_map lockdep_map;
  #endif
    char        name[WQ_NAME_LEN];    /*工作队列名称*/
    struct rcu_head     rcu;
```

```
    unsigned int    flags    ____cacheline_aligned;      /*工作队列标记成员*/
    struct pool_workqueue __percpu    *cpu_pwqs;        /*指向 pool_workqueue 实例*/
                                    /*绑定工作队列，对每个 CPU 核创建一个 pool_workqueue 实例*/
    struct pool_workqueue __rcu    *numa_pwq_tbl[];
                                    /*非绑定工作队列，由内存节点 node 索引的 pool_workqueue 实例*/
};
```

workqueue_struct 结构体主要成员简介如下：

●**pwqs**：双链表，链接关联的 pool_workqueue 实例，见下文创建工作队列函数。

●**list**：将工作队列实例链接到全局双链表 workqueues。

●**rescuer**：指向急救工作者，当创建工作队列时设置了 WQ_MEM_RECLAIM 标记位，将创建急救工作者。急救工作者用于处理动态创建空闲工作者失败时，紧急处理工作者池中已提交的工作。

●**maydays**：双链表，所有需要急救的 pool_workqueue 实例添加到此链表。

●**unbound_attrs**：指向工作队列属性，用于非绑定型工作队列。

●**dfl_pwq**：指向 pool_workqueue 实例，用于非绑定型工作队列，通过此实例关联默认的工作者池。

●**cpu_pwqs**：percpu 变量，指向 pool_workqueue 实例。绑定型工作队列使用此成员，通过 pool_workqueue 实例关联各 CPU 核的绑定型 worker_pool 实例。

●**numa_pwq_tbl[]**：pool_workqueue 指针数组，非绑定型工作队列使用此成员，每个内存节点对应一个数组项，通过 pool_workqueue 实例关联非绑定型 worker_pool 实例。

●**flags**：工作队列标记成员（创建工作队列时的标记），取值定义在/include/linux/workqueue.h 头文件：

```
enum {
    WQ_UNBOUND          = 1 << 1,   /*置位表示非绑定型工作队列，否则表示绑定型工作队列*/
    WQ_FREEZABLE        = 1 << 2,   /*系统 suspend 过程中，工作者处理完所有工作后冻结*/
    WQ_MEM_RECLAIM      = 1 << 3,   /*创建急救工作者*/
    WQ_HIGHPRI          = 1 << 4,   /*高优先级工作队列*/
    WQ_CPU_INTENSIVE    = 1 << 5,   /*工作队列中工作特别消耗 CPU，每个工作独占一个工作者*/
    WQ_SYSFS            = 1 << 6,       /*工作队列通过 sysfs 文件系统导出*/

    WQ_POWER_EFFICIENT  = 1 << 7,       /*工作队列考虑系统能耗*/

    __WQ_DRAINING       = 1 << 16,  /* internal: workqueue is draining */
    __WQ_ORDERED            = 1 << 17,  /*ordered 型工作队列*/

    WQ_MAX_ACTIVE           = 512,      /*工作队列中最大活跃工作数量*/
    WQ_MAX_UNBOUND_PER_CPU  = 4,    /*用来确定最大非绑定工作队列数量，4*cpus*/
    WQ_DFL_ACTIVE           = WQ_MAX_ACTIVE / 2,
};
```

poo_workqueue 结构体用来建立工作队列 workqueue_struct 与工作者池 worker_pool 之间的关联，结构体定义如下（/kernel/workqueue.c）：

```
struct pool_workqueue {
    struct worker_pool *pool;           /*指向关联的工作者池*/
```

```
    struct workqueue_struct *wq;       /*指向关联工作队列*/
    int         work_color;    /* L: current color  当前颜色值*/
    int         flush_color;   /* L: flushing color  当前冲刷颜色值，初始值为-1*/
    int         refcnt;        /* L: reference count  引用计数*/
    int         nr_in_flight[WORK_NR_COLORS];     /*各颜色值未处理完工作的数量*/
    int         nr_active;     /*当前活跃的工作数量（提交但未处理完的工作）*/
    int         max_active;    /*最大活跃工作数量*/
    struct list_head    delayed_works;    /*被延迟执行的工作添加到此链表，nr_active 达到最大值*/
    struct list_head    pwqs_node;    /*链入工作队列 wq->pwqs 双链表*/
    struct list_head    mayday_node;  /*链入工作队列 wq->maydays 双链表（急救双链表）*/

    struct work_struct  unbound_release_work; /*非绑定工作队列，用于释放 pool_workqueue 实例*/
    struct rcu_head         rcu;
} __aligned(1 << WORK_STRUCT_FLAG_BITS);     /*8 位，256 字节对齐*/
```

在创建工作队列 workqueue_struct 实例时，将动态创建 pool_workqueue 实例，并依此建立与工作者池的关联。对于绑定型工作队列，workqueue_struct.cpu_pwqs 成员是指向 pool_workqueue 实例的 percpu 变量，也就是说工作队列通过 pool_workqueue 实例与每个 CPU 核的绑定型工作者池关联。

对于非绑定工作队列，通过 workqueue_struct.dfl_pwq 指向的 pool_workqueue 实例关联默认的工作者池。workqueue_struct.numa_pwq_tbl[]是一个 pool_workqueue 指针数组，通过 pool_workqueue 实例关联绑定到此内存节点的工作者池，详见下文。

2 创建函数

内核在/include/linux/workqueue.h 头文件定义了创建工作队列的宏，如下所示：

```
#ifdef CONFIG_LOCKDEP
    ...
#else
#define alloc_workqueue(fmt, flags, max_active, args...)        \
    __alloc_workqueue_key((fmt), (flags), (max_active), NULL, NULL, ##args) /*/kernel/workqueue.c*/
#endif

/*以下是旧式的接口*/
#define alloc_ordered_workqueue(fmt, flags, args...)        \       /*ordered 非绑定型工作队列*/
    alloc_workqueue(fmt, WQ_UNBOUND | __WQ_ORDERED | (flags), 1, ##args)

#define create_workqueue(name)                              \
    alloc_workqueue("%s", WQ_MEM_RECLAIM, 1, (name))
#define create_freezable_workqueue(name)                    \
    alloc_workqueue("%s", WQ_FREEZABLE | WQ_UNBOUND | WQ_MEM_RECLAIM, 1, (name))
#define create_singlethread_workqueue(name)                 \
    alloc_ordered_workqueue("%s", WQ_MEM_RECLAIM, name)
```

void **destroy_workqueue**(struct workqueue_struct *wq)：注销工作队列，/kernel/workqueue.c。

创建工作队列的接口函数最终都是调用__alloc_workqueue_key()函数完成工作队列的创建，函数定义如下（/kernel/workqueue.c）：

```
struct workqueue_struct *__alloc_workqueue_key(const char *fmt,unsigned int flags,int max_active,  \
                                               struct lock_class_key *key,const char *lock_name, ...)
/*fmt：lock_name 指向字符格式，flags：工作队列标记，max_active：最大活跃工作数量，
*lock_name：名称。*/
{
    size_t tbl_size = 0;
    va_list args;
    struct workqueue_struct *wq;
    struct pool_workqueue *pwq;

    if ((flags & WQ_POWER_EFFICIENT) && wq_power_efficient)
                    /*wq_power_efficient 受 WQ_POWER_EFFICIENT_DEFAULT 配置项控制*/
        flags |= WQ_UNBOUND;        /*设为非绑定工作队列*/

    if (flags & WQ_UNBOUND)         /*非绑定型工作队列*/
        tbl_size = nr_node_ids * sizeof(wq->numa_pwq_tbl[0]);
                                                    /*workqueue_struct 实例末尾指针数量*/

    wq = kzalloc(sizeof(*wq) + tbl_size, GFP_KERNEL);
                                    /*分配 workqueue_struct 实例，附加 tbl_size 大小空间*/
    ...
    if (flags & WQ_UNBOUND) {               /*非绑定工作队列分配工作队列属性实例*/
        wq->unbound_attrs = alloc_workqueue_attrs(GFP_KERNEL);
        ...
    }

    va_start(args, lock_name);
    vsnprintf(wq->name, sizeof(wq->name), fmt, args);
    va_end(args);

    max_active = max_active ?: WQ_DFL_ACTIVE;
    max_active = wq_clamp_max_active(max_active, flags, wq->name);

    /*初始化工作队列*/
    wq->flags = flags;
    wq->saved_max_active = max_active;
    mutex_init(&wq->mutex);
```

```
atomic_set(&wq->nr_pwqs_to_flush, 0);
INIT_LIST_HEAD(&wq->pwqs);
INIT_LIST_HEAD(&wq->flusher_queue);
INIT_LIST_HEAD(&wq->flusher_overflow);
INIT_LIST_HEAD(&wq->maydays);

lockdep_init_map(&wq->lockdep_map, lock_name, key, 0);
INIT_LIST_HEAD(&wq->list);

if (alloc_and_link_pwqs(wq) < 0)     /*分配并设置 pool_workqueue 实例，关联到工作者池*/
        goto err_free_wq;

if (flags & WQ_MEM_RECLAIM) {       /*创建急救工作者*/
        struct worker *rescuer;

        rescuer = alloc_worker(NUMA_NO_NODE);        /*创建工作者*/
        if (!rescuer)
                goto err_destroy;

        rescuer->rescue_wq = wq;
        rescuer->task = kthread_create(rescuer_thread, rescuer, "%s",wq->name);
                                        /*急救工作者线程调用函数为 rescuer_thread()*/
        ...
        wq->rescuer = rescuer;
        rescuer->task->flags |= PF_NO_SETAFFINITY;       /*用户不可设置 CPU 亲和性*/
        wake_up_process(rescuer->task);    /*唤醒 rescuer 工作者*/
}

if ((wq->flags & WQ_SYSFS) && workqueue_sysfs_register(wq))       /*导出到 sysfs 文件系统*/
        goto err_destroy;

mutex_lock(&wq_pool_mutex);
mutex_lock(&wq->mutex);

for_each_pwq(pwq, wq)
        pwq_adjust_max_active(pwq);        /*调整各 pool_workqueue.max_active 值*/

mutex_unlock(&wq->mutex);
list_add_tail_rcu(&wq->list, &workqueues);     /*工作队列添加到全局 workqueues 双链表末尾*/
mutex_unlock(&wq_pool_mutex);

return wq;       /*返回 workqueue_struct 实例指针*/
```

```
    ...
}
```

__alloc_workqueue_key()函数执行的工作简述如下：

（1)如果 flags 标记参数设置了 WQ_POWER_EFFICIENT 标记位且 wq_power_efficient 变量值为 true，则设置工作队列为非绑定型工作队列。wq_power_efficient 变量值受 WQ_POWER_EFFICIENT_DEFAULT 配置选项控制，选择此选项时，值为 true，否则为 false。

（2）如果是非绑定型工作队列，计算 workqueue_struct 实例末尾 pool_workqueue 指针数量。

（3）分配工作队列 workqueue_struct 实例并初始化。对于非绑定型工作队列包含末尾的指针数组和工作队列属性实例。

（4）调用 **alloc_and_link_pwqs()**函数创建 pool_workqueue 实例（多个），以建立与 worker_pool 实例的关联。这是最重要的一个步骤，对于绑定和非绑定型工作队列有不同的操作，见下文。

（5）如果 flags 标记参数设置了 WQ_MEM_RECLAIM 标记位，则创建急救工作者，其内核线程调用函数为 **rescuer_thread()**。

（6）如果 flags 标记参数设置了 WQ_SYSFS 标记位，则将工作队列导出到 sysfs 文件系统。对于导出的工作队列用户可通过 sysfs 文件系统查看和修改 nice、CPU 亲和性等工作队列属性，请读者自行阅读相关源代码。

（7）将 workqueue_struct 实例添加到全局双链表 workqueues 末尾。

■创建 pool_workqueue 实例

下面重点来看一下 **alloc_and_link_pwqs()**函数的实现（/kernel/workqueue.c）：

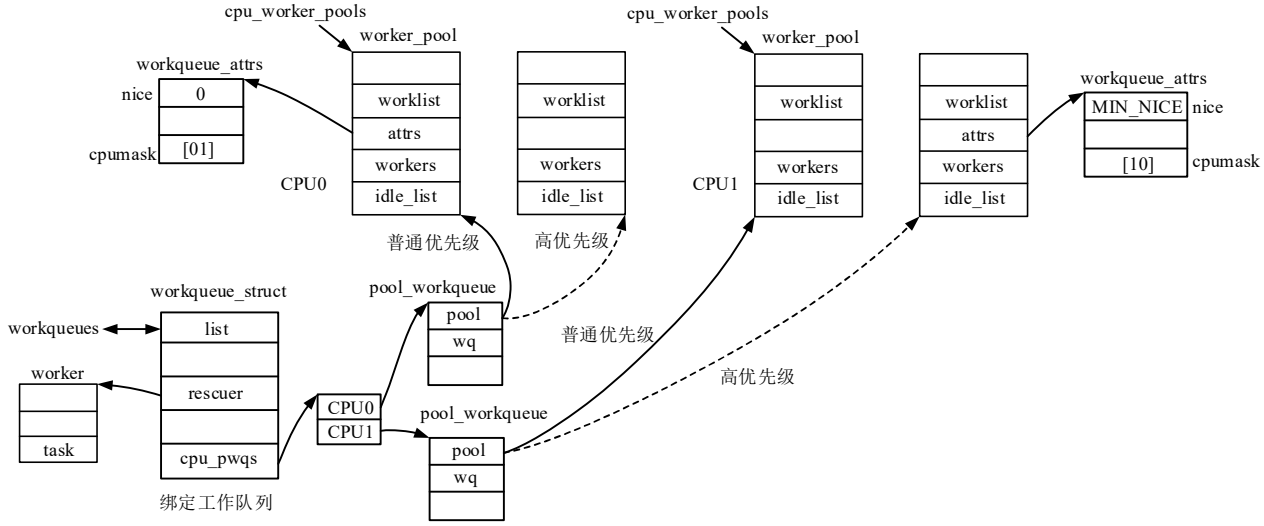
```
static int alloc_and_link_pwqs(struct workqueue_struct *wq)
{
    bool highpri = wq->flags & WQ_HIGHPRI;    /*是否是高优先级工作队列*/
    int cpu, ret;

    if (!(wq->flags & WQ_UNBOUND)) {     /*绑定型工作队列*/
        wq->cpu_pwqs = alloc_percpu(struct pool_workqueue);    /*分配 pool_workqueue 实例*/
        if (!wq->cpu_pwqs)
            return -ENOMEM;

        for_each_possible_cpu(cpu) {       /*关联各 CPU 核绑定型工作者池*/
            struct pool_workqueue *pwq =per_cpu_ptr(wq->cpu_pwqs, cpu);
            struct worker_pool *cpu_pools =per_cpu(cpu_worker_pools, cpu);    /*CPU 工作者池*/

            init_pwq(pwq, wq, &cpu_pools[highpri]);     /*初始化 pool_workqueue 实例*/

            mutex_lock(&wq->mutex);
            link_pwq(pwq);          /*将 pool_workqueue 实例添加到 wq->pwqs 双链表头部*/
            mutex_unlock(&wq->mutex);
        }
        return 0;
```

```
    } else if (wq->flags & __WQ_ORDERED) {     /*处理 ordered 非绑定型工作队列*/
        ret = apply_workqueue_attrs(wq, ordered_wq_attrs[highpri]);     /*/kernel/workqueue.c*/
        ...
        return ret;
    } else {                /*普通非绑定型工作队列*/
        return apply_workqueue_attrs(wq, unbound_std_wq_attrs[highpri]);
    }
}
```

alloc_and_link_pwqs()函数对绑定型工作队列和非绑定型工作队列分别采取不同的处理方式，如下所述。

●绑定型工作队列

对于绑定型工作队列，创建的 pool_workqueue 实例及关联的 worker_pool 实例如下图所示：



绑定型工作队列关联到各 CPU 核绑定型 worker_pool 实例，绑定型 worker_pool 实例由内核静态定义，并在 init_workqueues()初始化函数中初始化。

对于绑定型工作队列，alloc_and_link_pwqs()函数对每个 CPU 核为其分配一个 pool_workqueue 实例，通过此实例关联 CPU 核绑定型 worker_pool 实例。若工作队列为普通优先级则关联普通优先级 worker_pool 实例，高优先级工作队列关联高优先级 worker_pool 实例（见 init_pwq()函数）。

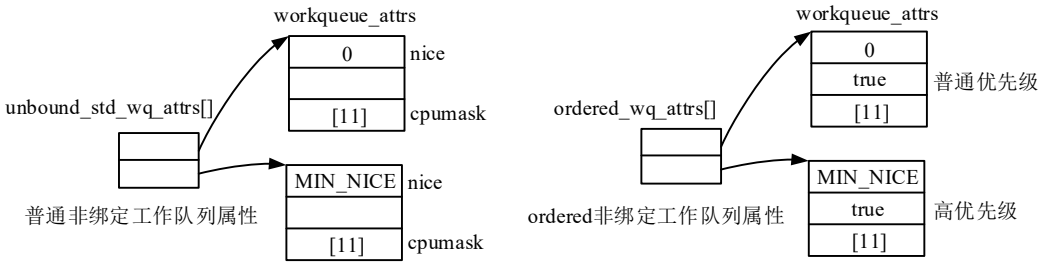
这里需要注意的是，普通优先级 worker_pool 实例关联的 workqueue_attrs 实例中，nice 值为 0，高优先级实例中为 MIN_NICE，CPU 位图中只标记了本 CPU 核，如上图所示。

link_pwq()函数将 pool_workqueue 实例添加到 workqueue_struct->pwqs 双链表头部。

若向绑定工作队列提交工作，将根据当前运行 CPU 核或指定 CPU 核对应的 pool_workqueue 实例，找到关联的 worker_pool 实例，由其接收并分配工作者执行工作。例如，假设当前提交工作的代码由 CPU1 执行（未特别指定 CPU 核），工作将提交到 CPU1 核的 worker_pool 实例。

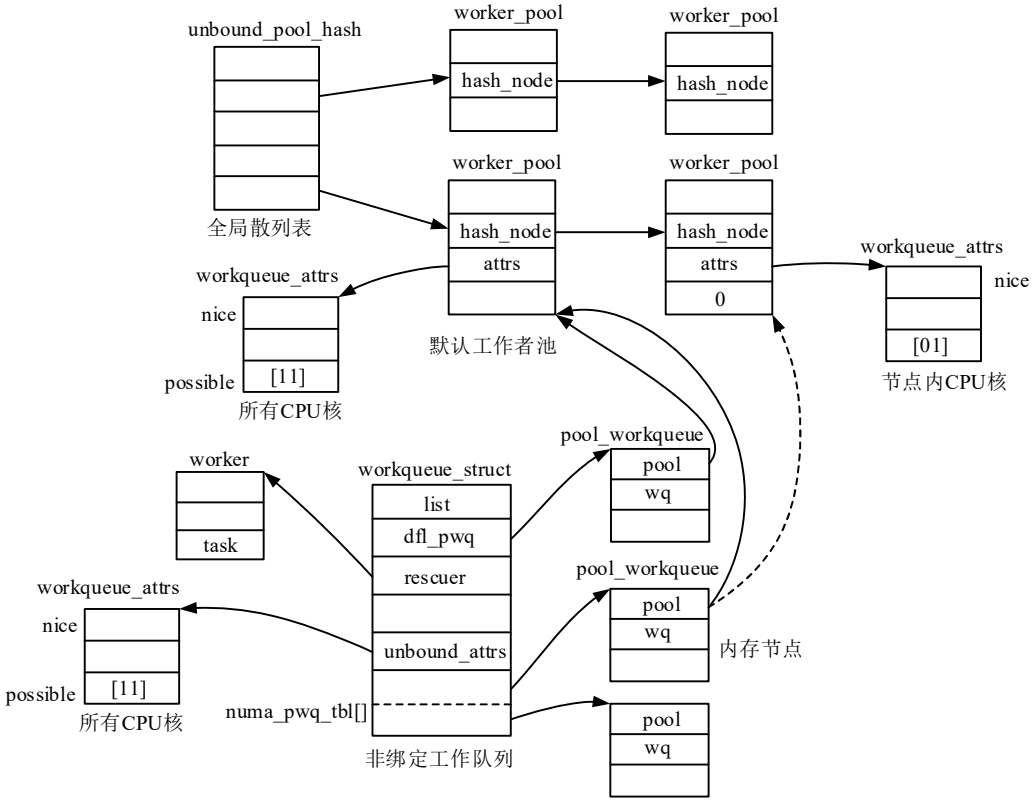
●非绑定型工作队列

对于非绑定型工作队列的处理要复杂一些。非绑定型工作队列又分为普通型和 ordered 型，创建工作队列时根据是否设置了__WQ_ORDERED 标记位区分。内核定义了这两种非绑定型工作队列属性，如下图所示：



unbound_std_wq_attrs[]表示普通非绑定工作队列属性指针数组，ordered_wq_attrs[]表示 ordered 型非绑定工作队列属性指针数组。在初始化函数 init_workqueues()中将为这两个指针数组分配并设置工作队列属性实例。ordered 型工作队列属性禁用内存节点亲和性。

alloc_and_link_pwqs()函数调用 apply_workqueue_attrs()函数为非绑定型工作队列分配 pool_workqueue 实例，并关联非绑定型 worker_pool 实例（需要时创建新实例）。对于普通非绑定型工作队列和 ordered 非绑定型工作队列，分别使用 unbound_std_wq_attrs[]和 ordered_wq_attrs[]表示的工作队列属性，创建的非绑定工作队列如下图所示。



apply_workqueue_attrs()函数内先根据传递的 workqueue_attrs 实例在全局散列表中查找是否有相同属性的工作者池，如果有则复用，没有则重新创建（见前面介绍的 get_unbound_pool()函数），将此工作者池实例通过 pool_workqueue 设为工作队列默认的工作者池。然后，对每个内存节点，查找或创建亲和性只包含本内存节点关联 CPU 核的工作者池，如果本内存节点包含所有可用 CPU 核，则复用默认工作者池，并通过 pool_workqueue 实例关联到工作队列。工作队列最后的 numa_pwq_tbl[]指针数组项就是指向各内存节点对应的 pool_workqueue 实例。

6.6.4 工作

内核中需要由工作队列完成的工作封装成 work_struct 结构体实例，通过某一工作队列提交。工作队列会选择工作者池，由工作池接收并选择/唤醒工作者执行工作。

如果是中断下半部使用工作队列机制，则在中断上半部工作中需要向工作队列提交工作。

1 创建工作

work_struct 结构体定义在/include/linux/workqueue.h 头文件内：

```
struct work_struct {
   atomic_long_t    data;       /*工作状态，标记等*/
   struct list_head    entry;   /*双链表成员，将实例添加到相应链表*/
   work_func_t func;          /*执行函数*/
#ifdef CONFIG_LOCKDEP
   struct lockdep_map lockdep_map;
#endif
};
```

work_struct 结构体成员简介如下：

●**entry**：双链表成员，将工作添加到工作队列机制的管理链表中。

●**func**：work_func_t 类型定义在/include/linux/workqueue.h 头文件，执行函数指针：

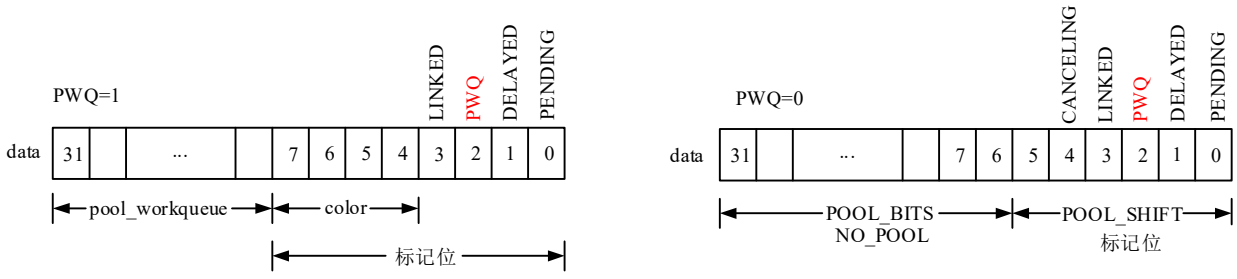
```
typedef    void (*work_func_t)(struct work_struct *work);
```

●**data**：表示工作状态标记等，各位语义如下（/include/linux/workqueue.h）：

```
#define work_data_bits(work)    ((unsigned long *)(&(work)->data))
enum {
      ...
   #ifdef CONFIG_DEBUG_OBJECTS_WORK
      ...
   #else
      WORK_STRUCT_COLOR_SHIFT  = 4,   /* color for workqueue flushing */
   #endif

      WORK_STRUCT_COLOR_BITS    = 4,

       /*以下是工作位于工作队列中时，data 成员布局（PWQ=1）*/
      WORK_STRUCT_PENDING     = 1 << WORK_STRUCT_PENDING_BIT,   /*工作挂起，bit0*/
      WORK_STRUCT_DELAYED     = 1 << WORK_STRUCT_DELAYED_BIT, /*工作被延迟，bit1*/
      WORK_STRUCT_PWQ         = 1 << WORK_STRUCT_PWQ_BIT,
                                          /*data 是否指向 pool_workqueue 实例，bit2*/
      WORK_STRUCT_LINKED      = 1 << WORK_STRUCT_LINKED_BIT,
                                  /*下一工作与本工作关联，需一起处理，bit3*/

#ifdef CONFIG_DEBUG_OBJECTS_WORK
      ...
#else
      WORK_STRUCT_STATIC  = 0,
```

```
#endif
    WORK_NR_COLORS      = (1 << WORK_STRUCT_COLOR_BITS) - 1,    /*0xF*/
    WORK_NO_COLOR       = WORK_NR_COLORS,   /*颜色数，0xF，有效颜色 0x0-0xE*/

    WORK_CPU_UNBOUND    = NR_CPUS,      /*工作不指定 CPU，默认由当前 CPU 处理*/
    WORK_STRUCT_FLAG_BITS   = WORK_STRUCT_COLOR_SHIFT +
                              WORK_STRUCT_COLOR_BITS,     /*8 位*/

    /*以下是工作不在工作队列中时，data 成员布局（PWQ=0）*/
    WORK_OFFQ_FLAG_BASE     = WORK_STRUCT_COLOR_SHIFT,

    __WORK_OFFQ_CANCELING   = WORK_OFFQ_FLAG_BASE,
    WORK_OFFQ_CANCELING     = (1 << __WORK_OFFQ_CANCELING),

    WORK_OFFQ_FLAG_BITS = 1,
    WORK_OFFQ_POOL_SHIFT    = WORK_OFFQ_FLAG_BASE + WORK_OFFQ_FLAG_BITS,
    WORK_OFFQ_LEFT      = BITS_PER_LONG - WORK_OFFQ_POOL_SHIFT,
    WORK_OFFQ_POOL_BITS = WORK_OFFQ_LEFT <= 31 ? WORK_OFFQ_LEFT : 31,
    WORK_OFFQ_POOL_NONE     = (1LU << WORK_OFFQ_POOL_BITS) - 1,

    /*常数*/
    WORK_STRUCT_FLAG_MASK = (1UL << WORK_STRUCT_FLAG_BITS) - 1,
    WORK_STRUCT_WQ_DATA_MASK = ~WORK_STRUCT_FLAG_MASK,
    WORK_STRUCT_NO_POOL     = (unsigned long)WORK_OFFQ_POOL_NONE <<
                              WORK_OFFQ_POOL_SHIFT,
                            /*高 27 位全置 1，表示不指示 worker_pool*/
    WORK_BUSY_PENDING   = 1 << 0,
    WORK_BUSY_RUNNING   = 1 << 1,

    WORKER_DESC_LEN     = 24,
};
```

data 成员布局如下图所示：



data 成员布局分两种情况，一是 work 在工作队列中（WORK_STRUCT_PWQ 标记位为 1），二是 work 不在工作队列中。

如果 WORK_STRUCT_PWQ_BIT 标记位为 1，data 高位表示 pool_workqueue 实例指针（高 24 位）。

由于此时 data 成员低 8 位用于标记（bit[4...7]表示颜色值），因此要求 pool_workqueue 实例 256 字节对齐。

如果 WORK_STRUCT_PWQ_BIT 标记位为 0（工作不在工作队列），data 高位（高 27 位）表示工作上次关联到的工作者池 worker_pool 实例的 ID 号，低位（5 位）表示标记位。

内核在/include/linux/workqueue.h 头文件定义了定义并初始化 work_struct 实例的宏，例如：

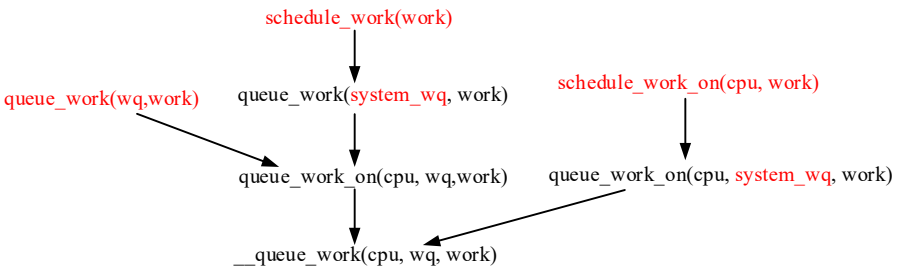
```
#define DECLARE_WORK(n, f)                    \      /*创建并初始化实例，实例名称为 n，执行函数 f()*/
    struct work_struct n = __WORK_INITIALIZER(n, f)

#define __WORK_INITIALIZER(n, f) {            \
    .data = WORK_DATA_STATIC_INIT(),    \
                        /*(WORK_STRUCT_NO_POOL | WORK_STRUCT_STATIC)*/
    .entry      = { &(n).entry, &(n).entry },      \
    .func = (f),                                   \
    __WORK_INIT_LOCKDEP_MAP(#n, &(n))     \
    }
```

初始化已定义 work_struct 实例的宏如下：

```
#define    INIT_WORK(_work, _func)        \
    __INIT_WORK((_work), (_func), 0)

#define __INIT_WORK(_work, _func, _onstack)     \
    do {                                        \
        __init_work((_work), _onstack);                 \      /*/kernel/workqueue.c*/
        (_work)->data = (atomic_long_t) WORK_DATA_INIT();  \
                                                    /*(WORK_STRUCT_NO_POOL)*/
        INIT_LIST_HEAD(&(_work)->entry);            \
        (_work)->func = (_func);                    \
    } while (0)
```

2 提交工作

工作 work_struct 实例初始化后需要提交到某一工作队列才会被执行。提交（调度）工作的接口函数定义在/include/linux/workqueue.h 头文件内，函数调用关系如下图所示：



●**bool queue_work(wq,work)**：将 work 提交到 **wq** 工作队列，不指定 CPU 核，默认由当前 CPU 核处理。

●**bool schedule_work(work)**：将 work 提交到内核 **system_wq** 工作队列，不指定 CPU 核。

●**bool schedule_work_on(cpu,work)**：将 work 提交到内核 **system_wq** 工作队列，使用 cpu 指定 CPU 核的工作者池接收并处理工作。

以上提交工作接口函数最终都是调用 **queue_work_on()**函数执行提交工作，这个函数本身也是一个接口函数，queue_work_on()函数定义如下：

```
bool queue_work_on(int cpu, struct workqueue_struct *wq,struct work_struct *work)
/*cpu：指定工作由哪个 CPU 核处理，WORK_CPU_UNBOUND 表示不指定，默认为当前 CPU 核；
*wq：工作队列指针，work：提交工作实例指针。
*/
{
    bool ret = false;
    unsigned long flags;
    local_irq_save(flags);      /*保存当前中断状态，并关中断*/

    if (!test_and_set_bit(WORK_STRUCT_PENDING_BIT, work_data_bits(work))) {
                                /*WORK_STRUCT_PENDING_BIT 标记位须为 0，并置位*/
        __queue_work(cpu, wq, work);    /*提交工作，/kernel/workqueue.c*/
        ret = true;
    }

    local_irq_restore(flags);       /*恢复中断状态*/
    return ret;
}
```

queue_work_on()函数执行工作如下：

（1）保存当前中断状态并关闭中断。

（2）检测 work->data 成员 WORK_STRUCT_PENDING_BIT 标记位，若为 0，则对其置位，然后调用__queue_work()函数提交工作。

（3）恢复中断状态，函数返回 true。

__queue_work()函数是在关中断状态下被调用的，函数定义如下：

```
static void   __queue_work(int cpu, struct workqueue_struct *wq,struct work_struct *work)
{
    struct pool_workqueue *pwq;
    struct worker_pool *last_pool;
    struct list_head *worklist;
    unsigned int work_flags;
    unsigned int req_cpu = cpu;

    WARN_ON_ONCE(!irqs_disabled());

    debug_work_activate(work);
```

```
    /*检测是否是由同一队列中工作提交的工作*/
    if (unlikely(wq->flags & __WQ_DRAINING) &&WARN_ON_ONCE(!is_chained_work(wq)))
        return;
retry:
    if (req_cpu == WORK_CPU_UNBOUND)          /*不指定 CPU 核*/
        cpu = raw_smp_processor_id();              /*默认使用当前 CPU 核*/

    if (!(wq->flags & WQ_UNBOUND))               /*绑定型工作队列*/
        pwq = per_cpu_ptr(wq->cpu_pwqs, cpu);   /*CPU 核对应的 pool_workqueue*/
    else
        pwq = unbound_pwq_by_node(wq, cpu_to_node(cpu));
                                                    /*CPU 核关联内存结点对应的 pool_workqueue*/
    last_pool = get_work_pool(work);
                                /*由 data 成员获取当前关联的工作者池或上次关联工作者池*/
    if (last_pool && last_pool != pwq->pool) {   /*上次关联工作者池与本次提交的不同*/
        struct worker *worker;
        spin_lock(&last_pool->lock);

        worker = find_worker_executing_work(last_pool, work);
                                    /*查找上次关联工作者池中工作者是否在执行当前工作*/
        if (worker && worker->current_pwq->wq == wq) {
            pwq = worker->current_pwq;           /*在执行，且 work 由同一队列提交*/
        } else {
            spin_unlock(&last_pool->lock);
            spin_lock(&pwq->pool->lock);
        }
    } else {      /*上次关联工作者池与本次提交的相同*/
        spin_lock(&pwq->pool->lock);
    }

    if (unlikely(!pwq->refcnt)) {        /*创建 pool_workqueue 时，引用计数设为 1*/
        if (wq->flags & WQ_UNBOUND) {
            spin_unlock(&pwq->pool->lock);
            cpu_relax();
            goto retry;
        }
        ...
    }

    trace_workqueue_queue_work(req_cpu, pwq, work);
```

```
        if (WARN_ON(!list_empty(&work->entry))) {       /*work 已经提交，返回*/
            spin_unlock(&pwq->pool->lock);
            return;
        }

        pwq->nr_in_flight[pwq->work_color]++;          /*当前颜色值工作数量加 1*/
        work_flags = work_color_to_flags(pwq->work_color);   /*颜色值写入 work_struct.data 成员*/

        if (likely(pwq->nr_active < pwq->max_active)) {     /*活跃工作数量未达到最大值*/
            trace_workqueue_activate_work(work);
            pwq->nr_active++;          /*活跃工作数量加 1，工作处理完后减 1*/
            worklist = &pwq->pool->worklist;       /*worker_pool.worklist 双链表*/
        } else {                       /*活跃工作数量达到或超过最大值*/
            work_flags |= WORK_STRUCT_DELAYED;   /*工作延时*/
            worklist = &pwq->delayed_works;
                                /*pool_workqueue.delayed_works 双链表*/
        }

        insert_work(pwq, work, worklist, work_flags);     /*work 添加到双链表，/kernel/workqueue.c*/

        spin_unlock(&pwq->pool->lock);
    }
```

__queue_work()函数需要查找正确的 pool_workqueue 实例。如果 pool_workqueue 实例中当前活跃工作数量还没有达到最大值，则调用 **insert_work()**函数添加工作到工作者池 worker_pool.worklist 双链表，否则添加到 pool_workqueue.delayed_works 双链表（延时执行工作）。

这里需要注意的是一个工作可能同时多次被提交，例如，发生了一个中断，某个 CPU 核在处理该中断，在中断上半部处理程序中正在提交工作或工作已经在执行了，同一个中断又发生了，此时中断处理程序又试图提交工作，就会产生多次提交的情况。因此在提交工作时需要判断工作是否已经提交或正在被执行。

insert_work()函数用于将工作添加到管理双链表中，定义如下：

```
static void insert_work(struct pool_workqueue *pwq, struct work_struct *work,     \
                                                    struct list_head *head, unsigned int extra_flags)
/*head：双链表头，extra_flags：标记*/
{
    struct worker_pool *pool = pwq->pool;     /*工作者池实例指针*/

    set_work_pwq(work, pwq, extra_flags);   /*设置 work->data 成员，/kernel/workqueue.c*/
    list_add_tail(&work->entry, head);       /*工作 work_struct 实例添加 head 双链表末尾*/
    get_pwq(pwq);     /*pwq->refcnt++*/
    smp_mb();
```

```
    if (__need_more_worker(pool))       /*worker_pool->nr_running 为 0 返回真*/
        wake_up_worker(pool);           /*唤醒工作者*/
}
```

insert_work()函数首先调用 set_work_pwq()设置 work->data 成员，函数定义如下：

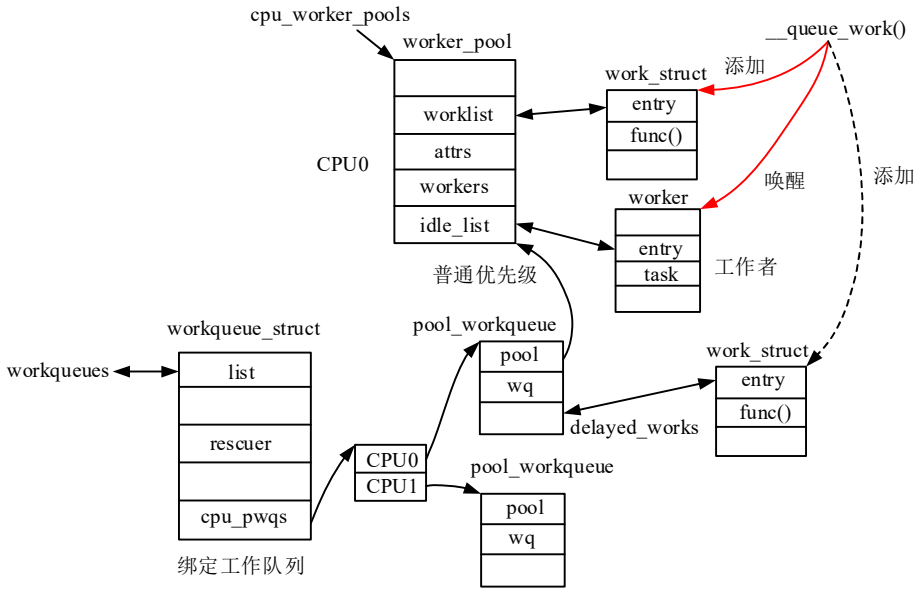
```
static void set_work_pwq(struct work_struct *work, struct pool_workqueue *pwq,
                                                    unsigned long extra_flags)
{
    set_work_data(work, (unsigned long)pwq,
                WORK_STRUCT_PENDING | WORK_STRUCT_PWQ | extra_flags);
                                        /*高位为 pool_workqueue 指针，低位为标记*/
}
```

insert_work()函数随后将 work_struct 实例添加到 worker_pool.worklist 或 pool_workqueue.delayed_works 双链表末尾，判断工作者池中当前正在运行的工作者是否为 0，如果为 0（没有正在运行的工作线程）则调用 wake_up_worker(pool)函数唤醒一个空闲工作者。

wake_up_worker()函数定义如下：

```
static void wake_up_worker(struct worker_pool *pool)
{
    struct worker *worker = first_idle_worker(pool);  /*空闲工作者链表中第一个成员*/

    if (likely(worker))
        wake_up_process(worker->task);     /*唤醒工作者*/
}
```

下图示意了提交工作执行的流程（由 CPU0 提交到绑定型工作队列）：



3 延时工作

前面介绍的提交工作操作将立即提交工作。内核还提供了在指定延期时间后提交工作的机制，称之为延时工作。延时工作只是前面介绍工作的一种特例。

■创建延时工作

延时工作由 delayed_work 结构体表示，定义如下（/include/linux/workqueue.h）：

```
struct delayed_work {
    struct work_struct    work;       /*工作实例*/
    struct timer_list    timer;        /*定时器*/

    struct workqueue_struct *wq;      /*工作队列*/
    int cpu;              /*绑定 CPU 核*/
};
```

创建并初始化延时工作的宏定义如下（/include/linux/workqueue.h）：

```
#define DECLARE_DELAYED_WORK(n, f)                    \
    struct delayed_work n = __DELAYED_WORK_INITIALIZER(n, f, 0)

#define __DELAYED_WORK_INITIALIZER(n, f, tflags) {      \
    .work = __WORK_INITIALIZER((n).work, (f)),          \
    .timer = __TIMER_INITIALIZER(delayed_work_timer_fn,     \   /*定时器执行函数*/
                    0, (unsigned long)&(n),      \
                    (tflags) | TIMER_IRQSAFE),          \
    }
```

初始化静态定义的延时工作实例的宏如下：

```
#define    INIT_DELAYED_WORK(_work, _func)          \
    __INIT_DELAYED_WORK(_work, _func, 0)

#define __INIT_DELAYED_WORK(_work, _func, _tflags)      \
    do {                                    \
        INIT_WORK(&(_work)->work, (_func));             \
        __setup_timer(&(_work)->timer, delayed_work_timer_fn, \   /*定时器执行函数*/
                    (unsigned long)(_work),             \
                    (_tflags) | TIMER_IRQSAFE);         \
    } while (0)
```

需要注意的是定时器到期执行函数为 **delayed_work_timer_fn()**，在此函数中将执行工作的提交，也就是在将来指定某一时刻将工作提交上去。

■调度延时工作

延时工作初始化后需要调度执行，调度 delayed_work 实例的接口函数如下：

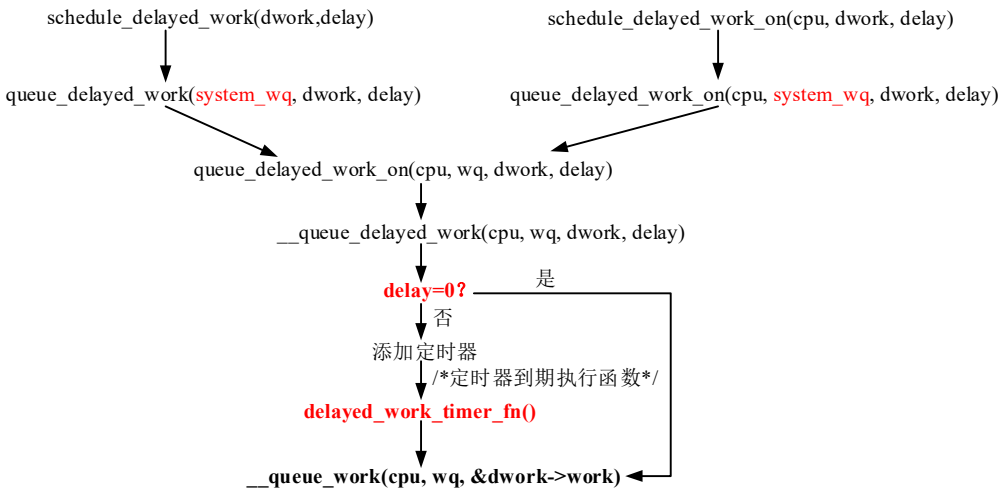
●bool **queue_delayed_work**(struct workqueue_struct *wq,struct delayed_work *dwork,unsigned long delay)：将 dwork 指向延时工作在延时 delay（单位是 jiffies，即内核节拍数）后，提交到 wq 工作队列，由当前 CPU 核关联工作者池执行。

●bool **queue_delayed_work_on**(int cpu, struct workqueue_struct *wq,struct delayed_work *work, unsigned long delay)：将 dwork 指向延时工作在延时 delay（单位是 jiffies，即内核节拍数）后，提交到 wq 工作队列，由 cpu 指定 CPU 核关联工作者池执行。

●bool **schedule_delayed_work**(struct delayed_work *dwork, unsigned long delay)：将 dwork 指向延时工作在延时 delay（单位是 jiffies，即内核节拍数）后，提交到 **system_wq** 工作队列，由当前 CPU 核关联工作者池执行。

●bool **schedule_delayed_work_on**(int cpu, struct delayed_work *dwork,unsigned long delay)：将 dwork 指向延时工作在延时 delay（单位是 jiffies，即内核节拍数）后，提交到 **system_wq** 工作队列，由 cpu 指定 CPU 核关联工作者池执行。

以上调度延时工作接口函数调用关系如下图所示：



所有接口函数最终汇集到__queue_delayed_work(cpu, wq,dwork, delay)函数。函数内判断如果延迟时间 delay 为 0，表示延时已到需要立即提交工作，则调用前面介绍的__queue_work()函数提交延时工作。如果 delay 不为 0，则将延时工作中的定时器实例注册到内核，函数返回。内核在该定时器到期时将调用定时器执行函数，即 delayed_work_timer_fn()函数，此函数内直接调用__queue_work()函数提交延时工作，函数源代码请读者自行阅读。

4 其它操作接口

内核在/kernel/workqueue.c 文件内还定义了工作的其它操作函数，例如（源代码请读者自行阅读）：

●bool **flush_work**(struct work_struct *work)：冲刷工作，等待工作执行完成。

●bool **cancel_work_sync(work)**：取消工作，在注销设备驱动等时机需要注销工作，取消工作前需要确保工作被执行完。

●bool **flush_delayed_work**(struct delayed_work *dwork)：立即提交延时工作（删除定时器），并等待工作完成。

●bool **cancel_delayed_work**(struct delayed_work *dwork)：取消延时工作，不等待其完成，可在中断处理程序中调用。

●bool **cancel_delayed_work_sync**(struct delayed_work *dwork)：取消延时工作，并等待其完成。

6.6.5 处理工作

工作提交到工作者池后，将由工作者线程执行，内核还定义了冲刷工作队列的接口函数等。本小节介绍工作队列机制对工作的处理。

1 工作者线程

在前面的提交工作中，如果工作池当前运行的工作者为 0（准确地说是 nr_running 值为 0，不一定工作者为 0），则唤醒一个空闲工作者执行工作。由前面介绍的创建工作者的 create_worker()函数可知，工作者就是一个内核线程，线程内将调用 **worker_thread()**函数，此函数用于执行工作。

worker_thread()函数定义如下（/kernel/workqueue.c）：

```
static int worker_thread(void *__worker)
{
    struct worker *worker = __worker;        /*工作者*/
    struct worker_pool *pool = worker->pool;      /*工作者池实例*/

    /*标记当前线程为工作者线程*/
    worker->task->flags |= PF_WQ_WORKER;     /*工作者线程标记*/
woke_up:            /*睡眠唤醒后从此处开始执行*/
    spin_lock_irq(&pool->lock);

    if (unlikely(worker->flags & WORKER_DIE)) {       /*检测 WORKER_DIE 标记位*/
                                    /*需要注销的工作者，由 destroy_worker()函数设置此标记位*/
        spin_unlock_irq(&pool->lock);
        WARN_ON_ONCE(!list_empty(&worker->entry));
        worker->task->flags &= ~PF_WQ_WORKER;        /*清标记位*/

        set_task_comm(worker->task, "kworker/dying");
        ida_simple_remove(&pool->worker_ida, worker->id);
        worker_detach_from_pool(worker, pool);        /*从工作者池中移除工作者*/
        kfree(worker);          /*释放 worker 实例*/
        return 0;             /*函数返回后，内核线程就会退出*/
    }

    worker_leave_idle(worker);
            /*工作者退出空闲状态，清 WORKER_IDLE 标记位，从空闲双链表移出，nr_idle--*/
recheck:
    if (!need_more_worker(pool))      /*再次检测是否需要更多工作者*/
        goto sleep;     /*不需要更多工作者，当前线程睡眠*/
```

```
    /*需要更多工作者，当前线程继续执行*/
    if (unlikely(!may_start_working(pool)) && manage_workers(worker))
                                                    /*若 pool->nr_idle 为 0，动态创建工作者*/
        goto recheck;

    WARN_ON_ONCE(!list_empty(&worker->scheduled));     /*worker->scheduled 需为空*/

    worker_clr_flags(worker, WORKER_PREP | WORKER_REBOUND);
                                                    /*清工作者标记位，pool->nr_running++*/
    do {        /*遍历工作者池 worklist 双链表*/
        struct work_struct *work =list_first_entry(&pool->worklist,struct work_struct, entry);

        if (likely(!(*work_data_bits(work) & WORK_STRUCT_LINKED))) {
                                                    /*work 实例与下一实例没有关联，直接处理*/
            process_one_work(worker, work);     /*/kernel/workqueue.c*/
            if (unlikely(!list_empty(&worker->scheduled)))   /*worker->scheduled 链表不为空*/
                process_scheduled_works(worker);   /*逐个执行 worker->scheduled 链表中工作*/
        } else {
            /*work 实例关联下一实例，添加到 worker->scheduled 双链表*/
            move_linked_works(work, &worker->scheduled, NULL);
            process_scheduled_works(worker);          /*逐个处理 worker->scheduled 链表中工作*/
        }
    } while (keep_working(pool));     /*worklist 双链表不为空，且 pool->nr_running 不大于 1，继续*/

    worker_set_flags(worker, WORKER_PREP);     /*设置工作者标记位，pool->nr_running--*/
sleep:      /*工作者进入睡眠*/
    worker_enter_idle(worker);      /*进入空闲状态*/
    __set_current_state(TASK_INTERRUPTIBLE);     /*设置工作者为可中断睡眠状态*/
    spin_unlock_irq(&pool->lock);
    schedule();     /*进程调度*/
    goto woke_up;     /*线程唤醒再次被调度执行时，跳转至 woke_up 处运行*/
}
```

worker_thread()函数执行流程简述如下：

（1）设置工作者线程的 PF_WQ_WORKER 标记位，标记当前线程为工作者线程，后面介绍的与调度器交互会用到此标记。

（2）判断工作者是否设置了 WORKER_DIE 标记位，若是则清线程 PF_WQ_WORKER 标记位，将工作者从工作者池中移出并释放，worker_thread()函数返回，工作者线程将退出。若没有设置 WORKER_DIE 标记位继续往下执行。

（3）再次检测是否需要工作者执行工作，是则继续往下执行，否则线程进入睡眠。

（4）若 pool->nr_idle 值为 0，表示没有空闲工作者了，调用 **manage_workers(worker)**函数动态创建工作者，见下文。

（5）调用 worker_clr_flags()函数清工作者 WORKER_PREP 和 WORKER_REBOUND 标记位，并视情将 pool->nr_running 值加 1。

特别注意，如果工作者设置了 WORKER_UNBOUND 或 WORKER_CPU_INTENSIVE 标记位，在 worker_clr_flags()函数中 pool->nr_running 值将不加 1，也就是说当前运行工作者被忽略，在提交工作时，不视其为正在运行的工作者。

由上可知，对于绑定型工作者池，通常只有一个工作者在工作（唤醒工作者时 nr_running 值加 1，睡眠时减 1），而对于非绑定型工作者池和 CPU 密集型工作队列关联的工作者池，可以有多个工作者同时在工作。

（6）从 pool->worklist 双链表取出一个工作，如果是普通工作则调用 process_one_work()函数处理单个工作。若工作设置了 WORK_STRUCT_LINKED 标记位，则调用 move_linked_works()函数取出从本工作开始（可以有多个连续设置 WORK_STRUCT_LINKED 标记位的工作），直至第一个没有设置此标记位的工作，添加到 worker->scheduled 双链表，然后调用 process_scheduled_works(worker)函数处理此链表中工作，此函数内逐个对 worker->scheduled 链表中工作调用 process_one_work()函数进行处理。

（7）若 pool->worklist 双链表不为空，且 pool->nr_running 不大于 1，则循环执行步骤（6），否则往下执行。

（8）调用 worker_set_flags()函数对工作添加 WORKER_PREP 标记位，视情对 pool->nr_running 值减 1（同步骤（5））。

（9）工作者线程设为可中断睡眠状态，执行进程调度（睡眠），唤醒后，跳至步骤（2）开始执行。

■处理单个工作

处理工作时，总是调用 **process_one_work**(worker, work)函数执行单个工作，函数定义如下：

```
static void process_one_work(struct worker *worker, struct work_struct *work)
__releases(&pool->lock)
__acquires(&pool->lock)
{
    struct pool_workqueue *pwq = get_work_pwq(work);   /*pool_workqueue 实例*/
    struct worker_pool *pool = worker->pool;
    bool cpu_intensive = pwq->wq->flags & WQ_CPU_INTENSIVE;      /*工作队列标记*/
    int work_color;
    struct worker *collision;
#ifdef CONFIG_LOCKDEP
   ...
#endif
    /*确保绑定型工作队列中工作者，在正确的 CPU 核上运行*/
    WARN_ON_ONCE(!(pool->flags & POOL_DISASSOCIATED) &&
                                        raw_smp_processor_id() != pool->cpu);
    collision = find_worker_executing_work(pool, work);   /*工作是否在其它工作者中执行*/
    if (unlikely(collision)) {
        move_linked_works(work, &collision->scheduled, NULL);   /*如果是则移到 scheduled 链表*/
        return;
    }
```

```
debug_work_deactivate(work);
hash_add(pool->busy_hash, &worker->hentry, (unsigned long)work);
                                                    /*工作添加到工作者池忙工作者散列表*/
worker->current_work = work;        /*当前工作*/
worker->current_func = work->func;
worker->current_pwq = pwq;
work_color = get_work_color(work);     /*获取工作颜色值*/

list_del_init(&work->entry);          /*工作从双链表中移出*/

if (unlikely(cpu_intensive))         /*CPU 密集型工作队列*/
      worker_set_flags(worker, WORKER_CPU_INTENSIVE);
                  /*设置工作者 WORKER_CPU_INTENSIVE 标记位，pool->nr_running--*/

if (need_more_worker(pool))     /*是否需要更多的工作者*/
      wake_up_worker(pool);        /*唤醒一个空闲工作者*/

set_work_pool_and_clear_pending(work, pool->id);       /*设置工作 data 成员*/

spin_unlock_irq(&pool->lock);

lock_map_acquire_read(&pwq->wq->lockdep_map);
lock_map_acquire(&lockdep_map);
trace_workqueue_execute_start(work);

worker->current_func(work);   /*调用工作执行函数*/

/*工作处理完之后*/
trace_workqueue_execute_end(work);
lock_map_release(&lockdep_map);
lock_map_release(&pwq->wq->lockdep_map);

if (unlikely(in_atomic() || lockdep_depth(current) > 0)) {
      ...
}

cond_resched_rcu_qs();
spin_lock_irq(&pool->lock);

if (unlikely(cpu_intensive))
      worker_clr_flags(worker, WORKER_CPU_INTENSIVE);
                  /*清工作者 WORKER_CPU_INTENSIVE 标记位，pool->nr_running++*/
```

```
        hash_del(&worker->hentry);          /*工作者从忙散列表中移出*/
        worker->current_work = NULL;
        worker->current_func = NULL;
        worker->current_pwq = NULL;
        worker->desc_valid = false;
        pwq_dec_nr_in_flight(pwq, work_color);      /*工作处理完成后工作，详见下文冲刷工作队列*/
    }
```

process_one_work()函数将 work 从其所在双链表中移出，如果工作队列设置了 WQ_CPU_INTENSIVE 标记位，则设置工作者 WORKER_CPU_INTENSIVE 标记位，且 pool->nr_running 值减 1（忽略本线程），然后调用 work 实例中的 **func()**执行函数处理工作。如果需要则清零工作者 WQ_CPU_INTENSIVE 标记位，pool->nr_running 值加 1。最后调用 pwq_dec_nr_in_flight()函数执行完成后的工作，详见下文冲刷工作队列。

■动态创建工作者

在初始化工作者池时，会为工作者池创建一个初始的工作者。工作者唤醒运行后（空闲工作者数减 1），在调用函数 worker_thread()中，将检测 pool->nr_idle 是否为 0，为 0 则表示工作者池中没有空闲工作者了，这时将调用 manage_workers(worker)函数动态创建空闲工作者。

manage_workers(worker)函数定义如下：

```
static bool manage_workers(struct worker *worker)
/*worker：当前运行的工作者*/
{
    struct worker_pool *pool = worker->pool;        /*工作者池*/
    if (!mutex_trylock(&pool->manager_arb))         /*获取锁*/
        return false;
    pool->manager = worker;

    maybe_create_worker(pool);      /*可能创建工作者*/

    pool->manager = NULL;
    mutex_unlock(&pool->manager_arb);
    return true;
}
```

maybe_create_worker()函数定义如下：

```
static void maybe_create_worker(struct worker_pool *pool)
{
restart:
    spin_unlock_irq(&pool->lock);
    mod_timer(&pool->mayday_timer, jiffies + MAYDAY_INITIAL_TIMEOUT);   /*10 毫秒*/

    while (true) {
```

```
        if (create_worker(pool) || !need_to_create_worker(pool))
            break;                    /*nr_running 和 nr_idle 都为 0，且 pool->worklist 链表非空*/

        schedule_timeout_interruptible(CREATE_COOLDOWN);
                        /*设置定时器（1 秒），并进行进程调度，定时器到期唤醒当前进程*/
        if (!need_to_create_worker(pool))       /*唤醒后此处开始执行*/
            break;
    }   /*while 循环结束*/

    del_timer_sync(&pool->mayday_timer);      /*移除定时器*/
    spin_lock_irq(&pool->lock);
    if (need_to_create_worker(pool))
        goto restart;               /*还需要创建工作者*/
}
```

maybe_create_worker()函数首先修改工作者池 mayday_timer 定时器到期时间为 10 毫秒以后，然后进入一个循环。循环内首先创建一个工作者，若成功则跳出循环。

若循环内 create_worker(pool)函数创建工作者失败，且需要创建工作者，则当前线程进入可中断睡眠状态，并设置定时器，1 秒后，定时器到期唤醒本线程。线程唤醒后再判断是否要创建工作者，若不需要跳出循环，需要则继续循环。

need_to_create_worker(pool)函数用于判断是否需要创建工作者，当工作者池 nr_running 和 nr_idle 成员都为 0，且 pool->worklist 链表非空时，函数返回 true，表示需要创建工作者。

maybe_create_worker()函数跳出循环后，表示已经创建了一个空闲工作者，将移除 mayday_timer 定时器，再判断是否要创建工作者，若不需要则函数返回，否则继续执行 maybe_create_worker()函数。

工作者池 mayday_timer 定时器就是用来处理直接调用 create_worker(pool)函数创建工作者失败的情形，它的到期时间是 10 毫秒之后，而当前线程睡眠要 1 秒后才唤醒，因此定时器到期函数有足够的时间来处理创建工作者失败的情形，在处理完成后，当前线程才会被唤醒。

mayday_timer 定时器到期执行函数为 pool_mayday_timeout()（见 init_worker_pool()函数），定义如下：

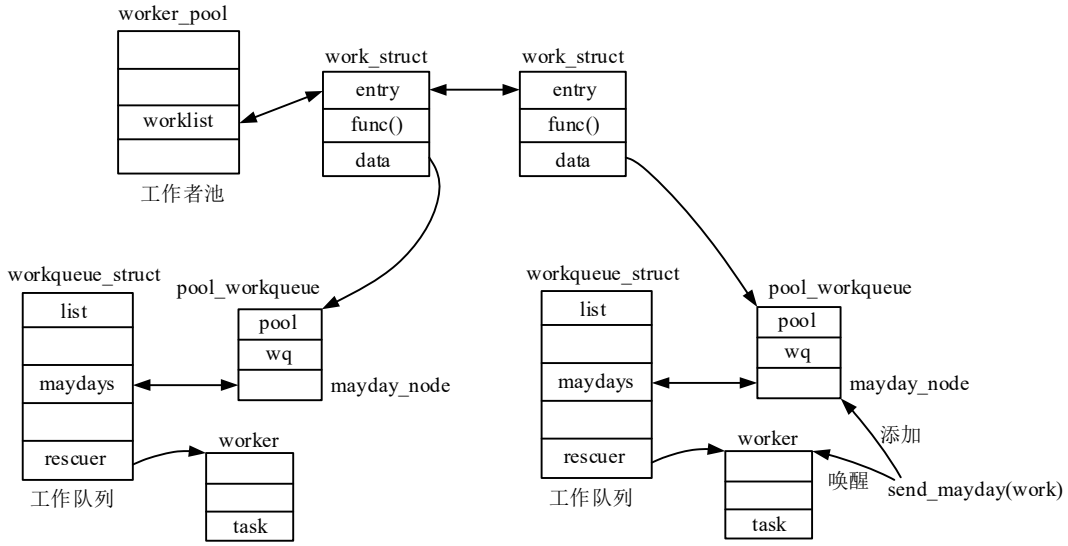
```
static void pool_mayday_timeout(unsigned long __pool)
/* __pool：worker_pool 指针*/
{
    struct worker_pool *pool = (void *)__pool;
    struct work_struct *work;

    spin_lock_irq(&pool->lock);
    spin_lock(&wq_mayday_lock);          /* for wq->maydays */

    if (need_to_create_worker(pool)) {      /*若需要创建工作者*/
        list_for_each_entry(work, &pool->worklist, entry)      /*遍历 pool->worklist 双链表中工作*/
            send_mayday(work);
              /*将工作关联 pool_workqueue 实例添加到 wq.maydays 链表，并唤醒急救工作者*/
    }
```

```
        spin_unlock(&wq_mayday_lock);
        spin_unlock_irq(&pool->lock);
        mod_timer(&pool->mayday_timer, jiffies + MAYDAY_INTERVAL);     /*定时器 10 毫秒后到期*/
    }
```

pool_mayday_timeout()函数判断是否需要创建工作者，如果是则扫描 pool->worklist 双链表中工作，对每个工作调用 send_mayday(work)函数。

```
    static void send_mayday(struct work_struct *work)
    {
        struct pool_workqueue *pwq = get_work_pwq(work);       /*pool_workqueue 实例，work.data*/
        struct workqueue_struct *wq = pwq->wq;          /*workqueue_struct 实例*/

        lockdep_assert_held(&wq_mayday_lock);

        if (!wq->rescuer)        /*工作队列没有急救工作者，直接返回*/
            return;

        if (list_empty(&pwq->mayday_node)) {
                /*将 pool_workqueue 实例添加到 wq.maydays 链表，并唤醒急救工作者*/
            get_pwq(pwq);
            list_add_tail(&pwq->mayday_node, &wq->maydays);
            wake_up_process(wq->rescuer->task);        /*唤醒急救工作者*/
        }
    }
```

send_mayday()函数执行的效果如下图所示，工作关联的 pool_workqueue 实例添加到 wq.maydays 双链表，并唤醒对应工作队列中的急救工作者。工作者池中的工作可能来自于不同的工作队列，因此可能唤醒多个工作队列的急救工作者。



急救工作者就是在创建工作者失败时，应急处理工作者池中的工作，详见下面的急救工作者线程调用

函数。

●急救工作者

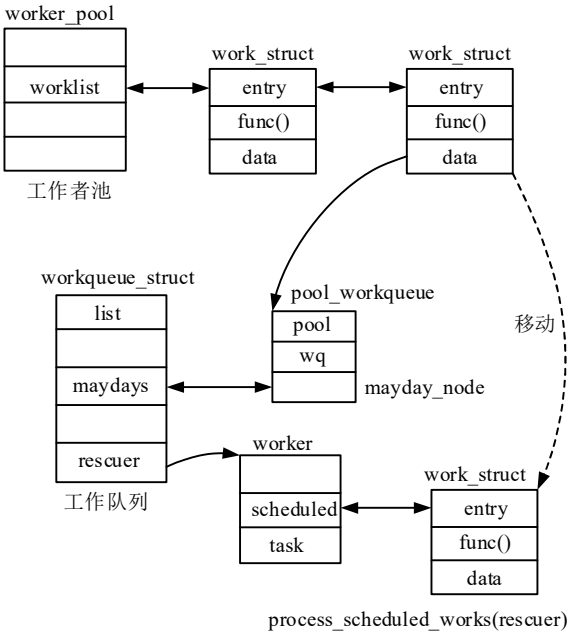
由创建工作队列的函数可知，工作队列急救工作者线程调用函数为 **rescuer_thread()**，函数定义如下：

```
static int rescuer_thread(void *__rescuer)
/*__rescuer：指向急救工作者*/
{
    struct worker *rescuer = __rescuer;
    struct workqueue_struct *wq = rescuer->rescue_wq;       /*工作队列*/
    struct list_head *scheduled = &rescuer->scheduled;      /*急救工作者中挂起工作双链表*/
    bool should_stop;

    set_user_nice(current, RESCUER_NICE_LEVEL);       /*设置工作者线程 nice 值*/
    rescuer->task->flags |= PF_WQ_WORKER;         /*标记为工作者线程*/
repeat:
    set_current_state(TASK_INTERRUPTIBLE);     /*设为可中断睡眠状态*/
    should_stop = kthread_should_stop();         /*是否要停止线程*/

    spin_lock_irq(&wq_mayday_lock);

    while (!list_empty(&wq->maydays)) {/*wq->maydays 双链表不为空，遍历 pool_workqueue 实例*/
        struct pool_workqueue *pwq = list_first_entry(&wq->maydays,
                                                      struct pool_workqueue, mayday_node);
        struct worker_pool *pool = pwq->pool;
        struct work_struct *work, *n;

        __set_current_state(TASK_RUNNING);       /*当前线程设为可运行状态*/
        list_del_init(&pwq->mayday_node);       /*pool_workqueue 实例从 wq->maydays 链表移出*/
        spin_unlock_irq(&wq_mayday_lock);
        worker_attach_to_pool(rescuer, pool);       /*急救工作者添加到工作者池中*/
        spin_lock_irq(&pool->lock);
        rescuer->pool = pool;

        WARN_ON_ONCE(!list_empty(scheduled));
        list_for_each_entry_safe(work, n, &pool->worklist, entry)
            if (get_work_pwq(work) == pwq)
                move_linked_works(work, scheduled, &n);
                    /*pool->worklist 链表中工作迁移到 rescuer->scheduled 双链表*/

        if (!list_empty(scheduled)) {
            process_scheduled_works(rescuer);
                        /*处理 rescuer->scheduled 双链表中挂起工作*/
```

```
                if (need_to_create_worker(pool)) {       /*仍需要创建工作者*/
                    spin_lock(&wq_mayday_lock);
                    get_pwq(pwq);
                    list_move_tail(&pwq->mayday_node, &wq->maydays);
                                        /*pool_workqueue 再次添加到 wq->maydays 双链表*/
                    spin_unlock(&wq_mayday_lock);
                }
            }

            put_pwq(pwq);

            if (need_more_worker(pool))
                wake_up_worker(pool);       /*尝试唤醒空闲工作者*/

            rescuer->pool = NULL;
            spin_unlock_irq(&pool->lock);
            worker_detach_from_pool(rescuer, pool);       /*断开急救工作者与工作者池关联*/
            spin_lock_irq(&wq_mayday_lock);
        }      /*while (!list_empty(&wq->maydays))结束*/

        /*wq->maydays 链表为空，急救工作者睡眠等待*/
        spin_unlock_irq(&wq_mayday_lock);

        if (should_stop) {       /*急救工作者线程要退出*/
            __set_current_state(TASK_RUNNING);
            rescuer->task->flags &= ~PF_WQ_WORKER;
            return 0;
        }

        WARN_ON_ONCE(!(rescuer->flags & WORKER_NOT_RUNNING));
        schedule();
        goto repeat;
    }
```

rescuer_thread()函数遍历 wq->maydays 双链表中 pool_workqueue 实例。对每个实例，使急救工作者添加到 pool_workqueue 实例关联的工作者池，取出工作者池 pool->worklist 链表中经 pool_workqueue 实例提交的工作，迁移到 rescuer->scheduled 双链表，如下图所示。然后，调用 process_scheduled_works(rescuer)函数处理急救工作者 rescuer->scheduled 双链表中工作，随后尝试唤醒工作者池中空闲工作者（其它路径可能创建），急救工作者从工作者池中移出。rescuer_thread()函数最后遍历完 wq->maydays 双链表中实例后进入睡眠等待。

简单地说，急救工作者就是临时代替工作者池中工作者，处理由各 pool_workqueue 实例提交的工作。



2 与调度器交互

在工作者线程中将逐个从链表中取出工作执行，如果某个工作执行函数 func()进入了睡眠，此时工作者将不能执行后面的工作了。因此在工作者线程进入睡眠后需要考虑唤醒其它工作者继续执行工作。

这就需要调度器的介入来解决这个问题了，看下面调度器实现__schedule()函数代码片断：

```
static void __sched __schedule(void)
{
    ...
    if (prev->state && !(preempt_count() & PREEMPT_ACTIVE))
    /*进程处于非可运行状态，且不是内核抢占引发的调度*/
    {
        if (unlikely(signal_pending_state(prev->state, prev)))   /*有挂起信号，不睡眠*/
        {
            prev->state = TASK_RUNNING;
        }
        else   /*进程不是可运行状态且没有挂起信号，从就绪队列移出*/
        {
            deactivate_task(rq, prev, DEQUEUE_SLEEP);        /*从就绪队列移出进程*/
            prev->on_rq = 0;          /*标记进程不在就绪队列中*/

            if (prev->flags & PF_WQ_WORKER)   /*如果当前进程是工作者线程*/
            {
                struct task_struct *to_wakeup;
                to_wakeup = wq_worker_sleeping(prev, cpu);
                                        /*需唤醒的空闲工作者线程，/kernel/workqueue.c*/
                if (to_wakeup)
```

```
                    try_to_wake_up_local(to_wakeup);
                                        /*唤醒空闲工作者线程，/kernel/sched/core.c*/
                }
            }
            ...
        }
        ...
    }
```

工作者线程可能因工作 func()函数进入睡眠，也可能是真的无事可做了进入睡眠，睡眠就意味着将发起进程调度。在调度器__schedule()函数中，如果前一进程不是可运行状态，没有挂起信号，且不是由于内核抢占引发的进程调度，前一进程将从就绪队列移出。

如果前一进程是工作者线程，__schedule()函数将调用 wq_worker_sleeping()函数判断是否需要唤醒一个空闲工作者线程，若需要则调用 try_to_wake_up_local()函数唤醒空闲工作者线程。

wq_worker_sleeping()函数定义如下（/kernel/workqueue.c）：

```
struct task_struct *wq_worker_sleeping(struct task_struct *task, int cpu)
{
    struct worker *worker = kthread_data(task), *to_wakeup = NULL;
    struct worker_pool *pool;

    if (worker->flags & WORKER_NOT_RUNNING)       /*非绑定或 CPU 密集型工作者，直接返回*/
        return NULL;

    /*普通工作者*/
    pool = worker->pool;         /*工作者池*/

    /*只能发生在本地 CPU*/
    if (WARN_ON_ONCE(cpu != raw_smp_processor_id() || pool->cpu != cpu))
        return NULL;

    if (atomic_dec_and_test(&pool->nr_running) &&!list_empty(&pool->worklist))
                                        /*pool->nr_running 减 1 后为 0，且 pool->worklist 链表非空*/
        to_wakeup = first_idle_worker(pool);     /*查找第一个空闲工作者*/
    return to_wakeup ? to_wakeup->task : NULL;         /*空闲工作者 task_struct 实例*/
}
```

wq_worker_sleeping()函数判断若当前工作者是非绑定型工作者池中工作者，或者是执行 CPU 密集型工作的工作者，函数直接返回 NULL。因为对这两种情况，该工作者被忽略，不计入工作者池运行工作者数量，其它工作提交时会唤醒空闲工作者，不需要在此处唤醒空闲工作者。

若是普通工作者，对 pool->nr_running 值减 1 后看是否为 0，若为 0 且 pool->worklist 链表非空（还有工作要处理），则查找第一个空闲工作者，返回工作者 task_struct 实例指针。在__schedule()函数中随后将唤醒第一个空闲工作者。

工作者线程在进入睡眠（调度出去）时，pool->nr_running 值将减 1，因此在唤醒工作者线程时，需要将 pool->nr_running 值加 1（见 ttwu_activate()函数），唤醒工作者线程的函数请读者自行阅读源代码。

3 冲刷工作队列

冲刷工作队列是指等待工作队列中提交的工作处理完成，保证在本次操作之前提交的工作都处理完成。

在执行冲刷操作的过程中还会有工作提交，哪么如何来确定哪些工作是在冲刷操作之前提交的，哪些是之后提交的呢？多个冲刷操作也可能同时发生，即一个操作还没完成，另一个冲刷操作已经开始了，内核如何解决这些问题呢？

工作队列机制定义了颜色值，颜色值初始值为 0，范围是[0,0xe]，每发起一次冲刷操作颜色值加 1（达到最大值后返回 0），设为工作队列当前的颜色值 work_color（初始值为 0）。当前颜色值将写入工作实例 data 成员中，pool_workqueue 实例中有一个数组，每个颜色值对应一个数组项，记录尚未处理完的某个颜色值工作的数量。提交工作时，对应数组项值加 1，工作处理完成后，颜色值对应数组项值减 1。

工作队列中还有一个当前冲刷颜色值 flush_color，从 0 开始，表示本次冲刷是等待颜色值为 flush_color 的工作完成。本次冲刷完成后，下一次冲刷 flush_color 值加 1，达到最大值后返回 0。

例如，第一次冲刷时，flush_color 为 0，work_color 值为 1，也就是说发起第一次冲刷操作后，再提交的工作颜色值为 1，第一次冲刷操作就是等待颜色值为 0 的工作处理完成，依此类推。

冲刷操作可以连续发起，即 work_color 值持续增加，但是 flush_color 值必须一次冲刷完成后才能增加。因此发起冲刷操作时，work_color 值会增加，但 flush_color 值不一定增加。内核要记录每次冲刷操作，逐个完成。

一次冲刷操作如何判定已完成呢？最开始冲刷颜色值是 0，工作队列每个 pool_workqueue 实例中数组成员 nr_in_flight[0]数组项，记录了颜色值为 0 且尚未处理完的工作数量，提交工作此数组项值加 1，工作处理完后减 1。发起第一次冲刷操作的进程将在一个完成量上睡眠等待，等到所有 pool_workqueue 实例中 nr_in_flight[0]数组项为 0 时，内核将会唤醒在冲刷操作上等待的进程，表示第一次冲刷操作完成。

若在冲刷操作还没有完成时又发起了冲刷操作，冲刷操作将挂起在工作队列上，等到前一次冲刷操作完成，再执行下一个冲刷操作。其实冲刷操作并没有做什么工作，只是让发起冲刷操作的进程在冲刷操作上睡眠等待，工作完成后将其唤醒而已。

每个冲刷操作由 wq_flusher 结构体表示，定义如下（/kernel/workqueue.c）：

```
struct wq_flusher {
    struct list_head    list;       /*冲刷操作双链表，挂起到工作队列上*/
    int    flush_color;             /*本次操作的冲刷颜色值*/
    struct completion    done;          /*完成量，发起进程在这里等待*/
};
```

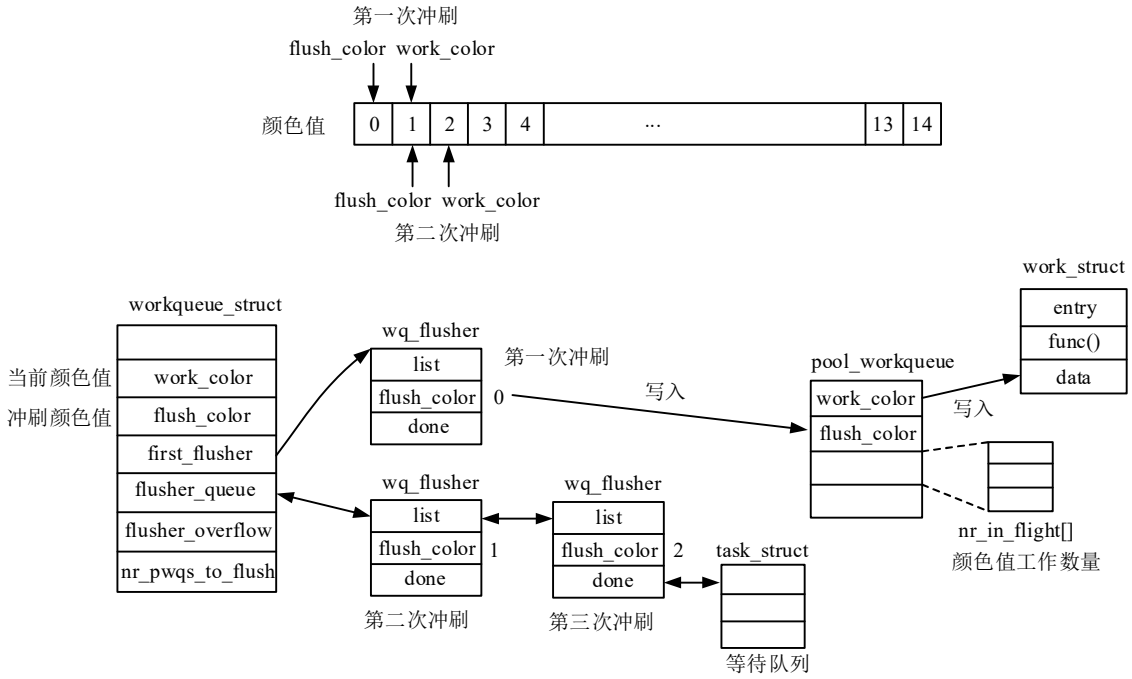
下面先用图示来简要描述一下冲刷操作的原理，然后再看函数代码。如下图所示，最开始当前颜色值 work_color，冲刷颜色值 flush_color 都是 0，提交工作颜色值为 0。

发起第一次冲刷时 work_color=1，flush_color=0，这两个值会更新到 workqueue_struct 实例和其下所有 pool_workqueue 实例，此后提交的工作颜色值为 1。workqueue_struct.first_flusher 指向表示第一次冲刷操作的 wq_flusher 实例，发起冲刷操作的进程在 first_flusher.done 上睡眠等待，所有颜色值为 0 的工作处理完后，将唤醒发起第一次冲刷操作的进程，并且 workqueue_struct.first_flusher 设为 NULL。

如果冲刷操作总是按顺序发起，即一次操作完成后，再发起下一次操作，那么之后的操作和第一次冲刷操作是一样的，只不过 work_color 和 flush_color 值都增加了 1，达到最大值后再返回 0，如此而已。

麻烦之处在于，如果一次冲刷操作还没完成，又发起了下一次冲刷操作，此时要将冲刷操作挂起。如

下图所示，假设第一次冲刷还没完成，又发起了两次，此时 work_color 值会更新为 3，flush_color 值仍为 0。后面发起的操作挂起到工作队列 flusher_queue 双链表，当第一次冲刷完成时，唤醒等待第一次操作完成的进程，将 flush_color 值设为 1，将 workqueue_struct.first_flusher 指向第二次冲刷操作 wq_flusher 实例即可。当颜色值为 1 的工作都处理完成后，将会唤醒等待第二次冲刷完成的进程，并将 flush_color 值设为 2，将 workqueue_struct.first_flusher 指向表示第三次冲刷的 wq_flusher 实例，如此循环。



workqueue_struct 结构体中 nr_pwqs_to_flush 成员表示当前冲刷操作中还有颜色值为 flush_color 的工作没有完成的 pool_workqueue 实例的数量，当此值为 0 时，就可以唤醒等待本次冲刷完成的进程了。

另外还有一种情况需要处理，那就是当一次冲刷都还没有完成时，多次发起的冲刷操作使 work_color 值等于 flush_color 值的情形（回环了，一次冲刷还没完成，work_color 值已经转了一圈了）。在这之后发起的冲刷操作将挂起，且 work_color 值不增加，冲刷操作挂起到工作队列 flusher_overflow 双链表（溢出链表）。当完成一次冲刷操作后，将此链表所有冲刷操作迁移至 flusher_queue 链表末尾并更新 work_color 值（只加 1），即相当于延迟提交冲刷操作。

下面将先介绍更新工作队列 work_color 和 flush_color 值的函数，然后介绍冲刷工作队列的函数，最后介绍唤醒等待冲刷完成进程的函数。

■更新颜色值

flush_workqueue_prep_pwqs()函数用于更新工作队列 work_color 和 flush_color 值，函数定义如下：

```
static bool flush_workqueue_prep_pwqs(struct workqueue_struct *wq, int flush_color, int work_color)
/*flush_color：冲刷颜色值，-1 表示不更新，work_color：当前颜色值，-1 表示不更新*/
{
    bool wait = false;
    struct pool_workqueue *pwq;

    if (flush_color >= 0) {        /*要更新冲刷颜色值，表示开始一次冲刷*/
        WARN_ON_ONCE(atomic_read(&wq->nr_pwqs_to_flush));
        atomic_set(&wq->nr_pwqs_to_flush, 1);       /*nr_pwqs_to_flush 值置 1*/
```

```
        }

        for_each_pwq(pwq, wq) {       /*遍历工作队列 pool_workqueue 实例*/
            struct worker_pool *pool = pwq->pool;
            spin_lock_irq(&pool->lock);

            if (flush_color >= 0) {
                WARN_ON_ONCE(pwq->flush_color != -1);

                if (pwq->nr_in_flight[flush_color]) {        /*有 flush_color 颜色值工作没有完成*/
                    pwq->flush_color = flush_color;       /*设置 pool_workqueue 冲刷颜色值*/
                    atomic_inc(&wq->nr_pwqs_to_flush);       /*加 1*/
                    wait = true;          /*返回值 true，表示本次冲刷需要等待*/
                }
            }

            if (work_color >= 0) {
                WARN_ON_ONCE(work_color != work_next_color(pwq->work_color));
                pwq->work_color = work_color;         /*设置 pool_workqueue 当前颜色值*/
            }
            spin_unlock_irq(&pool->lock);
        }

        if (flush_color >= 0 && atomic_dec_and_test(&wq->nr_pwqs_to_flush))
            complete(&wq->first_flusher->done);         /*若 nr_pwqs_to_flush 为 0，可唤醒等待进程*/
        return wait;       /*返回 true 表示开始了一次冲刷操作，且未完成*/
    }
```

参数 flush_color 若大于或等于 0，表示开始一次冲刷操作，将更新 pool_workqueue.flush_color 冲刷颜色值。参数 work_color 若大于或等于 0，将更新 pool_workqueue.work_color 值，此值在提交工作时将写入工作 data 成员。

flush_workqueue_prep_pwqs()函数返回 true 表示开始了一次冲刷操作，且未完成。返回 false 表示只更新了 work_color 值，或者是开始了一次冲刷操作但是冲刷已完成，不需要等待（可以进行下一次了）。

■冲刷操作

冲刷工作队列的 **flush_workqueue()**函数定义如下（/kernel/workqueue.c）：

```
void flush_workqueue(struct workqueue_struct *wq)
{
    struct wq_flusher this_flusher = {              /*表示冲刷操作*/
        .list = LIST_HEAD_INIT(this_flusher.list),
        .flush_color = -1,
        .done = COMPLETION_INITIALIZER_ONSTACK(this_flusher.done),
```

```
};
int next_color;
lock_map_acquire(&wq->lockdep_map);
lock_map_release(&wq->lockdep_map);
mutex_lock(&wq->mutex);

next_color = work_next_color(wq->work_color);       /*下一颜色值加 1（设为当前颜色值）*/
if (next_color != wq->flush_color) {                /*下一颜色值不等于当前冲刷颜色值，没有溢出*/
    WARN_ON_ONCE(!list_empty(&wq->flusher_overflow));  /*链表需为空*/
    this_flusher.flush_color = wq->work_color;      /*本次冲刷颜色值*/
    wq->work_color = next_color;                    /*设置工作队列当前颜色值*/

    if (!wq->first_flusher) {                       /*当前没有其它冲刷操作*/
        WARN_ON_ONCE(wq->flush_color != this_flusher.flush_color);
        wq->first_flusher = &this_flusher;          /*设为第一个冲刷者*/

        if (!flush_workqueue_prep_pwqs(wq, wq->flush_color,wq->work_color)) {
            /*更新颜色值，若不需要等待，返回*/
            wq->flush_color = next_color;           /*冲刷值设为当前颜色值*/
            wq->first_flusher = NULL;
            goto out_unlock;
        }
    } else {        /*不是第一个冲刷者，已有冲刷还没完成*/
        WARN_ON_ONCE(wq->flush_color == this_flusher.flush_color);
        list_add_tail(&this_flusher.list, &wq->flusher_queue); /*添加到 wq->flusher_queue 链表*/
        flush_workqueue_prep_pwqs(wq, -1, wq->work_color);     /*只更新当前颜色值*/
    }
} else {                /*已溢出，直接添加到 wq->flusher_overflow 双链表*/
    list_add_tail(&this_flusher.list, &wq->flusher_overflow);
}

mutex_unlock(&wq->mutex);
wait_for_completion(&this_flusher.done);      /*当前进程在 this_flusher 上睡眠等待*/

/*进程唤醒后执行以下代码*/
if (wq->first_flusher != &this_flusher)       /*不是第一个冲刷者，直接返回，冲刷完成*/
    return;
/*是第一个冲刷者*/
mutex_lock(&wq->mutex);

if (wq->first_flusher != &this_flusher)
    goto out_unlock;
```

```
wq->first_flusher = NULL;          /*设为 NULL*/
...   /*错误检查*/

/*第一个冲刷者，设置下一次冲刷操作为第一个冲刷者*/
while (true) {       /*无限循环*/
      struct wq_flusher *next, *tmp;
                        /*遍历 wq->flusher_queue 链表中 wq_flusher 实例*/
      list_for_each_entry_safe(next, tmp, &wq->flusher_queue, list) {
            if (next->flush_color != wq->flush_color)        /*冲刷值不等于当前冲刷值，跳出*/
                  break;
            list_del_init(&next->list);
            complete(&next->done);       /*冲刷值等于当前冲刷值，从链表移出，唤醒等待进程*/
      }

      ...   /*错误检查*/

      wq->flush_color = work_next_color(wq->flush_color);       /*下一个冲刷值*/

      if (!list_empty(&wq->flusher_overflow)) {       /*flusher_overflow 链表非空*/
            list_for_each_entry(tmp, &wq->flusher_overflow, list)
                  tmp->flush_color = wq->work_color;   /*冲刷操作冲刷颜色值都设为当前颜色值*/

            wq->work_color = work_next_color(wq->work_color);     /*当前颜色值加 1*/

            list_splice_tail_init(&wq->flusher_overflow,&wq->flusher_queue);
                                    /*flusher_overflow 链表添加到 flusher_queue 链表末尾*/
            flush_workqueue_prep_pwqs(wq, -1, wq->work_color);     /*更新颜色值*/
      }

      if (list_empty(&wq->flusher_queue)) {   /*flusher_queue 链表空，没有下一次操作，跳出*/
            WARN_ON_ONCE(wq->flush_color != wq->work_color);
            break;
      }

      ...   /*错误检查*/

      list_del_init(&next->list);       /*next 指向下一次冲刷操作，从 flusher_queue 链表移出*/
      wq->first_flusher = next;       /*设为第一个冲刷者*/

      if (flush_workqueue_prep_pwqs(wq, wq->flush_color, -1))     /*开始新的冲刷*/
            break;              /*需要等待，跳出循环，已经开始了一个新的冲刷*/
```

```
            wq->first_flusher = NULL;       /*新开启的冲刷不需要等待，遍历下一个 wq_flusher 实例*/
        }   /*while (true)  循环结束，启动了下一次冲刷操作，或没有冲刷操作了*/
    out_unlock:
        mutex_unlock(&wq->mutex);
    }
```

请读者结合前面介绍的原理自行理解 flush_workqueue()函数，呵呵！

内核还定义了 void **drain_workqueue**(struct workqueue_struct *wq)函数，它是 flush_workqueue()函数的包装器，用于等待工作队列为空，是更彻底的冲刷操作，请读者自行阅读源代码。

■唤醒冲刷者

调用者在调用 flush_workqueue()函数后，将在 wq_flusher.done 上睡眠等待，直到所有冲刷颜色值的工作都处理完之后，将被唤醒。

在处理单个工作的 process_one_work(worker, work)函数最后，将调用 **pwq_dec_nr_in_flight()**函数完成完唤醒工作（还有其它工作）。

pwq_dec_nr_in_flight()函数定义如下：

```
static void pwq_dec_nr_in_flight(struct pool_workqueue *pwq, int color)
/*color：本次处理工作的颜色值*/
{
    if (color == WORK_NO_COLOR)
        goto out_put;

    pwq->nr_in_flight[color]--;       /*颜色值对应尚未完成工作数量减 1*/
    pwq->nr_active--;                 /*pool_workqueue 中活跃工作数减 1，提交工作时加 1*/
    if (!list_empty(&pwq->delayed_works)) {          /*处理 pwq->delayed_works 链表挂起工作*/
        if (pwq->nr_active < pwq->max_active)
            pwq_activate_first_delayed(pwq);
                        /*从 pwq->delayed_works 链表迁出一个工作至 pool->worklist 链表*/
    }

    if (likely(pwq->flush_color != color))       /*本次处理工作的颜色值不是本次冲刷的颜 色值，返回*/
        goto out_put;

    /*本次处理工作的颜色值是本次冲刷颜色值，但还有其它同颜色的工作没有处理完，返回*/
    if (pwq->nr_in_flight[color])
        goto out_put;

    /*本次处理工作的颜色值是本次冲刷的颜 色值，且同颜色的工作都处理完了*/
    pwq->flush_color = -1;          /*恢复默认值*/
    /*pwq->wq->nr_pwqs_to_flush 值减 1 后，判断是否为 0*/
```

```
        if (atomic_dec_and_test(&pwq->wq->nr_pwqs_to_flush))      /*所有 pool_workqueue 实例冲刷完*/
            complete(&pwq->wq->first_flusher->done);              /*唤醒等待冲刷完成进程*/
    out_put:
        put_pwq(pwq);
    }
```

pwq_dec_nr_in_flight()函数的主要工作有两项：

（1）从 pwq->delayed_works 链表迁出一个工作至 pool->worklist 链表（如果有挂起的工作），就是说每处理完一个工作，就从 pwq->delayed_works 链表迁出一个工作至工作者池。pwq->delayed_works 链表中延时工作是在提交工作时，pool_workqueue 实例活跃工作数量达到了最大值，工作将添加到此链表。

（2）将 pwq->nr_in_flight[color]中统计的 color 颜色值工作数量减 1（提交时加 1）。若 color 是当前冲刷颜色值，且 nr_in_flight[color]值为 0，则 pwq->flush_color 赋予初始值-1。若所有 pool_workqueue 实例 nr_in_flight[color]值都为 0（wq->nr_pwqs_to_flush=0），则表示当前冲刷颜色值的工作都处理完了，唤醒等待本次冲刷完成的进程。

至此，工作队列机制的全部内容就介绍完了。

6.7 通知链

前面介绍的软中断、工作队列中执行的工作都是由内核决定何时执行的，或指定延时后执行。本节介绍的通知链管理的工作是在某一事件或操作发生时，需要执行的工作。例如，CPU 核热插拔时需要执行的工作，系统进入待机时需执行的工作等。当事件发生时将调用执行通知链中注册的工作。注册到通知链中的工作这里称之为通知。

6.7.1 通知链实现

1 通知链类型

内核支持 4 种类型的通知链，分别是：

（1）原子通知链：在中断或原子上下文中执行通知链中通知，通知执行函数不允许阻塞（睡眠）。

（2）可阻塞通知链：在进程上下文中执行通知链中通知，允许阻塞。

（3）原始通知链：自身不包含保护机制的通知链，锁及保护机制由调用者实现。

（4）SRCU 通知链：可阻塞通知链的变体，使用的保护机制不同。

每种类型通知链在/include/linux/notifier.h 头文件定义了一个对应的链表头结构，通知实例插入到链表中，各类型通知链表头定义如下：

```
struct atomic_notifier_head {              / *原子通知链表头*/
    spinlock_t lock;                       /*保护链表的自旋锁*/
    struct notifier_block __rcu *head;     /*指向链表中第一个通知成员（notifier_block 实例）*/
};

struct blocking_notifier_head {            /*可阻塞通知链表头*/
    struct rw_semaphore rwsem;             /*读写信号量，用于保护通知链*/
    struct notifier_block __rcu *head;
```

```
};

struct raw_notifier_head {                  /*原始通知链表头，由使用者实现保护机投制*/
    struct notifier_block __rcu *head;
};

struct srcu_notifier_head {                 /*SRCU 通知链表头*/
    struct mutex mutex;
    struct srcu_struct srcu;
    struct notifier_block __rcu *head;
};
```

内核在/include/linux/notifier.h 头文件定义了各链表头声明和初始化的宏，例如：

```
#define ATOMIC_NOTIFIER_HEAD(name)          \   /*定义及初始化原子通知链表头*/
    struct atomic_notifier_head name =      \
        ATOMIC_NOTIFIER_INIT(name)

#define BLOCKING_NOTIFIER_HEAD(name)        \   /*定义及初始化可阻塞通知链表头*/
    struct blocking_notifier_head name =    \
        BLOCKING_NOTIFIER_INIT(name)

#define RAW_NOTIFIER_HEAD(name)             \    /*定义及初始化原始通知链表头*/
    struct raw_notifier_head name =         \
        RAW_NOTIFIER_INIT(name)
```

由通知链表头结构定义可知，各种类型通知链实际上管理的是 **notifier_block** 结构体实例，该结构体表示通知，即需要执行的工作。

notifier_block 结构体定义在/include/linux/notifier.h 头文件：

```
struct notifier_block {
    notifier_fn_t    notifier_call;         /*执行函数*/
    struct notifier_block __rcu *next;      /*指向通知链表中下一成员*/
    int priority;                           /*优先级*/
};
```

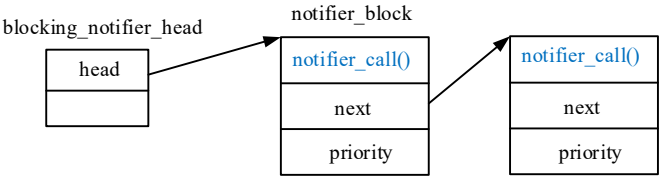
通知执行函数原型定义如下（函数参数来自于执行通知链的调用者）：

```
typedef    int (*notifier_fn_t)(struct notifier_block *nb,unsigned long action, void *data);
```

action 参数表示动作，data 参数表示数据，这都是由使用通知链的代码定义。函数返回值如下：

```
#define    NOTIFY_DONE    0x0000        /* Don't care */
#define    NOTIFY_OK      0x0001        /*正确地执行了*/
#define    NOTIFY_STOP_MASK      0x8000        /*不要执行后面的通知了*/
#define    NOTIFY_BAD          (NOTIFY_STOP_MASK|0x0002)
#define    NOTIFY_STOP         (NOTIFY_OK|NOTIFY_STOP_MASK)
```

下图示意了可阻塞通知链的结构，其它类型通知链与此类似：



2 注册通知

内核静态定义了多个通知链实例，如：CPU 通知链，重启通知链等（/include/linux/notifier.h）。在某一事件发生时需要执行的通知 notifier_block 实例需注册到相应的通知链表中。

注册通知实例的函数声明如下（实现函数在/kernel/notifier.c）：

int atomic_notifier_chain_register(struct atomic_notifier_head *nh,struct notifier_block *nb);

int blocking_notifier_chain_register(struct blocking_notifier_head *nh,struct notifier_block *nb);

int raw_notifier_chain_register(struct raw_notifier_head *nh,struct notifier_block *nb);

int srcu_notifier_chain_register(struct srcu_notifier_head *nh,struct notifier_block *nb);

int blocking_notifier_chain_cond_register(struct blocking_notifier_head *nh,struct notifier_block *nb);

各注册通知函数内部都是调用 notifier_chain_register()函数按优先级数值从大到小排序，将通知实例添加到通知链表中。

注销通知的函数声明如下：

int atomic_notifier_chain_unregister(struct atomic_notifier_head *nh,struct notifier_block *nb);

int blocking_notifier_chain_unregister(struct blocking_notifier_head *nh,struct notifier_block *nb);

int raw_notifier_chain_unregister(struct raw_notifier_head *nh,struct notifier_block *nb);

int srcu_notifier_chain_unregister(struct srcu_notifier_head *nh,struct notifier_block *nb);

注销函数很简单，只是将通知 notifier_block 实例从链表中移出（没有销毁）。

3 执行通知链

当某一事件发生时，内核需要执行通知链中注册的通知，处理函数声明如下：

int atomic_notifier_call_chain(struct atomic_notifier_head *nh,unsigned long val, void *v);

int blocking_notifier_call_chain(struct blocking_notifier_head *nh,unsigned long val, void *v);

int raw_notifier_call_chain(struct raw_notifier_head *nh,unsigned long val, void *v);

int srcu_notifier_call_chain(struct srcu_notifier_head *nh,unsigned long val, void *v);

执行通知链函数第一个参数为需要处理的通知链表头指针，val 和 v 参数是需要传递给每一个通知执行函数 notifier_call()的参数，val 通常表示动作或事件，v 表示数据。

执行通知链函数内都是调用 notifier_call_chain()函数执行通知链中通知，在调用这个函数前会获取通知链保护锁，以保证操作不被影响。

notifier_call_chain()函数定义如下（/kernel/notifier.c）：

static int notifier_call_chain(struct notifier_block **nl,unsigned long val, void *v, \

```
                                                    int nr_to_call, int *nr_calls)
/*nr_to_call：执行通知数量，通常为-1，表示执行所有通知，nr_calls：通常为 NULL*/
{
    int ret = NOTIFY_DONE;
    struct notifier_block *nb, *next_nb;

    nb = rcu_dereference_raw(*nl);        /*第一个通知*/

    while (nb && nr_to_call) {        /*遍历通知链表中实例*/
        next_nb = rcu_dereference_raw(nb->next);

    #ifdef CONFIG_DEBUG_NOTIFIERS
        ...
    #endif
        ret = nb->notifier_call(nb, val, v);        /*调用通知执行函数*/

        if (nr_calls)
            (*nr_calls)++;        /*保存真正执行通知的数量*/

        if ((ret & NOTIFY_STOP_MASK) == NOTIFY_STOP_MASK)
            break;
        nb = next_nb;
        nr_to_call--;       /*数量减 1*/
    }
    return ret;
}
```

notifier_call_chain()函数遍历通知链表中通知，依次调用实例中 notifier_call()函数。若某一实例执行函数返回值为 NOTIFY_STOP_MASK 时，将停止执行，不再执行后面的通知了。

6.7.2 通知链示例

本小节简要介绍两个内核中常用的通知链实例。

1 CPU 通知链

CPU 通知链主要用于 CPU 核热插拔。如果内核配置选择了 SMP 选项，内核在/kernel/cpu.c 文件内将定义原始通知链：

```
static RAW_NOTIFIER_HEAD(cpu_chain);       /*CPU 通知链（原始通知链）*/
```

内核在/include/linux/cpu.h 头文件内定义了向 CPU 通知链注册通知的宏：

```
#define cpu_notifier(fn, pri) {                    \    /*需选择 SMP 和 HOTPLUG_CPU 选项*/
    static struct notifier_block fn##_nb =         \    /*定义通知实例*/
        { .notifier_call = fn, .priority = pri };    \
```

```
    register_cpu_notifier(&fn##_nb);            \    /*向 cpu_chain 通知链注册通知*/
}
#define hotcpu_notifier(fn, pri)    cpu_notifier(fn, pri)
```

cpu_notifier(fn,pri)宏内创建通知实例 fn_nb，通知执行函数为 fn()，并向向 cpu_chain 通知链注册。

void __ref unregister_cpu_notifier(struct notifier_block *nb)函数用于注销 cpu_chain 通知链中通知。

cpu_notify(unsigned long val, void *v)函数用于执行 cpu_chain 通知链注册的通知，val 表示动作，*v 保存 CPU 核编号。CPU 通知优先级以及动作（val）定义在/include/linux/cpu.h 头文件。

在 CPU 热插拔时将调用此函数，如：_cpu_up()函数、take_cpu_down()函数等。

如果没有选择 SMP 或 HOTPLUG_CPU 选项，cpu_notifier(fn, pri)将转换成定义函数指针 fn()。

2 PM 通知链

PM 通知链是电源管理通知链，通知链定义在/kernel/power/main.c 文件内（需选择 PM_SLEEP 配置项）：

```
static BLOCKING_NOTIFIER_HEAD(pm_chain_head);    /*可阻塞通知链*/
```

内核在/include/linux/suspend.h 头文件内定义了注册 PM 通知链通知的宏：

```
#define pm_notifier(fn, pri) {                  \
    static struct notifier_block fn##_nb =      \    /*定义通知实例，执行函数 fn()*/
        { .notifier_call = fn, .priority = pri };  \
    register_pm_notifier(&fn##_nb);             \    /*注册通知*/
}
```

int unregister_pm_notifier(struct notifier_block *nb)函数用于注销 PM 通知链通知。

pm_notifier_call_chain(unsigned long val)函数用于执行 PM 通知链通知，在 CPU 电源（功耗）管理中调用。val 表示动作，取值定义在/include/linux/suspend.h 头文件。

6.8 时间管理

内核中的时间值并不是以我们常用的年月日、时分秒的时间值来表示。内核时间用基于某个时间起点到现在所经历的时间长度来表示（时间一直累加），单位为秒数、微秒数或纳秒数等。

年月日、时分秒表示的时间称为真实时间（挂钟时间）。真实时间在内核中是以格林尼治时间（GMT）1970 年 1 月 1 日 0 点（UNIX/Linux 纪元的起点）为起点到现在所经历的时长（秒数）来表示。内核中真实时间的表示格式与年月日、时分秒格式之间的转换由库函数完成。

内核中主要记录了两个时间，一个是单调递增时间（CLOCK_MONOTONIC），它从系统启动时开始计时，与真实时间同步。系统启动时的真实时间加上当前单调递增时间值就是当前真实时间值。

另一个是原始单调递增时间（CLOCK_MONOTONIC_RAW），它以时间管理子系统初始化时为起点开始计时。

内核中通过时钟硬件的周期计数值来记录经历的时长。

内核中除了需要时间值外，还需要按指定时长（周期）产生的时钟事件（硬件中断），在时钟事件中完成内核的周期性、例行性工作，如进程时间统计、进程调度、处理定时器等。时钟事件也由时钟硬件产生，在硬件中断处理函数中调用时钟事件处理函数。

本节主要介绍内核时间管理通用框架、时间值的产生和记录、获取和设置时间值的接口函数、时钟事件的触发和处理、定时器的设置和处理、以及进程时间统计和定时等内容，相关源代码位于/kernel/time/目录下。

6.8.1 概述

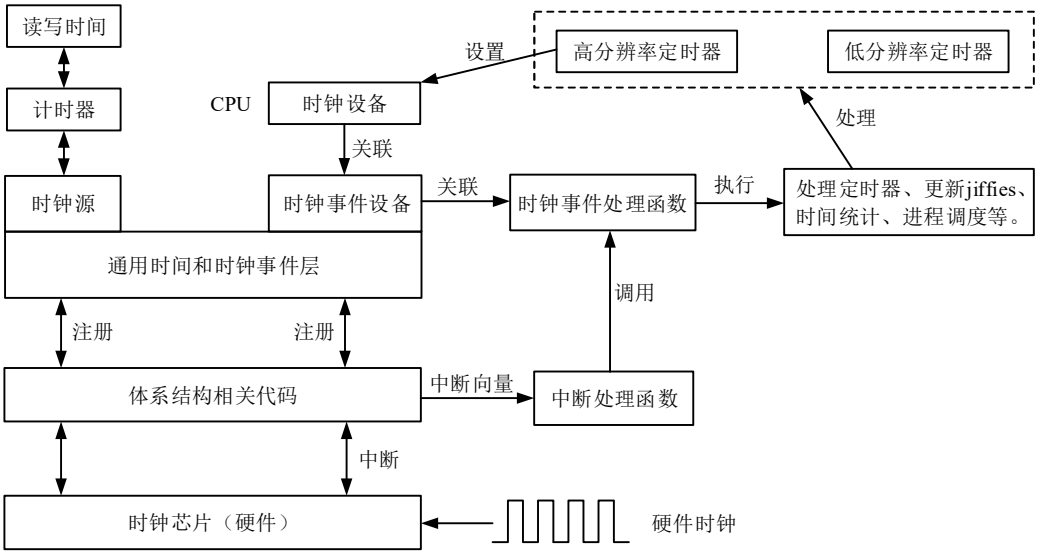
1 通用时间框架

内核定义了用于时间管理的通用时间框架（需选择 GENERIC_CLOCKEVENTS，MIPS 默认选择），通用时间框架主要提供两个功能：

（1）记录时间流，为内核和用户提供时间值。

（2）提供了时钟事件设备，各 CPU 核可通过此设备产生时钟事件，简单地说就是按照给定时间产生中断。在时钟事件处理函数（由中断处理函数调用）中，完成内核周期性、例行性的工作。例如：进程时间统计、进程调度、处理到期定时器等。

内核通用时间框架如下图所示：



通用时间框架需要基于一个（或多个）时钟硬件，时钟硬件主要提供两个功能：一是利用硬件时钟周期的计数值来记录时间值（流逝的时间长度），二是可以通过对其编程使其在指定的时间间隔后产生硬件中断，并指定中断处理函数。前一个功能用于实现通用时间框架的计时功能，后者用于实现时钟事件设备。一个时钟硬件可以用于这两个功能，也可以只用于一个，或者不使用。

时钟源是对时钟硬件周期时钟计数功能的抽象。时钟源中包含读取时钟硬件计数值的函数，以及将周期计数值转为时间值（纳秒数）所需的信息等。时钟硬件若需要向内核提供时钟源的功能，需要向内核注册时钟源实例，内核负责管理时钟源。

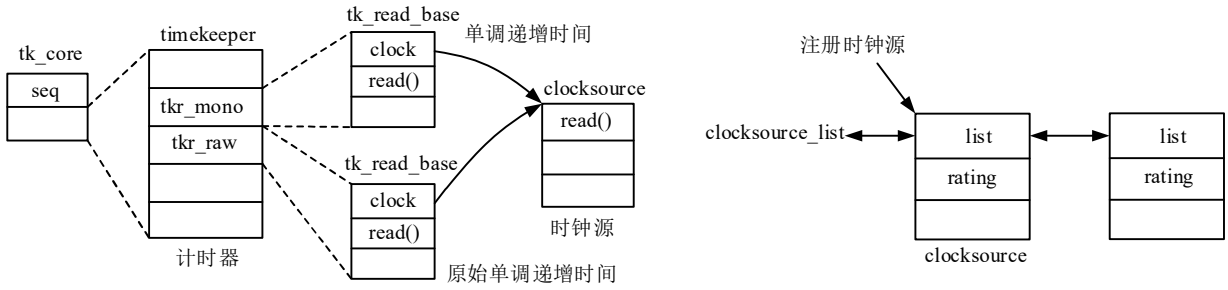
计时器相当于内核的时钟，它通过时钟源获取时间值。内核提供的获取当前时间值的接口函数从计时器中读取时间值，计时器又是从时钟源中获取时间值。内核中可以有多个时钟源，但是计时器是由内核静态定义的，并且一个计时器只能关联一个时钟源，在注册时钟源时内核会自动为计时器选择最优的时钟源。

时钟事件设备是对时钟硬件触发时钟事件（中断）的抽象，即可对其进行编程，以便在给定的时间间隔后产生中断，中断处理函数中调用相应的时钟事件处理函数。在时钟事件处理函数执行相应的内核工作，如执行进程调度，处理到期定时器等。时钟硬件相关代码可以向时间管理通用层注册时钟事件设备，每个 CPU 核通过时钟设备关联一个最优的时钟事件设备。

时钟硬件可以同时注册时钟源和时钟事件设备，内核中可以有多个注册的时钟源和时钟事件设备。

2 计时器

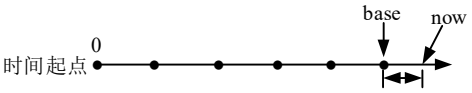
通用时间框架最基本的功能就是记录时间，为内核和用户提供时间值。通用时间框架中用于记录时间的部分如下图所示：



时钟源由 clocksource 结构体表示，由体系结构相关代码或时钟硬件驱动程序注册。CPU 核中通常有一个计数器寄存器，它在硬件时钟的驱动下计数器，达到最大值后返回 0。体系结构相关代码通常会将此计数器注册为时钟源。时钟源的主要作用是将计数器的计数周期数转化为时间间隔值（纳秒数）。系统内可以注册多个时钟源，由全局双链表管理，时钟源在双链表中按精度排序。

计时器是真正的内核时钟，记录从某个时间点开始到现在经历的时间值。计时器由 timekeeper 结构体表示。内核静态定义了计时器，它是全局的，供所有 CPU 核获取时间值。计时器关联到精度最高的时钟源，在注册时钟源时，若新注册的时钟源精度更高，则会替换计时器原先关联的时钟源。

计时器计时原理如下图所示：



内核会周期性地更新计时器时间值（上图中黑点），即读取时钟源的计数值，并与前一次读取的计数值计算差值，由计数差值和计数周期算出时间间隔以累加出当前时间值。当前时间值是一个基准时间值，就是最近一次更新的时间值，下次更新时在此基础上加上时间间隔。更新时间时还会记录本次更新读取的计数器值。

获取时间值时，从时钟源读取当前计数值，计算其与最近一次更新时间时的计数差值，转为时间间隔，时间间隔加上基准时间就是当前时间值，简单地说就是分段计算时间间隔，然后累加。

内核中用 **ktime/ktime_t** 结构体来表示以纳秒数计时的时间值，结构体定义在/include/linux/ktime.h 头文件：

```
union ktime {
        s64    tv64;          /*以纳秒为单位的时间值，64 位有符号整型数*/
};
typedef union ktime ktime_t;
```

内核在/include/linux/ktime.h 头文件定义了 ktime_t 变量的操作接口函数，例如：

●ktime_to_ns(kt)：返回 ktime_t 实例 tv64 成员表示的纳秒数。

●ktime_set(const s64 secs, const unsigned long nsecs)：用秒、纳秒表示的时间值设置 ktime_t 实例。

●**ktime_sub**(lhs, rhs)：返回(lhs).tv64-(rhs).tv64 值，用于设置 ktime_t 实例，下同。

●**ktime_add**(lhs, rhs)：返回(lhs).tv64 +(rhs).tv64 值。
●**ktime_add_ns**(kt, nsval)：返回(kt).tv64+(nsval)值。
●**ktime_sub_ns**(kt, nsval)：返回(kt).tv64 -(nsval)值。

timespec64 结构体用于由秒和纳秒表示的时间值，定义如下（/include/linux/time64.h）：

```
typedef __s64    time64_t;              /*64 位有符号数*/
#if __BITS_PER_LONG == 64
    # define timespec64    timespec     /*64 位系统 timespec64 与 timespec 相同，见下文*/
#else
    struct timespec64 {                 /*32 位系统*/
        time64_t    tv_sec;             /*秒，64 位*/
        long        tv_nsec;            /*纳秒*/
    };
#endif
```

内核在/include/linux/ktime.h 头文件定义了 timespec64 与 ktime_t 相互转换的接口函数，例如：
●ktime_t timespec64_to_ktime(struct timespec64 ts)：timespec64 转 ktime_t。
●ktime_to_timespec64(kt)：ktime_t 转 timespec64。

用户空间表示时间值的数据结构定义如下（/include/uapi/linux/time.h）：
timespec：由秒、纳秒表示的时间值（时间间隔）。

```
struct timespec {
    __kernel_time_t     tv_sec;         /*秒*/
    long        tv_nsec;                /*纳秒*/
};
```

timeval：由秒、微秒表示的时间值（时间间隔）。

```
struct timeval {
    __kernel_time_t          tv_sec;        /*秒*/
    __kernel_suseconds_t     tv_usec;       /*微秒*/
};
```

内核在/include/linux/ktime.h 头文件定义了 timespec、timeval 与 ktime_t 之间的转换函数，例如：
●ktime_t timespec_to_ktime(struct timespec ts)：timespec 转为 ktime_t 的时间表示。
●ktime_to_timespec(kt)：ktime_t 转为 timespec 的时间表示。
●ktime_t timeval_to_ktime(struct timeval tv)：timeval 转为 ktime_t 的时间表示。
●ktime_to_timeval(kt)：ktime_t 转为 timeval 的时间表示。

内核中还有一个比较常用的记录时间的变量，那就是 jiffies（jiffies64）。jiffies 表示的是什么意思呢？CPU 核的时钟设备通常会以一个固定的时间间隔，周期性地产生时钟事件（中断），以便 CPU 核完成周期性的工作，例如进程调度等。每个事件（中断）称为一个节拍，jiffies 记录的就是节拍的数量，系统启动时从 0 开始计数。若要获取时间值，将 jiffies 值乘以周期就可以了。内核配置常数 HZ 表示的是一秒内

产生的节拍数量，周期就是 1/HZ 秒。

jiffies（jiffies64）是一个全局变量，而每个 CPU 核都有自己的时钟设备，因此只需要一个 CPU 核的时钟设备去更新 jiffies（jiffies64）值即可，这称为全局时钟设备。

jiffies 是一个无符号整型数，（声明在/include/linux/jiffies.h），在 32 位系统中为 32 位，在 64 位系统中为 64 位。

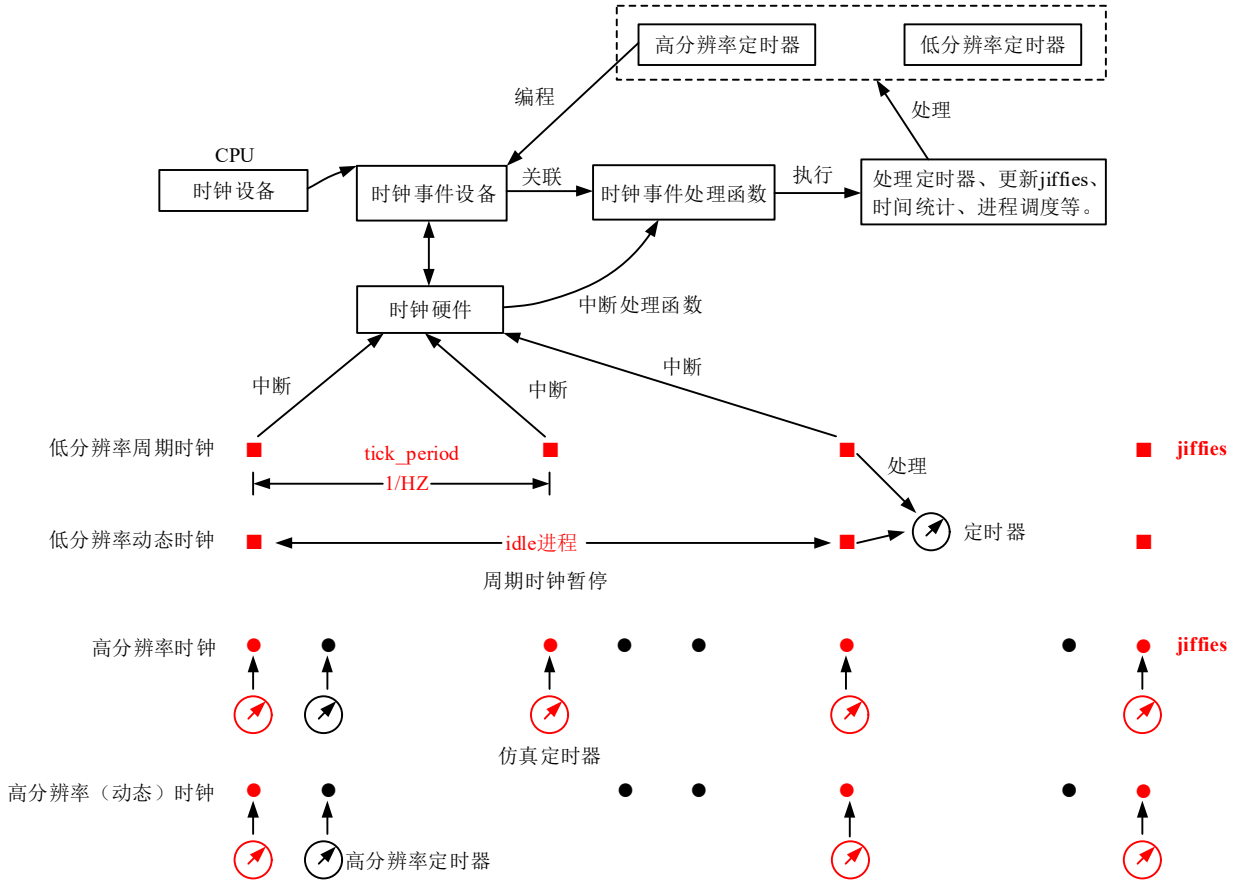
jiffies_64 是一个 64 位无符号整型数，不管在什么系统中（声明在/include/linux/jiffies.h）。在 64 位系统中 jiffies_64 与 jiffies 相同，是同一个变量的两个名称，在 32 位系统中 jiffies 是 jiffies_64 的低 32 位。这两个变量定义在链接文件/arch/mips/kernel/vmlinux.lds.S 中，指向同一个地址。

jiffies/jiffies_64 值的更新在全局时钟设备周期事件中进行，读取 jiffies 值可直接进行，与读整型数的操作相同，读取 jiffies_64 值需要调用接口函数 get_jiffies_64()，因为在 32 位系统中读取 jiffies_64 值不是原子操作。jiffies/jiffies_64 值的操作接口函数都定义在/include/linux/jiffies.h 头文件，请读者自行查阅。

3 时钟设备

通用时间框架另一个重要的功能是向 CPU 核提供时钟设备。时钟设备可视为一个可编程的设备，可设置其在指定时间间隔后产生事件（中断，可以周期性或单次触发），并可设置事件处理函数。事件处理函数可用来更新全局 jiffies/jiffies_64 值、执行进程调度、处理定时器等。

时钟设备框架如下图所示：



CPU 核的时钟设备关联到时钟事件设备。时钟事件设备是对时钟硬件在指定时间产生硬件中断功能的抽象。内核可对时钟事件设备进行编程，使其在指定的时间点产生硬件中断，中断处理函数中调用时钟事件设备注册的时钟事件处理函数。

每个 CPU 核具有自己的时钟设备，关联的时钟事件设备需要绑定至 CPU 核，因为时钟事件是由中断

触发的，此中断需由绑定的 CPU 核处理。与时钟源一样，内核也通过双链表来管理时钟事件设备，注册时钟事件设备时，需要判断是否要取代 CPU 核时钟设备当前关联的时钟事件设备。时钟事件设备具有 CPU 亲和属性，可以在 CPU 核之间迁移。

时钟设备产生的时钟事件这里也称它为时钟，最初的时钟事件是周期产生的，类似于硬件时钟的计数器，故而称它为时钟。

依据时钟设备产生时钟事件（中断）的精度，即时间间隔的精度，时钟设备分为低分辨率和高分辨率时钟设备。依据时钟事件是否是周期产生的，分为周期时钟和动态时钟，因此时钟设备一共有 4 钟工作模式，如下所示。

（1）低分辨率周期时钟：这是最初传统的工作模式，时钟事件以 1/HZ 为周期产生，时钟设备的定时精度较低，也就是周期间隔较大。低分辨率定时器以 jiffies/jiffies_64 值定时，在时钟事件处理函数中将检测定时器到期时间是否小于等于当前 jiffies/jiffies_64 值，若是（超时）则调用定时器处理函数。

（2）低分辨率动态时钟：与低分辨率周期时钟的区别是当 CPU 在运行 idle 进程时将暂停周期时钟，以节省电能，退出 idle 进程时恢复周期时钟。

（3）高分辨率周期时钟：时钟事件设备具有比较高的定时精度。高分辨率时钟事件总是由高分辨率定时器到期触发的，而不像低分辨率时钟一样是以固定周期触发的。内核会用最近到期的高分辨率定时器的到期时间对时钟事件设备编程。内核定义了一个高分辨率定时器，以 1/HZ 为周期到期，模拟周期时钟，在其到期处理函数中执行内核例行性工作。此模式下，时钟事件处理函数只需要调用到期（过期）高分辨率定时器的处理函数即可。

（4）高分辨率动态时钟：与高分辨率周期时钟的区别是模拟周期时钟的定时器在 CPU 运行 idle 进程时不到期（到期时间无限长），即暂停周期时钟（其它定时器不受影响），退出 idle 进程时重启定时器。

高分辨率周期时钟和高分辨率动态时钟内核并没有区分而是使用相同的处理函数，因为它们的区别只是模拟定时器设置的到期时间会有变化，其它操作都是一样的。

另外，内核时间管理子系统还将统计进程时间，为进程提供定时和休眠等功能，还支持 POSIX 标准时钟和定时器。

6.8.2 计时器与时钟源

通用时间框架中的计时器用来记录从某个时间点开始到现在经历的时间值（长度），精度为纳秒。计时器是内核中的时钟，内核可从中获取当前时间值。

内核中主要有两个计时器，一个是记录单调递增时间，起点是系统启动时刻，它与真实时间同步，加上系统启动时的真实时间就是当前的真实时间，内核利用单调递增时间来记录真实时间；另一个是原始单调递增时间，它从时间管理子系统初始化时开始计时。

计时器关联到时钟源，利用时钟源来计算经历时间的长度，以累加出时间值。内核在周期时钟事件中会更新计时器时间值，以免时钟源计数器溢出（回环）导致计时失真。

计时器是全局的，初始化时关联到初始的 clocksource_jiffies 时钟源。体系结构或平台相关代码可以注册时钟源，注册时内核会自动选择最优的时钟源关联到计时器，为其提供计时功能。

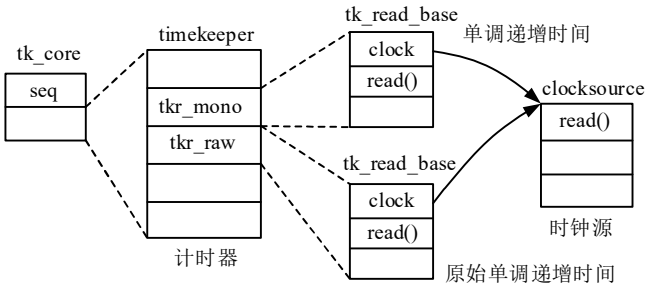
用户进程可读取/设置当前真实时间值。内核代码中可获取当前单调递增时间值和原始单调递增时间值。

计时器相关代码位于/kernel/time/timekeeping.c 文件内，时钟源相关代码位于/kernel/time/clocksource.c 文件内。

1 数据结构

计时器由 timekeeper 结构体表示，内核静态定义了此结构体实例。时钟源由 clocksource 结构体表示，内核中可以有多个时钟源（可动态变化），但是计时器是固定的。在向内核注册时钟源时，会自动选择最优的时钟源关联到计时器。

计时器相关数据结构组织关系如下图所示：



■计时器

tk_core 结构体实例封装了内核常用的两个计时器，如下所示（/kernel/time/timekeeping.c）：

```
static struct {
    seqcount_t          seq;      /*保护 timekeeper 实例的顺序锁*/
    struct timekeeper  timekeeper;          /*计时器*/
} tk_core    ____cacheline_aligned;

static struct timekeeper shadow_timekeeper;     /*tk_core 的影子，在修改 tk_core 时使用*/
```

timekeeper 结构体定义如下（/include/linux/timekeeper_internal.h）：

```
struct timekeeper {
    struct tk_read_base     tkr_mono;       /*单调递增时间*/
    struct tk_read_base     tkr_raw;        /*原始单调递增时间*/
    u64                 xtime_sec;      /*当前真实时间值，秒数，剩余纳秒数保存在 tkr_mono.xtime_nsec*/
    unsigned long           ktime_sec;
                            /*当前单调递增时间值，秒数，剩余纳秒数保存在 tkr_mono.xtime_nsec*/
    struct timespec64   wall_to_monotonic;    /*真实时间至单调递增时间的偏移量，负数*/
    ktime_t             offs_real;    /*单调递增时间至真实时间的偏移量，正数，纳秒数*/
    ktime_t             offs_boot;    /*单调递增时间至启动时间的偏移量，纳秒*/
    ktime_t             offs_tai;     /*单调递增时间至国际原子时间的偏移量，纳秒*/
    s32                 tai_offset;   /*UTC 至 TAI 时间的偏移量，秒*/
    unsigned int            clock_was_set_seq;    /**/
    ktime_t                 next_leap_ktime;
    struct timespec64   raw_time;     /*原始单调递增时间，用于设置 tkr_raw.base 值*/

    /*timekeeper 内部使用成员*/
    cycle_t             cycle_interval;     /*一个节拍内时钟源计数值，即一个节拍内有几个计数周期*/
    u64                 xtime_interval;
```

```
    s64             xtime_remainder;
    u32             raw_interval;    /*一个节拍中所含的纳秒数*/
    u64             ntp_tick;
    s64             ntp_error;
    u32             ntp_error_shift;
    u32             ntp_err_mult;
#ifdef CONFIG_DEBUG_TIMEKEEPING
    ...
#endif
};
```

timekeeper 结构体主要成员简介如下：

●**tkr_mono**：tk_read_base 结构体成员，记录单调递增时间，定义见下文。

●**tkr_raw**：tk_read_base 结构体成员，所用时钟源与 tkr_mono 相同，记录原始单调递增时间。

●**xtime_sec**：最近一次更新真实时间值，单位秒，剩余纳秒数保存在 **tkr_mono.xtime_nsec**。

●**ktime_sec**：最近一次更新单调递增时间值，单位秒，剩余纳秒数保存在 **tkr_mono.xtime_nsec**。

●**wall_to_monotonic**：真实时间至单调递增时间的偏移量，负数，timespec64 结构体表示，真实时间加 wall_to_monotonic 得到单调递增时间，由于此值为负值，相当于减操作。

●**offs_real**：单调递增时间至真实时间的偏移量，正数，纳秒数，单调递增时间加 offs_real 得到真实时间。

●**raw_time**：最近一次更新原始单调递增时间值，由 timespec64 结构体表示，用于设置 tkr_raw.base 基准值。

tk_read_base 结构体定义在/include/linux/timekeeper_internal.h 头文件：

```
struct tk_read_base {
    struct clocksource  *clock;        /*指向当前使用的时钟源*/
    cycle_t         (*read)(struct clocksource *cs);    /*读取时钟源计数器值函数*/
    cycle_t         mask;              /*计数器掩码，同关联时钟源中的值*/
    cycle_t         cycle_last;        /*最近一次更新时间值时读取时钟源计数器中的值*/
    u32             mult;          /*计数值转时间值的乘数，同关联时钟源中的值*/
    u32             shift;         /*计数值转时间值的位移数，同关联时钟源中的值*/
    u64             xtime_nsec;    /*时间值中不足 1 秒剩余的纳秒数乘以 2^shift（mult）*/
    ktime_t         base;              /*最近一次更新时间值，单位纳秒*/
};
```

tk_read_base 结构体主要成员简单如下：

●**clock**：关联的时钟源。

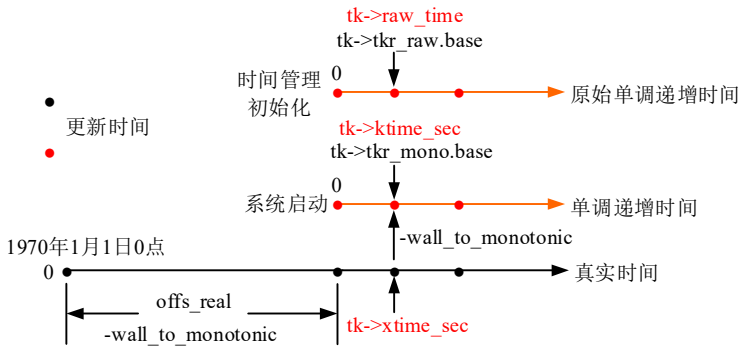
●**read()**：读取时钟源计数器值函数，同关联 clocksource 实例中的 read()函数。

●**cycle_last**：最近一次更新时间时读取时钟源计数器值。

●**xtime_nsec**：时间值中不足 1 秒剩余的纳秒数乘以 2^{shift}。

●**base**：最近一次更新的时间值，单位纳秒。

先看下图，说明一下以上数据结构中各成员的语义：



原始单调递增时间从时间管理子系统初始化时开始计时，从 0 开始，这个时间是严格单调递增的，用户不可修改。

单调递增时间从系统启动时开始计时，从 0 开始，它与真实时间是同步的，当前真实时间减去系统启动时的真实时间就是单调递增时间。时间管理初始化时，会从外部获取当前真实时间，这是一个基准，随后内核通过在此基础上累加单调递增时间用于记录真实时间。用户修改真实时间时，也会修改与单调递增时间的偏移量，以保持与单调递增时间同步。

tk_core 实例中 timekeeper 成员这里用 tk 来简写。tk.xtime_sec 成员记录的是真实时间的秒数，剩余纳秒数保存在 tk.tkr_mono.xtime_nsec 成员中。

tk.ktime_sec 成员记录单调递增时间的秒数，剩余纳秒数保存在 tk.tkr_mono.xtime_nsec 成员中。

tk.raw_time 成员记录单调递增时间值。

tk.wall_to_monotonic 是真实时间至单调递增时间的偏移量，负数，即真实时间加此值得单调递增时间。

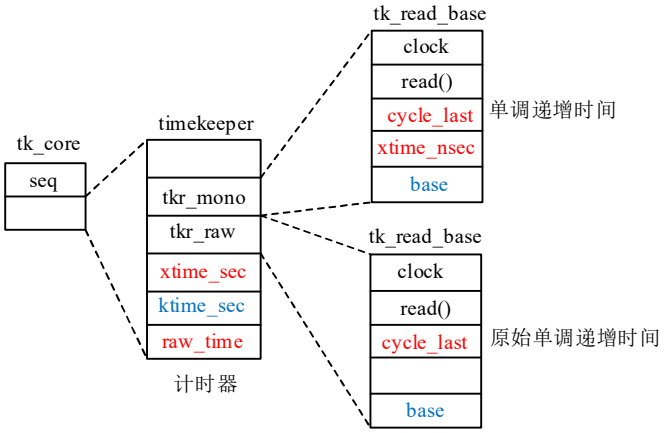
tk.offs_real 单调递增时间至真实时间的偏移量，正数，单调递增时间加上此值得真实时间。

内核会在周期时钟事件等时机更新各时间值（上图中圆点），完成以下工作：

（1）由关联时钟源读取当前计数值，由上次更新时的计数值和本次读取的计数值，计算出时间间隔，累加到真实时间 tk.xtime_sec 和原始单调递增时间 tk.raw_time，并更新计数值 tk->tkr_mono.cycle_last 和 tk->tkr_raw.cycle_last。

（2）将真实时间 tk.xtime_sec 值加 wall_to_monotonic 后赋予 **tk->tkr_mono.base** 成员，作为单调递增时间基准，将 tk->raw_time 值赋予 **tk->tkr_raw.base** 成员作为原始单调递增时间基准，即最近一次更新的单调递增时间和原始单调递增时间。更新单调递增时间 tk.ktime_sec 值。

如下图所示，红色值是第（1）步更新的值，蓝色值是第（2）步更新的值。



tk_read_base.base 保存的是最近一次更新时间值，tk_read_base.cycle_last 保存的是最近一次更新时间值时读取的计数器值。获取当前时间时，读取当前时钟源计数器值，计算其与 cycle_last 值的差值，从而计算出时间间隔，累加到 base 时间基准上即得到当前时间。注意，在读取时间时并不会更新以上数据结

构中的任何成员值，以上成员值只会在内核更新时间值时修改。

内核在/kernel/time/timekeeping.c 内还定义了 tk_fast 结构体及实例（与 tk_core 类似），可用于在不可屏蔽中断（NMI）中获取时间值，并提供了相应的接口函数：

```
struct tk_fast {
    seqcount_t          seq;
    struct tk_read_base     base[2];
};
```

tk_fast 实例定义如下：

```
static struct tk_fast   tk_fast_mono ____cacheline_aligned;      /*单调递增时间*/
static struct tk_fast   tk_fast_raw    ____cacheline_aligned;    /*原始单调递增时间*/
```

tk_fast_mono 和 tk_fast_raw 与 tk_core 实例使用相同的时钟源。

■时钟源

时钟源是对时钟硬件周期时钟计数功能的抽象，用于获取计数器的计数值，并转为时间值（时间间隔）。时钟硬件计数器在外部硬件时钟的驱动下计数，每个时钟计数值加 1，计数值达到最大值时将循环从 0 开始计数。通过计数值和硬件时钟周期即可计算出所经历的时间间隔。

时钟源在内核中由 clocksource 结构体表示，结构体定义在/include/linux/clocksource.h 头文件：

```
struct clocksource {
    cycle_t    (*read)(struct clocksource *cs);      /*读取当前计数器值的函数指针*/
    cycle_t    mask;        /*如果 read()返回值不是 64 位，表示对结果的掩码，屏蔽不使用的位*/
    u32 mult;               /*计数值转为纳秒值时使用的乘数*/
    u32 shift;              /*计数值转为纳秒值时，乘以 mult 后右移 shift 位*/
    u64 max_idle_ns;     /*时钟源允许最大空闲时间，纳秒值*/
    u32 maxadj;             /*最大调整值*/
#ifdef CONFIG_ARCH_CLOCKSOURCE_DATA
    struct arch_clocksource_data    archdata;
#endif
    u64 max_cycles;         /*最大允许计数器值*/
    const char *name;       /*时钟源名称*/
    struct list_head list;    /*双链表成员，将 clocksource 实例添加到全局双链表*/
    int rating;             /*表示时钟源精度，值越大表示时钟源质量越好*/
    int    (*enable)(struct clocksource *cs);       /*使能时钟源*/
    void    (*disable)(struct clocksource *cs);     /*关闭时钟源*/
    unsigned long    flags;                         /*标记*/
    void    (*suspend)(struct clocksource *cs);     /*暂停时钟源*/
    void    (*resume)(struct clocksource *cs);      /*重启时钟源*/

  #ifdef CONFIG_CLOCKSOURCE_WATCHDOG
    struct list_head wd_list;
```

```
    cycle_t cs_last;
    cycle_t wd_last;
  #endif
    struct module *owner;      /*模块指针，可以在模块中注册时钟源*/
} ____cacheline_aligned;
```

clocksource 结构体主要成员简介如下：

●**read()**：读取时钟硬件当前计数器值的函数指针，返回值为 cycle_t 类型，定义在/include/linux/types.h 头文件内，为 64 位无符号整数。

●**mask**：如果时钟硬件计数器不是 64 位的，mask 标记实际使用的位。CLOCKSOURCE_MASK(bits)宏用于获取 mask 值，bits 表示计数器实际使用的二进制位数。

●**list**：双链表成员，内核定义了全局双链表 clocksource_list 用于管理所有的时钟源实例。

●**rating**：时钟源等级，表示时钟源质量。1-99：非常差的时钟源，只有在内核启动或万不得已时采用；100-199：实际可以使用的时钟源；300-399：快速且准确的时钟源；400-499：完美理想的时钟源。rating 计算公式为 200 + mips_hpt_frequency / 10000000（10MHz），mips_hpt_frequency 表示时钟频率，即频率（Hz）每增加 10MHz，rating 值加 1（基数为 200）。

●**enable**：使能时钟源函数指针。

●**disable**：关闭时钟源函数指针。

●**flags**：标记成员，表示时钟源特性，取值定义如下（/include/linux/clocksource.h）：

```
#define   CLOCK_SOURCE_IS_CONTINUOUS   0x01 /*连续时钟源，高分辨率时钟源必须设置*/
#define   CLOCK_SOURCE_MUST_VERIFY     0x02

#define   CLOCK_SOURCE_WATCHDOG                0x10
#define   CLOCK_SOURCE_VALID_FOR_HRES    0x20   /*时钟源是否支持高分辨率特性*/
                                  /*这是启用低分辨率动态时钟和高分辨率时钟的必要条件*/
#define   CLOCK_SOURCE_UNSTABLE                0x40   /*不稳定时钟源*/
#define   CLOCK_SOURCE_SUSPEND_NONSTOP    0x80    /*系统睡眠时不停止*/
#define   CLOCK_SOURCE_RESELECT                0x100
```

●**mult、shift**：将计数器值转换成纳秒时间值的乘数和右移的位数。

clocksource_cyc2ns()函数用于计算纳秒时间值，函数定义在/include/linux/clocksource.h 头文件：

```
static inline s64 clocksource_cyc2ns(cycle_t cycles, u32 mult, u32 shift)
/*cycles：一段时间间隔内的硬件时钟周期数（计数值）*/
{
    return ((u64) cycles * mult) >> shift;    /*mult、shift 为 clocksource 结构体成员值*/
}
```

在注册时钟源时，注册函数根据传递的时钟源频率值计算 mult 和 shift 值，并赋予 clocksource 实例。

下面来看一下 mult 和 shift 值的含义。要计算 cycles 个时钟周期表示多少个纳秒，最简单的方法就是用 cycles 乘以一个时钟周期（单位纳秒）。但是，如果一个时钟周期长度小于一个纳秒或时钟周期长度不是纳秒的整数倍，将会引入小数运算。为此选择合适的 mult 个纳秒时长，时长内包含 2^{shift} 个时钟周期数，mult 乘以 cycles 再除以 2^{shift}，就计算得出 cycles 个时钟周期所含纳秒数，计算公式如下：

$$ns = cycles * \frac{mult}{2^{shift}} = (cycles * mult) >> shift$$

在上面的公式中可先计算 cycles * mult 值，再将结果除以 2^{shift}，而除法可以用移位操作代替。mult 值的含义就是选取的适当时长，单位纳秒，2^{shift} 表示 mult 时长内时钟周期数量。

内核定义了初始的以 jiffies 为计数器的 clocksource_jiffies 时钟源，在初始化时关联到 tk_core 实例。随后，体系结构或平台相关代码注册时钟源 clocksource 实例时，内核会自动将 tk_core 实例关联到最高质量的时钟源。下面先介绍计时器的初始化，后面再介绍时钟源的注册。

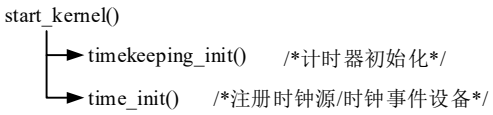
2 计时器初始化

计时器正常工作需要关联到时钟源，内核启动阶段会初始化计时器 tk_core 实例，将其关联到由内核定义的 clocksource_jiffies 时钟源，这是必然存在的一个时钟源，它以 jiffies 为计数器。

内核在/kernel/time/jiffies.c 文件内，定义了以内核节拍 jiffies 计数值为时钟周期计数器的时钟源实例 clocksource_jiffies，如下所示：

```
static struct clocksource clocksource_jiffies = {
    .name           = "jiffies",
    .rating         = 1,            /*最低精度的时钟源*/
    .read       = jiffies_read,
    .mask           = 0xffffffff,       /*32bits*/
    .mult           = NSEC_PER_JIFFY << JIFFIES_SHIFT, /* details above */
    .shift          = JIFFIES_SHIFT,        /*通常为 8*/
    .max_cycles     = 10,
};
```

内核在启动函数 start_kernel()内调用 timekeeping_init()函数初始化 **tk_core** 实例，并关联初始时钟源 clocksource_jiffies 实例，随后调用体系结构相关的 time_init()函数注册时钟源和时钟事件设备，调用关系如下图所示：



下面先看 timekeeping_init()函数的定义（/kernel/time/timekeeping.c），time_init()函数后面再介绍：

```
void __init timekeeping_init(void)
{
    struct timekeeper *tk = &tk_core.timekeeper;
    struct clocksource *clock;
    unsigned long flags;
    struct timespec64 now, boot, tmp;     /*now：当前真实时间值，boot：系统启动时间值*/
    /*从由电池供电的持久时钟源（时钟设备 RTC）获取当前真实时间值，
    *由体系结构（板级）代码实现，没有持久时钟源，时间值返回 0。
    */
    read_persistent_clock64(&now);      /*从持久时钟源读取当前时间值，默认 now 初始化为 0*/
    if (!timespec64_valid_strict(&now)) {     /*判断 now 值有效性（值为 0 是有效的）*/
```

```
        ...
        now.tv_sec = 0;       /*若 now 无效则对其赋 0*/
        now.tv_nsec = 0;
    } else if (now.tv_sec || now.tv_nsec)     /*now 值不为 0*/
        persistent_clock_exists = true;       /*设置具有持久时钟源标记*/

    read_boot_clock64(&boot);       /*获取系统启动时的真实时间值*/
                /*获取系统启动时的真实时间值，体系结构（板级）代码实现，boot 默认为 0*/
    if (!timespec64_valid_strict(&boot)) {     /*如果 boot 中值无效，boot 赋 0*/
        ...
        boot.tv_sec = 0;
        boot.tv_nsec = 0;
    }

    raw_spin_lock_irqsave(&timekeeper_lock, flags);
    write_seqcount_begin(&tk_core.seq);
    ntp_init();         /*全局变量清零，/kernel/time/ntp.c*/

    clock = clocksource_default_clock();     /*/kernel/time/jiffies.c*/
                /*如果体系结构没有定义此函数，返回 clocksource_jiffies 时钟源实例*/
    if (clock->enable)
        clock->enable(clock);         /*使能时钟源*/
    tk_setup_internals(tk, clock);
                /*tk_core 实例关联 clocksource_jiffies 时钟源，/kernel/time/timekeeping.c*/

    tk_set_xtime(tk, &now);     /*将 now 保存的当前真实时间值写入 tk->xtime_sec*/
    tk->raw_time.tv_sec = 0;        /*原始单调递增时间初始化为 0*/
    tk->raw_time.tv_nsec = 0;
    if (boot.tv_sec == 0 && boot.tv_nsec == 0)        /*boot 通常为 0*/
        boot = tk_xtime(tk);                /*从 tk->tkr_mono 获取当前真实时间赋予 boot*/

    set_normalized_timespec64(&tmp, -boot.tv_sec, -boot.tv_nsec); /*设置 tmp 值，/kernel/time/time.c*/
    tk_set_wall_to_mono(tk, tmp);   /*设置时间偏移量，/kernel/time/timekeeping.c*/

    timekeeping_update(tk, TK_MIRROR);
                                    /*更新时间基准等，/kernel/time/timekeeping.c*/
    write_seqcount_end(&tk_core.seq);
    raw_spin_unlock_irqrestore(&timekeeper_lock, flags);
}
```

timekeeping_init()函数执行以下工作：

（1）从系统持久时钟源，即由电池供电的外部时钟设备，读取当前时间值至 now（此时函数由体系结构或板级相关代码实现），此时间为当前真实时间值。如果没有持久时钟源则 now 为 0。

（2）调用 read_boot_clock64()函数获取系统启动时的真实时间值保存至 boot，若体系结构没有定义此函数 boot 为 0（若不为 0，boot 应小于 now）。系统启动时间为单调递增时间起点。

（3）关联默认的 clocksource_jiffies 时钟源，将 now 保存的真实时间值写入 tk_core 实例。

（4）原始单调递增时间 **tk->raw_time** 设为 0，以时间管理子系统初始化时间为起点。

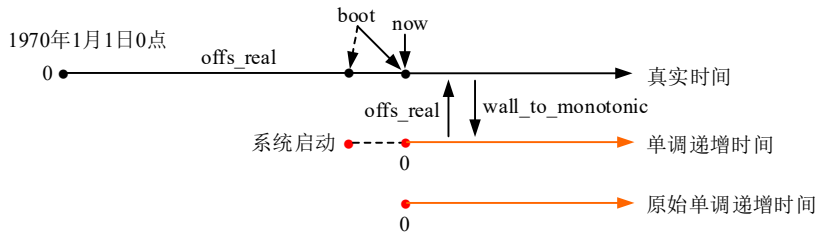
（5）若 boot 为 0，从 tk->tkr_mono 获取当前真实时间赋予 boot。boot 即保存当前真实时间，boot 表示系统启动时的真实时间。boot 若为 0 则当前时间视为系统启动时间。

（5）用-boot 设置 tmp 局部变量（timespec64 实例），表示真实时间到单调递增时间的偏移量。当前真实时间加上 tmp（负值）即得到当前单调递增时间。

将 tmp 表示的偏移量写入 tk->wall_to_monotonic 成员（负值），将-tmp 值写入 tk->offs_real 成员等。

（6）更新计时器时间基准。初始化时设置了真实时间、原始单调递增时间的起始值，要将其写入对应 tk_read_base.base 成员中并更新单调递增时间值。

下图示意了真实时间、单调递增时间、原始单调递增时间的关系：



如上图所示，真实时间转单调递增时间就是加上 tk->wall_to_monotonic 值，即减去启动时的真实时间值，单调递增时间转真实时间就是加上 tk->offs_real 值，即加上系统启动时的真实时间。如果系统没有设置系统启动时间，则当前时间设为系统启动时间。

原始单调递增时间以时间管理初始化时为起点。

timekeeping_init()函数调用 **tk_setup_internals(tk, clock)**函数建立 tk_core.timekeeper 实例与默认时钟源 clocksource_jiffies 的关联，函数定义见下文。

timekeeping_init()函数调用 tk_set_xtime(tk, &now)函数，将当前真实时间值 now 写入 timekeeper 实例中相关成员，函数定义如下（/kernel/time/timekeeping.c）：

```
static void tk_set_xtime(struct timekeeper *tk, const struct timespec64 *ts)
/*ts：当前真实时间值*/
{
  tk->xtime_sec = ts->tv_sec;          /*设置真实时间值，秒数*/
  tk->tkr_mono.xtime_nsec = (u64)ts->tv_nsec << tk->tkr_mono.shift;
                                       /*剩余不足 1 秒的纳秒数乘以 2^shift，即 ts->tv_nsec*2^shift*/
}
```

tk->xtime_sec 成员记录当前真实时间值的秒数，tk->tkr_mono.xtime_nsec 记录了不足 1 秒剩余的纳秒数乘以 2^{shift}。

如果 boot 表示的时间值为 0（通常为 0），则调用 tk_xtime()函数从 tk_core.timekeeper 中读取当前真实时间值至 boot，视为系统启动时间。tk_xtime()函数定义如下：

```
static inline struct timespec64 tk_xtime(struct timekeeper *tk)      /*获取当前真实时间值*/
{
```

```
    struct timespec64 ts;
    ts.tv_sec = tk->xtime_sec;      /*真实时间秒数*/
    ts.tv_nsec = (long)(tk->tkr_mono.xtime_nsec >> tk->tkr_mono.shift);
                                                    /*真实时间中不足 1 秒的纳秒数*/
    return ts;
}
```

timekeeping_init()函数最后调用 tk_set_wall_to_mono()函数和 timekeeping_update()函数设置 tk_core 实例中的各时间偏移量和计时器基准时间。下面先介绍前面关联时钟源的 tk_setup_internals()函数实现，然后再介绍这两个函数实现。

■关联时钟源

tk_setup_internals()函数用于建立计时器与时钟源的关联，函数定义如下：

```
static void tk_setup_internals(struct timekeeper *tk, struct clocksource *clock)
/*clock：时钟源*/
{
    cycle_t interval;
    u64 tmp, ntpinterval;
    struct clocksource *old_clock;        /*原使用的时钟源*/
     /*单调递增时间和原始单调递增时间采用相同的时钟源*/
    old_clock = tk->tkr_mono.clock;       /*单调递增时间原来使用的时钟源*/
    tk->tkr_mono.clock = clock;           /*设置单调递增时间使用的时钟源*/
    tk->tkr_mono.read = clock->read;      /*赋予时钟源的 read()函数*/
    tk->tkr_mono.mask = clock->mask;
    tk->tkr_mono.cycle_last = tk->tkr_mono.read(clock);   /*设置当前时钟源计数器值*/

    tk->tkr_raw.clock = clock;       /*设置原始单调递增时间使用的时钟源*/
    tk->tkr_raw.read = clock->read;
    tk->tkr_raw.mask = clock->mask;
    tk->tkr_raw.cycle_last = tk->tkr_mono.cycle_last;      /*设置当前时钟源计数器值*/

    tmp = NTP_INTERVAL_LENGTH;     /*(NSEC_PER_SEC/HZ)，一个节拍包含的纳秒数*/
    tmp <<= clock->shift;     /*一个节拍内时钟周期数量 cycle 乘 mult，即 cycle*mult*/
    ntpinterval = tmp;
    tmp += clock->mult/2;
    do_div(tmp, clock->mult);   /*tmp 表示一个节拍内时钟周期数量*/
    if (tmp == 0)
        tmp = 1;

    interval = (cycle_t) tmp;
    tk->cycle_interval = interval;     /*一个节拍内时钟周期数量*/
```

```
    tk->xtime_interval = (u64) interval * clock->mult;      /*一个节拍纳秒数乘以 mult，即 tmp*mult*/
    tk->xtime_remainder = ntpinterval - tk->xtime_interval;   /*cycle*mult 与 tmp*2^shift 差值*/
    tk->raw_interval =((u64) interval * clock->mult) >> clock->shift; /*一个节拍包含的纳秒数，取整*/

     /*如果单调递增时间更改了时钟源*/
    if (old_clock) {
        int shift_change = clock->shift - old_clock->shift;
        if (shift_change < 0)
            tk->tkr_mono.xtime_nsec >>= -shift_change;
        else
            tk->tkr_mono.xtime_nsec <<= shift_change;
    }
    tk->tkr_raw.xtime_nsec = 0;     /*纳秒数设为 0，因为 tk->raw_time 的单位就是纳秒，不是秒*/
    /*时钟源中的成中值赋予计时器*/
    tk->tkr_mono.shift = clock->shift;
    tk->tkr_raw.shift = clock->shift;

    tk->ntp_error = 0;
    tk->ntp_error_shift = NTP_SCALE_SHIFT - clock->shift;
    tk->ntp_tick = ntpinterval << tk->ntp_error_shift;

    tk->tkr_mono.mult = clock->mult;
    tk->tkr_raw.mult = clock->mult;
    tk->ntp_err_mult = 0;
}
```

tk_setup_internals()函数将 tk->tkr_mono、tk->tkr_raw 关联到新时钟源，并更新其成员值，注意需要读取新时钟源当前计数器值写入 tk->tkr_mono.cycle_last 和 tk->tkr_raw.cycle_last 成员。

■设置时间偏移量

timekeeping_init()函数在对各时间值设置了初始值后，调用 tk_set_wall_to_mono()函数设置 tk_core 实例中各时间值之间的偏移量：

```
static void tk_set_wall_to_mono(struct timekeeper *tk, struct timespec64 wtm)
/*wtm：真实时间至单调递增时间的偏移量，此处为负值，-boot（系统启动真实时间）*/
{
    struct timespec64 tmp;

    set_normalized_timespec64(&tmp, -tk->wall_to_monotonic.tv_sec,-tk->wall_to_monotonic.tv_nsec);
    WARN_ON_ONCE(tk->offs_real.tv64 != timespec64_to_ktime(tmp).tv64);
    tk->wall_to_monotonic = wtm;     /*真实时间至单调递增时间的偏移量，负数*/
    set_normalized_timespec64(&tmp, -wtm.tv_sec, -wtm.tv_nsec);  /*负负得正，正时间值*/
```

```
    tk->offs_real = timespec64_to_ktime(tmp);     /*单调递增时间至真实时间的偏移量，正数*/
    tk->offs_tai = ktime_add(tk->offs_real, ktime_set(tk->tai_offset, 0));        /*与 offs_real 相同*/
}
```

tk->wall_to_monotonic 的含义是真实时间加上 wall_to_monotonic 即得到单调递增时间，由于此值为负值，相当于减操作。

tk->offs_real 值的含义是单调递增时间转为真实时间的偏移量，真实时间值更大，单调递增时间加上 offs_real 值即得到真实时间。

■更新计时器

timekeeping_init()函数最后调用 timekeeping_update()函数在更新各时间值后，执行更新单调递增时间和原始单调递增时间基准，更新单调递增时间值等工作，函数定义如下：

```
static void timekeeping_update(struct timekeeper *tk, unsigned int action)
/*action：TK_MIRROR*/
{
    if (action & TK_CLEAR_NTP) {
        tk->ntp_error = 0;
        ntp_clear();
    }

    tk_update_leap_state(tk);       /*更新 tk->next_leap_ktime 时间值*/
    tk_update_ktime_data(tk);     /*更新时间基准等，见下文*/

    update_vsyscall(tk);
    update_pvclock_gtod(tk, action & TK_CLOCK_WAS_SET);     /*执行 pvclock_gtod_chain 通知链*/

    update_fast_timekeeper(&tk->tkr_mono, &tk_fast_mono);
                                                    /*用 tk->tkr_mono 填充 tk_fast_mono*/
    update_fast_timekeeper(&tk->tkr_raw,   &tk_fast_raw);
                                                    /*用 tk->tkr_raw 填充 tk_fast_raw*/

    if (action & TK_CLOCK_WAS_SET)        /*更改了计时器关联的时钟源*/
        tk->clock_was_set_seq++;
    if (action & TK_MIRROR)
        memcpy(&shadow_timekeeper, &tk_core.timekeeper, sizeof(tk_core.timekeeper));
                                        /*设置 shadow_timekeeper 与 tk_core.timekeeper 相同*/
}
```

tk_update_ktime_data()函数用于更新单调递增时间和原始单调递增时间的基准值等，函数定义如下：

```
static inline void tk_update_ktime_data(struct timekeeper *tk)
{
    u64 seconds;
```

```
    u32 nsec;

    seconds = (u64)(tk->xtime_sec + tk->wall_to_monotonic.tv_sec);   /*真实时间加偏移量*/
                                                                     /*真实时间转单调递增时间，秒数*/
    nsec = (u32) tk->wall_to_monotonic.tv_nsec;     /*纳秒数*/
    tk->tkr_mono.base = ns_to_ktime(seconds * NSEC_PER_SEC + nsec);
                                                                     /*设置单调递增时间基准值，纳秒*/
    tk->tkr_raw.base = timespec64_to_ktime(tk->raw_time);     /*设置原始单调递增时间基准值*/

    nsec += (u32)(tk->tkr_mono.xtime_nsec >> tk->tkr_mono.shift);     /*纳秒数*/
    if (nsec >= NSEC_PER_SEC)
        seconds++;
    tk->ktime_sec = seconds;      /*更新单调递增时间值，秒数*/
}
```

3 注册时钟源

由前文可知，内核定义了 clocksource_jiffies 时钟源，它是系统内精度最低的时钟源。体系结构或平台相关代码或时钟硬件驱动程序（/drivers/clocksource/）可定义并向内核注册 clocksource 实例。注册时钟源时，内核会自动选择最优的时钟源关联到计时器。

内核在/kernel/time/clocksource.c 文件内定义了当前使用时钟源指针和全局双链表，双链表用于管理所有注册的时钟源实例：

```
static struct clocksource   *curr_clocksource;      /*当前最优时钟源实例指针*/
static   LIST_HEAD(clocksource_list);               /*全局双链表头*/
static char override_name[CS_NAME_LEN];     /*用户指定使用的时钟源（名称）*/
```

用户可以通过"clocksource=xxx"或"clock=xxx"命令行参数指定计时器使用时钟源的名称。

向内核注册 clocksource 实例的函数有（/include/linux/clocksource.h）：

●int **clocksource_register_hz**(struct clocksource *cs, u32 hz)：参数 hz 表示时钟源计数频率，单位 Hz；

●int **clocksource_register_khz**(struct clocksource *cs, u32 khz)：参数 hz 表示时钟源计数频率，单位 KHz；

注册函数直接将工作转交给__clocksource_register_scale()函数执行（/kernel/time/clocksource.c）：

```
int __clocksource_register_scale(struct clocksource *cs, u32 scale, u32 freq)
/*scale：频率单位为 Hz 时为 1，为 KHz 时为 1000；freq：时钟源计数频率，单位为 hz 或 khz*/
{

    /*初始化时钟源 mult、shift、max_idle_ns 成员值*/
    __clocksource_update_freq_scale(cs, scale, freq);   /*/kernel/time/clocksource.c*/

    mutex_lock(&clocksource_mutex);
```

```
        clocksource_enqueue(cs);            /*以 rating 值降序将实例添加到 clocksource_list 双链表*/
        clocksource_enqueue_watchdog(cs);   /*没有选配 CLOCKSOURCE_WATCHDOG 选项，空操作*/
        clocksource_select();       /*选择最优时钟源关联到 tk_core 实例，/kernel/time/clocksource.c*/
        mutex_unlock(&clocksource_mutex);
        return 0;
    }
```

__clocksource_register_scale()函数调用__clocksource_update_freq_scale()函数计算时钟源实例 mult、shift 和 max_idle_ns 成员值，然后调用 clocksource_enqueue(cs)函数以 rating 值从高到低将时钟源实例插入 clocksource_list 全局双链表，最后调用 clocksource_select()函数选择最优的时钟源关联到计时器 **tk_core** 实例（见下文）。

■选择时钟源

在向内核注册时钟源 clocksource 实例的函数中将调用 **clocksource_select()**函数选择最优的时钟源关联到计时器，clocksource_select()函数定义如下（/kernel/time/clocksource.c）：

```
static void clocksource_select(void)
{
    return   __clocksource_select(false);
}
```

__clocksource_select()函数定义如下：

```
static void __clocksource_select(bool skipcur)
/*skipcur：是否跳过当前使用的时钟源（curr_clocksource），此处为 false*/
{
    bool oneshot = tick_oneshot_mode_active();   /*当前 CPU 核时钟设备是否是单触发模式*/
    struct clocksource *best, *cs;

    best = clocksource_find_best(oneshot, skipcur); /*选择质量最好的时钟源（链表中第一个成员）*/
    if (!best)
        return;

    /*检查是否是 override_name 指定的时钟源（若 override_name 不为空）*/
    list_for_each_entry(cs, &clocksource_list, list) {      /*遍历双链表*/
        if (skipcur && cs == curr_clocksource)
            continue;
        if (strcmp(cs->name, override_name) != 0)       /*不是指定的时钟源，遍历下一个*/
            continue;
        /*是指定名称的时钟源，时钟设备是单触发的，时钟源须支持高分辨率模式*/
        if (!(cs->flags & CLOCK_SOURCE_VALID_FOR_HRES) && oneshot) {
            ...
            override_name[0] = 0;
        } else
```

```
                best = cs;          /*找到了 override_name 指定的时钟源*/
                break;
        }

        if (curr_clocksource != best && !timekeeping_notify(best)) {     /*如果更改了时钟源*/
                ...
                curr_clocksource = best;   /*当前使用时钟源*/
        }
    }
```

__clocksource_select()函数首先查找全局双链表中质量最好的时钟源，如果 override_name 不为空，则再查找 override_name 名称的时钟源是否存在且可用，若是则用它，否则用前面找到的最好时钟源。最后，如果更改了计时器使用的时钟源，则调用 timekeeping_notify()函数，更改 tk_core.timekeeper 关联的时钟源。

timekeeping_notify()函数定义如下：

```
int timekeeping_notify(struct clocksource *clock)
{
        struct timekeeper *tk = &tk_core.timekeeper;

        if (tk->tkr_mono.clock == clock)
                return 0;
        stop_machine(change_clocksource, clock, NULL);     /*/kernel/stop_machine.c*/
        tick_clock_notify();         /*/kernel/time/tick-sched.c*/
        return tk->tkr_mono.clock == clock ? 0 : -1;
}
```

timekeeping_notify()函数向 CPU 停机工作中添加工作，执行函数为 change_clocksource()，函数定义如下：

```
static int change_clocksource(void *data)       /*data：clocksource*/
{
        struct timekeeper *tk = &tk_core.timekeeper;
        struct clocksource *new, *old;
        unsigned long flags;

        new = (struct clocksource *) data;

        raw_spin_lock_irqsave(&timekeeper_lock, flags);
        write_seqcount_begin(&tk_core.seq);

        timekeeping_forward_now(tk);    /*更新计时器各时间值*/
        if (try_module_get(new->owner)) {
                if (!new->enable || new->enable(new) == 0) {     /*使能时钟源*/
                        old = tk->tkr_mono.clock;
```

```
            tk_setup_internals(tk, new);     /*关联新时钟源*/
            if (old->disable)
                old->disable(old);     /*关闭旧时钟源*/
            module_put(old->owner);
        } else {
            module_put(new->owner);
        }
    }
    timekeeping_update(tk, TK_CLEAR_NTP | TK_MIRROR | TK_CLOCK_WAS_SET);
                                                    /*更新时间基准值等，见上文*/
    write_seqcount_end(&tk_core.seq);
    raw_spin_unlock_irqrestore(&timekeeper_lock, flags);

    return 0;
}
```

在选择了新时钟源后，需要切换计时器关联到新时钟源。切换流程如下：利用原来的时钟源计算出当前的时间值并写入计时器，使能新时钟源，关联新时钟源，关闭旧时钟源，最后更新单调递增时间和原始单调递增时间基准值。

change_clocksource()函数调用 timekeeping_forward_now(tk)函数用于更新计时器时间值，最后调用函数 timekeeping_update()更新计时器中时间基准值等。

timekeeping_update()函数前面介绍过，下面看一下更新计时器时间值的 timekeeping_forward_now(tk)函数的定义：

```
static void timekeeping_forward_now(struct timekeeper *tk)
{
    struct clocksource *clock = tk->tkr_mono.clock;        /*单调递增时间关联时钟源*/
    cycle_t cycle_now, delta;
    s64 nsec;

    cycle_now = tk->tkr_mono.read(clock);       /*读取时钟源计数器值*/
    delta = clocksource_delta(cycle_now, tk->tkr_mono.cycle_last, tk->tkr_mono.mask);
                                                    /*计数值差值*/
    tk->tkr_mono.cycle_last = cycle_now;       /*更新计数器值*/
    tk->tkr_raw.cycle_last  = cycle_now;       /*更新计数器值*/

    tk->tkr_mono.xtime_nsec += delta * tk->tkr_mono.mult;     /*累加到纳秒数，即 delta*mult*/

    /*加上体系结构定义的偏移量*/
    tk->tkr_mono.xtime_nsec += (u64)arch_gettimeoffset() << tk->tkr_mono.shift;

    tk_normalize_xtime(tk);    /*将 tk->tkr_mono.xtime_nsec 中的秒数累加到 tk->xtime_sec 值*/
```

```
    nsec = clocksource_cyc2ns(delta, tk->tkr_raw.mult, tk->tkr_raw.shift);    /*经历的纳秒数*/
    timespec64_add_ns(&tk->raw_time, nsec);      /*经历的纳秒数累加到原始单调递增时间*/
}
```

timekeeping_forward_now(tk)函数通过单调递增时间关联的时钟源，计算出本次更新与上次更新之间的计数器差值，转为时间值。将时间值中的整秒数累加到 tk->xtime_sec 成员上，表示真实时间，将纳秒表示的时间值累加到 tk->raw_time 表示的原始单调递增时间。

4 接口函数

时间管理子系统初始化后，内核通过单调递增时间跟踪真实时间。内核在周期节拍工作中会定期更新计时器时间值。

时间管理子系统向内核提供了获取当前时间值的接口函数，并向用户进程提供了设置/获取真实时间的系统调用。设置真实时间时会同步更新真实时间与单调递增时间值之间的偏移量。

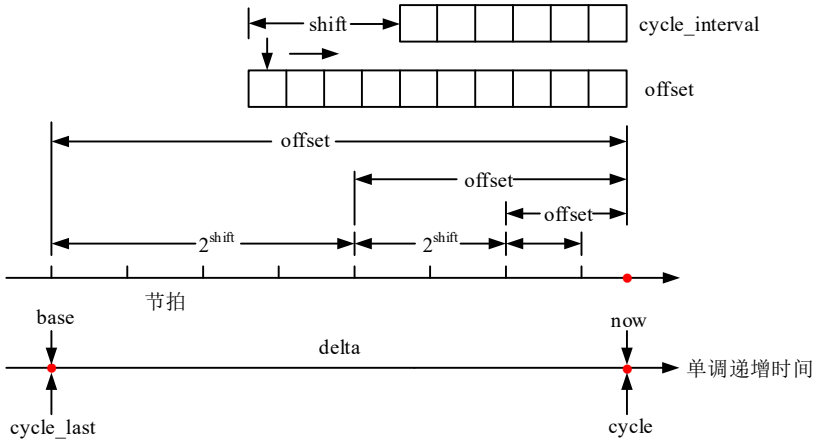
■更新时间值

内核在全局周期时钟事件（节拍，时钟中断）中将调用 **update_wall_time()**函数更新计时器时间值，函数定义如下（/kernel/time/timekeeping.c）：

```
void update_wall_time(void)
{
    struct timekeeper *real_tk = &tk_core.timekeeper;      /*计时器*/
    struct timekeeper *tk = &shadow_timekeeper;            /*影子计时器*/
    cycle_t offset;
    int shift = 0, maxshift;
    unsigned int clock_set = 0;
    unsigned long flags;

    raw_spin_lock_irqsave(&timekeeper_lock, flags);

    if (unlikely(timekeeping_suspended))        /*计时器是否暂停*/
        goto out;

#ifdef CONFIG_ARCH_USES_GETTIMEOFFSET
    offset = real_tk->cycle_interval;
#else
    offset = clocksource_delta(tk->tkr_mono.read(tk->tkr_mono.clock),  \
                               tk->tkr_mono.cycle_last, tk->tkr_mono.mask);
                                    /*单调递增时间时钟源计数值差值*/
#endif

    /*更新间隔没有超过一个节拍，返回*/
    if (offset < real_tk->cycle_interval)
        goto out;
```

```
        /*时间间隔大于等于一个节拍，启用动态时钟时可能超过一个节拍，安全检查*/
        timekeeping_check_update(real_tk, offset);

        /*shift 用于间隔时间大于一个节拍时的情况*/
        shift = ilog2(offset) - ilog2(tk->cycle_interval);          /*有效比特位数差值*/
        shift = max(0, shift);
        /*Bound shift to one less than what overflows tick_length*/
        maxshift = (64 - (ilog2(ntp_tick_length())+1)) - 1;
        shift = min(shift, maxshift);
        while (offset >= tk->cycle_interval) {     /*间隔不小于一个节拍*/
            offset = logarithmic_accumulation(tk, offset, shift,&clock_set);  /*更新时间值，见下文*/
            if (offset < tk->cycle_interval<<shift)
                shift--;
        }

        /* correct the clock when NTP error is too big */
        timekeeping_adjust(tk, offset);

        old_vsyscall_fixup(tk);

        clock_set |= accumulate_nsecs_to_secs(tk);   /*返回 TK_CLOCK_WAS_SET*/
                        /*将 tk->tkr_mono.xtime_nsec 中的整秒数累加到 tk->xtime_sec，剩余纳秒数保留*/
        write_seqcount_begin(&tk_core.seq);

        timekeeping_update(tk, clock_set);     /*更新时间基准等，见上文*/
        memcpy(real_tk, tk, sizeof(*tk));
        write_seqcount_end(&tk_core.seq);
    out:
        raw_spin_unlock_irqrestore(&timekeeper_lock, flags);
        if (clock_set)
            clock_was_set_delayed();
    }
```

如果启用了动态时钟时，在 CPU 核无事可做时会暂停周期节拍，当再次激活周期节拍时，可能经历了多个节拍的时间间间隔，如下图所示。上面函数代码中 shift 的含义是将经历时长按 2^{shift} 个整数节拍进行划分，shift 从高到低遍历 offset 中置位的二进制位。offset 表示整个时间间隔中时钟源计数器值所增加的值，即所经历的时钟周期数。logarithmic_accumulation()函数用于计算 2^{shift} 个节拍间隔所经历的时间，并更新到计时器中，返回原 offset 中除去本次计算节拍内包含时钟周期数所剩余的周期计数，用于下次计算。



logarithmic_accumulation()用于更新单调递增时间值和原始单调递增时间值，函数定义如下：

```
static cycle_t logarithmic_accumulation(struct timekeeper *tk, cycle_t offset, \
                                        u32 shift,unsigned int *clock_set)
/*offset：时间间隔中周期计数器差值，shift：本次计算 2^shift 个节拍的时间间隔*/
{
    cycle_t interval = tk->cycle_interval << shift;      /*2^shift 个节拍包含的周期计数值，相当于乘 2^shift*/
    u64 raw_nsecs;

    /*间隔小于 cycle_interval*2^shift*/
    if (offset < interval)
        return offset;          /*返回原计数值，进行下一轮计算，shift 减 1*/

    offset -= interval;      /*offset 表示剩余的周期计数，作为函数返回值*/
    tk->tkr_mono.cycle_last += interval;    /*设置周期计数值*/
    tk->tkr_raw.cycle_last  += interval;

    tk->tkr_mono.xtime_nsec += tk->xtime_interval << shift;    /*节拍转纳秒值*/
    *clock_set |= accumulate_nsecs_to_secs(tk);    /*返回 TK_CLOCK_WAS_SET*/
                /*将 tk->tkr_mono.xtime_nsec 中的整秒数累加 tk->xtime_sec，剩余纳秒数保留*/

    /*更新原始单调递增时间*/
    raw_nsecs = (u64)tk->raw_interval << shift;      /*经历的纳秒数*/
    raw_nsecs += tk->raw_time.tv_nsec;       /*累加经历的纳秒数*/
    if (raw_nsecs >= NSEC_PER_SEC) {
        u64 raw_secs = raw_nsecs;
        raw_nsecs = do_div(raw_secs, NSEC_PER_SEC);
        tk->raw_time.tv_sec += raw_secs;    /*整秒数累加到 tk->raw_time.tv_sec */
    }
    tk->raw_time.tv_nsec = raw_nsecs;    /*剩余纳秒数*/

    /* Accumulate error between NTP and clock interval */
```

```
        tk->ntp_error += tk->ntp_tick << shift;
        tk->ntp_error -= (tk->xtime_interval + tk->xtime_remainder) <<(tk->ntp_error_shift + shift);

        return offset;      /*返回剩余时钟周期计数值*/
    }
```

综上所述，update_wall_time()函数计算本次更新与上次更新之间所经历的整数节拍时长，未满一个节拍的周期计数留着下次计算。将所经历的时间累加到真实时间 tk->xtime_sec 中（整的秒数）和原始单调递增时间 tk->raw_time 中，然后用当前真实时间（减偏移量）和原始单调递增时间更新单调递增时间和原始单调递增时间的基准值，以及单调递增时间值。

■获取单调递增时间

内核接口函数 ktime_get()用于获取当前单调递增时间值，函数定义如下（/kernel/time/timekeeping.c）：

```
ktime_t ktime_get(void)
{
    struct timekeeper *tk = &tk_core.timekeeper;
    unsigned int seq;
    ktime_t base;
    s64 nsecs;

    WARN_ON(timekeeping_suspended);

    do {
        seq = read_seqcount_begin(&tk_core.seq);
        base = tk->tkr_mono.base;              /*单调递增时间基准*/
        nsecs = timekeeping_get_ns(&tk->tkr_mono);   /*时间间隔的纳秒值*/

    } while (read_seqcount_retry(&tk_core.seq, seq));
    return ktime_add_ns(base, nsecs);      /*基准时间加上时间间隔作为返回值，base 值并没有改变*/
}
```

ktime_get()函数以单调递增时间为基准，调用 timekeeping_get_ns()函数获取时钟源计数器差值，并将其转换成纳秒的时间间隔，在基准时间 base 加上时间间隔就得到当前单调递增时间值，单位纳秒。

注意：ktime_get()函数并没有更新计时器中任何成员值，更新操作在更新时间值时执行，而不会在读取时间值时执行。

其它的接口函数定义在/kernel/time/timekeeping.c 文件内，例如，**ktime_get_raw()**用于读取原始单调递增时间值，请读者自行阅读源代码。

■设置真实时间

内核启动时会获取当前的真实时间写入计时器，然后内核通过单调递增时间来跟踪真实时间，它们是同步的。用户可通过系统调用设置和获取当前真实时间，设置当前真实时间时会同时更新其与单调递增时间之间的偏移量，因为它们必须是同步的。

内核在/kernel/time/time.c 文件内实现了设置和获取真实时间的系统调用，在用户空间真实时间由结构体 timeval 和 timezone 表示，timeval 结构体内用秒和微秒来表示时间，timezone 结构体表示时区的信息，但现在弃用了，系统调用中设为 NULL。

设置真实时间的系统调用为 settimeofday()，实现函数如下：

```
SYSCALL_DEFINE2(settimeofday, struct timeval __user *, tv,struct timezone __user *, tz)
{
    struct timeval    user_tv;          /*秒、微秒*/
    struct timespec    new_ts;          /*秒、纳秒*/
    struct timezone    new_tz;

    if (tv) {
        if (copy_from_user(&user_tv, tv, sizeof(*tv)))     /*复制 timeval 信息至内核空间*/
            return -EFAULT;

        if (!timeval_valid(&user_tv))
            return -EINVAL;

        new_ts.tv_sec = user_tv.tv_sec;     /*转换成由 timespec 实例表示的时间，秒*/
        new_ts.tv_nsec = user_tv.tv_usec * NSEC_PER_USEC;          /*纳秒*/
    }
    if (tz) {
        ...
    }

    return do_sys_settimeofday(tv ? &new_ts : NULL, tz ? &new_tz : NULL);       /*/kernel/time/time.c*/
}
```

在复制完时间信息后，系统调用将工作转交给 do_sys_settimeofday()函数，它调用 **do_settimeofday**(tv)函数完成真实时间的设置。do_settimeofday()函数定义如下（/include/linux/timekeeping.h）：

```
static inline int do_settimeofday(const struct timespec *ts)
{
    struct timespec64 ts64;

    ts64 = timespec_to_timespec64(*ts);       /*timespec 转 timespec64*/
    return do_settimeofday64(&ts64);          /*/kernel/time/timekeeping.c*/
}
```

do_settimeofday64()函数定义如下：

```
int do_settimeofday64(const struct timespec64 *ts)
{
    struct timekeeper *tk = &tk_core.timekeeper;              /*计时器*/
    struct timespec64 ts_delta, xt;
    unsigned long flags;
```

```
    if (!timespec64_valid_strict(ts))       /*ts 有效性判断*/
        return -EINVAL;

    raw_spin_lock_irqsave(&timekeeper_lock, flags);
    write_seqcount_begin(&tk_core.seq);

    timekeeping_forward_now(tk);        /*更新计时器时间值*/

    xt = tk_xtime(tk);     /*当前真实时间值，秒*/
    ts_delta.tv_sec = ts->tv_sec - xt.tv_sec;       /*设置时间与当前真实时间差值*/
    ts_delta.tv_nsec = ts->tv_nsec - xt.tv_nsec;

    tk_set_wall_to_mono(tk, timespec64_sub(tk->wall_to_monotonic, ts_delta));
                                        /*设置新的真实时间与单调递增时间之间的偏移量*/

    tk_set_xtime(tk, ts);     /*设置计时器中真实时间值*/
    timekeeping_update(tk, TK_CLEAR_NTP | TK_MIRROR | TK_CLOCK_WAS_SET);
                                    /*更新单调递增时间（及基准值）、原始单调递增时间基准值等*/
    write_seqcount_end(&tk_core.seq);
    raw_spin_unlock_irqrestore(&timekeeper_lock, flags);

    /*通知定时器真实时间改变了*/
    clock_was_set();       /*/kernel/time/hrtimer.c*/
    return 0;
}
```

设置真实时间的主要工作是用新的真实时间值设置 tk->xtime_sec 值，修改真实时间与单调递增时间之间的偏移量，并依此重新设置单调递增时间值和时间基准，以及原始单调递增时间值和时间基准。

最后调用的 clock_was_set()函数需要通知以真实时间值计时的高分辨定时器，真实时间改变了，需要做出相应的调整。

■获取真实时间

获取真实时间的系统调用为 settimeofday()，实现函数定义如下：

```
SYSCALL_DEFINE2(gettimeofday, struct timeval __user *, tv,struct timezone __user *, tz)
{
    if (likely(tv != NULL)) {
        struct timeval ktv;
        do_gettimeofday(&ktv);          /*/kernel/time/timekeeping.c*/
        if (copy_to_user(tv, &ktv, sizeof(ktv)))       /*向用户空间复制时间值*/
            return -EFAULT;
    }
```

```
    ...
    return 0;
}
```

do_gettimeofday()函数定义如下：

```
void do_gettimeofday(struct timeval *tv)
{
    struct timespec64 now;

    getnstimeofday64(&now);        /*/kernel/time/timekeeping.c*/
    tv->tv_sec = now.tv_sec;           /*秒*/
    tv->tv_usec = now.tv_nsec/1000;    /*纳秒转微秒*/
}
```

getnstimeofday64()函数调用__getnstimeofday64()函数从计时器中读取当前真实时间值，定义如下：

```
int __getnstimeofday64(struct timespec64 *ts)    /*/kernel/time/timekeeping.c*/
{
    struct timekeeper *tk = &tk_core.timekeeper;        /*计时器*/
    unsigned long seq;
    s64 nsecs = 0;

    do {
        seq = read_seqcount_begin(&tk_core.seq);
        ts->tv_sec = tk->xtime_sec;          /*当前真实时间值，秒数*/
        nsecs = timekeeping_get_ns(&tk->tkr_mono);          /*纳秒数*/

    } while (read_seqcount_retry(&tk_core.seq, seq));

    ts->tv_nsec = 0;
    timespec64_add_ns(ts, nsecs);          /*增加纳秒数，整秒数会累加到 ts->tv_sec*/

    if (unlikely(timekeeping_suspended))
        return -EAGAIN;
    return 0;
}
```

另外，内核还定义了调整真实时间的 adjtimex()系统调用，获取真实时间（秒数）的 time()系统调用等，请读者自行阅读源代码。

6.8.3 时钟设备

通用时间框架第二个主要的功能是为每个 CPU 核提供时钟设备，时钟设备可周期性地或按指定时间间隔触发时钟事件，并调用相应的时钟事件处理函数，在处理函数中完成内核例行性、周期性地工作。前面介绍的计时器是内核全局的，而时钟设备是特定于 CPU 核的，只为关联的 CPU 核服务。

时钟设备通过关联的时钟事件设备来实现以上功能，每个时钟设备关联一个时钟事件设备。可以向内

核注册多个时钟事件设备，内核会决定时钟设备关联哪个时钟事件设备。

时钟事件设备是对时钟硬件中断功能的抽象。通过对时钟事件设备的编程，可设置时钟硬件何时、以何种方式产生中断，中断由时钟事件设备关联的 CPU 核处理。中断处理函数中将调用时钟事件设备中指定的时钟事件处理函数。

系统中所有 CPU 核都有自己的时钟设备，有一个被用作全局时钟设备，负责内核一些公共的周期性工作，如更新全局变量 jiffies（jiffies64）值、更新计时器时间值等。

根据关联时钟事件设备产生中断的精度，时钟设备分为低分辨率时钟和高分辨率时钟。前者的定时精度较低（精度是一个节拍），后者能产生较高的定时精度，以适应对时间要求较高的应用。

根据时钟事件（中断）产生的形式，分为周期时钟和动态时钟（又称无时钟）。前者以固定的周期产生时钟事件，后者在 CPU 核无事可做时暂停周期时钟，以节省电能。

因此，时钟设备有 4 种工作模式，分别是：低分辨率周期时钟、低分辨率动态时钟、高分辨率周期时钟、高分辨率动态时钟。

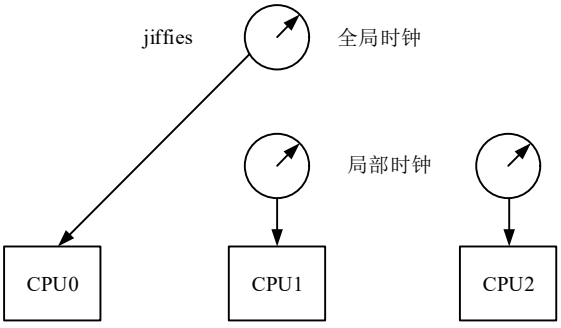
本节主要介绍时钟设备、时钟事件设备的定义，以及时钟事件设备的注册，下节将介绍时钟设备的各种工作模式。

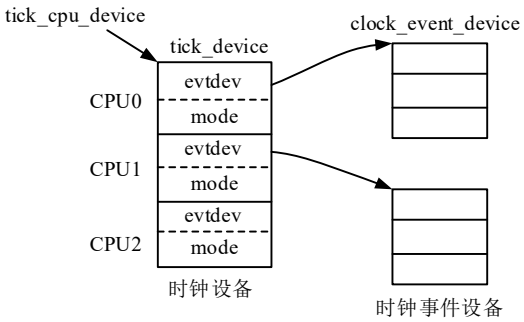
1 数据结构

每个 CPU 核具有一个时钟设备，时钟设备关联到一个时钟事件设备，负责产生和处理时钟事件。时钟设备由内核定义，时钟事件设备由体系结构（平台）相关代码注册，注册时会关联到时钟设备。

■时钟设备

不管在单核还是多核处理器系统中，每个 CPU 核都有专属于自己的时钟设备，用于触发和处理时钟事件。CPU 核专属的时钟设备触发的时钟事件称之为局部时钟，在此时钟事件中将处理关联 CPU 核的工作，如进程调度等。另外，内核还需要一个全局时钟，用于处理内核公共事务，例如更新 jiffies（jiffies64）值、计时器时间值等。全局时钟通常由某一 CPU 核的局部时钟充当，如下图所示。





时钟设备由 tick_device 结构体表示，结构体定义在/kernel/time/tick-sched.h 头文件：

```
struct tick_device {
    struct clock_event_device *evtdev;    /*指向关联的时钟事件设备，注意是指针*/
    enum tick_device_mode mode;           /*时钟设备模式*/
};
```

时钟设备模式由枚举类型表示如下（/kernel/time/tick-sched.h）：

```
enum tick_device_mode {
    TICKDEV_MODE_PERIODIC,    /*周期模式*/
```

```
    TICKDEV_MODE_ONESHOT,    /*单触发模式*/
};
```

内核在/kernel/time/tick-common.c 文件内定义了 percpu 变量，即各 CPU 核时钟设备 tick_device 实例：

```
DEFINE_PER_CPU(struct tick_device, tick_cpu_device);  /*各 CPU 核对应的 tick_device 实例*/
```

内核在/kernel/time/tick-common.c 文件内还定义了以下全局变量：

```
ktime_t   tick_next_period;
ktime_t   tick_period;
int   tick_do_timer_cpu __read_mostly = TICK_DO_TIMER_BOOT;   /*初始值设为-2*/
```

●tick_next_period：全局时钟产生下一个时钟事件的时间间隔（多久以后），单位为纳秒。

●**tick_period**：内核节拍周期值，单位为纳秒，由 HZ 常数计算而得。

●tick_do_timer_cpu：CPU 核编号，该 CPU 核局部时钟用作全局时钟。

■时钟事件设备

时钟事件设备是对时钟硬件产生时钟事件（中断）功能的抽象表示。时钟事件设备由 clock_event_device 结构体表示，定义在/include/linux/clockchips.h 头文件：

```
struct clock_event_device {
    void   (*event_handler)(struct clock_event_device *);   /*时钟事件处理函数，硬件中断中调用*/
    int (*set_next_event)(unsigned long evt, struct clock_event_device *); /*设置下一个时钟事件*/
    int (*set_next_ktime)(ktime_t expires, struct clock_event_device *);   /*设置下一个时钟事件*/
    ktime_t          next_event;    /*在单触发模式时，表示下次事件发生的时间，纳秒值*/
    u64              max_delta_ns;       /*时钟事件最大时间间隔，纳秒值*/
    u64              min_delta_ns;       /*时钟事件最小时间间隔，纳秒值*/
    u32              mult;               /*纳秒数转为周期计数的乘数（与时钟源相反）*/
    u32              shift;              /*纳秒数转为周期计数的右移位数*/
    enum   clock_event_mode    mode;                  /*时钟事件设备工作模式*/
    enum   clock_event_state   state_use_accessors;   /*时钟事件设备状态*/
    unsigned int         features;                    /*时钟事件设备具有的物理特性*/
    unsigned long        retries;
    void   (*set_mode)(enum clock_event_mode mode, struct clock_event_device *);   /*设置工作模式*/
    int   (*set_state_periodic)(struct clock_event_device *);    /*设为周期状态，!set_mode()*/
    int   (*set_state_oneshot)(struct clock_event_device *);     /*设为单触发状态，!set_mode()*/
    int   (*set_state_oneshot_stopped)(struct clock_event_device *);
                                                  /*设为单次触发停止状态，!set_mode()*/
    int   (*set_state_shutdown)(struct clock_event_device *);    /*设为关闭模式，!set_mode()*/
    int   (*tick_resume)(struct clock_event_device *);          /*重启时钟事件设备，!set_mode()*/

    void   (*broadcast)(const struct cpumask *mask);          /*广播事件处理函数*/
    void   (*suspend)(struct clock_event_device *);           /*暂停时钟事件*/
```

```
    void   (*resume)(struct clock_event_device *);          /*重启时钟事件*/
    unsigned long       min_delta_ticks;     /*最小间隔节拍数*/
    unsigned long       max_delta_ticks;     /*最大间隔节拍数*
    const char      *name;       /*名称*/
    int         rating;          /*时钟事件设备质量*/
    int         irq;             /*时钟硬件中断号*/
    int         bound_on;        /*当前绑定的 CPU 核编号*/
    const struct cpumask    *cpumask;    /*时钟事件设备能绑定到哪些 CPU 核（CPU 亲和性）*/
    struct list_head    list;            /*双链表成员，将实例链接到全局链表*/
    struct module       *owner;
} ____cacheline_aligned;
```

clock_event_device 结构体中主要成员简介如下：

●**event_handler**：时钟事件处理函数，在注册时钟事件设备时，此函数由内核自动赋值，在时钟硬件中断处理函数中调用此函数，改变时钟设备工作模式时将修改此函数指针（指向不同的处理函数）。

●**set_next_event**：设置下一个时钟事件（硬件中断），时间间隔由时钟源周期计数值表示。

●**set_next_ktime**：设置下一个时钟事件（硬件中断），下次事件时间为单调递增时间值。

通用代码中并不需要直接调用以上两个函数，因为内核提供了一个辅助函数封装了这两个函数：

int clockevents_program_event(struct clock_event_device *dev, ktime_t expires,bool force);

expires 表示下一个时钟事件的到期时间（纳秒），force 表示如果没有设置 expires 参数，是否以最小间隔设置下一个时钟事件，函数成功返回 0，不成功返回错误码。

●**mode**：时钟事件设备工作模式，定义如下（/include/linux/clockchips.h）：

```
enum clock_event_mode {
    CLOCK_EVT_MODE_UNUSED,          /*时钟事件设备未使用*/
    CLOCK_EVT_MODE_SHUTDOWN,        /*关闭模式*/
    CLOCK_EVT_MODE_PERIODIC,        /*周期工作模式，周期性地产生中断*/
    CLOCK_EVT_MODE_ONESHOT,         /*单次触发模式，处理函数中需要重新对设备编程*/
    CLOCK_EVT_MODE_RESUME,
};
```

●**state_use_accessors**：时钟事件设备工作状态（/include/linux/clockchips.h）：

```
enum clock_event_state {
    CLOCK_EVT_STATE_DETACHED,     /*未被通用时间框架采用（未关联时钟设备）*/
    CLOCK_EVT_STATE_SHUTDOWN,     /*关闭状态*/
    CLOCK_EVT_STATE_PERIODIC,         /*周期工作状态，周期性产生中断*/
    CLOCK_EVT_STATE_ONESHOT,          /*单触发工作状态，每次触发后需要重新编程*/
    CLOCK_EVT_STATE_ONESHOT_STOPPED,      /*处于单触发状态，但暂时为停止状态*/
};
```

时钟事件设备工作于周期模式，则在时钟事件处理函数中不需要对设备编程，它会自动周期性地产生时钟事件（中断）。如果工作于单触发模式，则在时钟事件处理函数中需要重新对设备进行编程，以便产生下一个事件，否则将不再产生时钟事件。

●**features**：标记时钟事件设备固有的特性，通常表示硬件的物理特性，取值如下：

```
# define   CLOCK_EVT_FEAT_PERIODIC        0x000001     /*可工作于周期模式*/
# define   CLOCK_EVT_FEAT_ONESHOT         0x000002     /*可工作于单触发模式*/
# define   CLOCK_EVT_FEAT_KTIME           0x000004     /*支持纳秒时间编程*/
# define   CLOCK_EVT_FEAT_C3STOP          0x000008     /*x86(64)*/
# define   CLOCK_EVT_FEAT_DUMMY           0x000010     /*x86(64)*/
# define   CLOCK_EVT_FEAT_DYNIRQ          0x000020     /*时钟中断可动态关联 CPU 核*/
# define   CLOCK_EVT_FEAT_PERCPU          0x000040     /*时钟中断可指定 CPU 核*/
# define   CLOCK_EVT_FEAT_HRTIMER         0x000080     /*时钟事件由高分辨率定时器驱动*/
```

●**set_mode**：设置时钟事件设备工作模式，模式参数为 clock_event_mode 枚举类型定义的模式。

●**set_state_xxx**：在没有定义 set_mode()函数时用于设置时钟事件设备模式，如果定义了 set_mode()函数则不需要这些函数。

●**rating**：时钟事件设备质量。

●**irq**：时钟硬件的中断编号。

●**bound_on**：当前绑定的 CPU 核编号。

●**cpumask**：时钟事件设备能绑定到哪些 CPU 核（CPU 亲和性）。

●**list**：双链表成员，将实例添加到全局双链表。

内核在/kernel/time/clockevents.c 文件内定义了全局双链表头用于管理注册的 clock_event_device 实例：

```
static LIST_HEAD(clockevent_devices);        /*当前使用的时钟事件设备*/
static LIST_HEAD(clockevents_released);      /*释放的时钟事件设备*/
```

2 注册时钟事件设备

时钟事件设备 clock_event_device 实例由体系结构（平台）代码或时钟硬件驱动程序定义并向内核注册。注册时钟事件设备时，内核会将其插入到全局双链表 clockevent_devices 中，并选择最好的时钟事件设备关联到当前 CPU 核的时钟设备。

注册时钟事件设备 clock_event_device 实例的函数定义如下（/kernel/time/clockevents.c）：

```
void clockevents_register_device(struct clock_event_device *dev)
{
    unsigned long flags;

    BUG_ON(clockevents_sanity_check(dev));    /*安全性检查*/
    clockevent_set_state(dev, CLOCK_EVT_STATE_DETACHED); /*设置时钟事件设备初始状态*/

    if (!dev->cpumask) {    /*如果未分配 cpumask 实例*/
        WARN_ON(num_possible_cpus() > 1);
        dev->cpumask = cpumask_of(smp_processor_id());
                                        /*创建只包含当前 CPU 核的 cpumask 实例*/
    }
    raw_spin_lock_irqsave(&clockevents_lock, flags);
```

```
        list_add(&dev->list, &clockevent_devices); /*将实例添加到全局 clockevent_devices 双链表头部*/
        tick_check_new_device(dev);      /*设置时钟设备（按需设置），/kernel/time/tick-common.c*/
        clockevents_notify_released();       /*/kernel/time/clockevents.c*/
                /*扫描 clockevents_released 双链表是否有更适合的时钟事件设备可关联到时钟设备*/
        raw_spin_unlock_irqrestore(&clockevents_lock, flags);
    }
```

默认情况下，由哪个 CPU 核调用 clockevents_register_device()函数注册的时钟事件设备，就只能由该 CPU 核使用（关联其时钟设备），除非 clock_event_device 实例中设置了其 CPU 核亲和性。

注册时钟事件设备函数首先设置时钟事件设备初始状态为 DETACHED，然后将实例添加到全局双链表 clockevent_devices 头部；随后调用 **tick_check_new_device(dev)**函数判断是否可以将新注册的时钟事件设备关联到当前 CPU 核的时钟设备，如果可以则关联到时钟设备，否则不关联；最后，扫描全局双链表 clockevents_released，将此双链表中实例移至 clockevent_devices 链表，并检查是否有更适合作为当前 CPU 核时钟设备关联时钟事件设备的，如果有则关联（调用 tick_check_new_device(dev)函数），没有则跳过。

下面将主要介绍 tick_check_new_device(dev)函数的实现。

■是否更换时钟事件设备

tick_check_new_device(dev)函数用于判断 dev 指向时钟事件设备是否可以设为当前 CPU 核时钟设备关联的时钟事件设备，如果可以则设置，否则不设置。

tick_check_new_device()函数定义在/kernel/time/tick-common.c 文件内：

```
void tick_check_new_device(struct clock_event_device *newdev)
{
    struct clock_event_device *curdev;
    struct tick_device *td;
    int cpu;

    cpu = smp_processor_id();       /*当前 CPU 核编号*/
    if (!cpumask_test_cpu(cpu, newdev->cpumask))    /*如果 cpumask 位图中不含当前 CPU，返回*/
        goto out_bc;

    td = &per_cpu(tick_cpu_device, cpu);      /*当前 CPU 核时钟设备实例指针*/
    curdev = td->evtdev;              /*当前关联的时钟事件设备*/

    if (!tick_check_percpu(curdev, newdev, cpu))   /*判断 newdev 及中断与当前 CPU 核的亲和性*/
        goto out_bc;

    if (!tick_check_preferred(curdev, newdev))   /*在 newdev 和中 curdev 选择最优的*/
        goto out_bc;            /*优先选择具有单触发特性的实例，然后选择 rating 值更大的实例*/

    if (!try_module_get(newdev->owner))       /*需要加载模块，则加载模块*/
        return;
```

```
    /*运行至此说明需要用 newdev 替换 curdev 作为当前 CPU 核关联的时钟事件设备*/
    if (tick_is_broadcast_device(curdev)) { /*!GENERIC_CLOCKEVENTS_BROADCAST，返回 0*/
        clockevents_shutdown(curdev);
        curdev = NULL;
    }
    clockevents_exchange_device(curdev, newdev);   /*curdev 移入 clockevents_released 链表*/
                                                   /*关闭 newdev，/kernel/time/clockevents.c*/
    tick_setup_device(td, newdev, cpu, cpumask_of(cpu));
                    /*设置 newdev 为当前 CPU 核关联时钟事件设备，/kernel/time/tick-common.c*/
    if (newdev->features & CLOCK_EVT_FEAT_ONESHOT)   /*如果时钟事件设备具有单触发特性*/
        tick_oneshot_notify();
                                /*设置 tick_sched->check_clocks 成员 bit0 位，/kernel/time/tick-sched.c*/
    return;

out_bc:
    tick_install_broadcast_device(newdev); /*需选择 GENERIC_CLOCKEVENTS_BROADCAST 选项*/
                        /*判断是否可将其用作广播设备，由平台选择，/kernel/time/tick-broadcast.c*/
}
```

tick_check_new_device()函数获取当前 CPU 核的原时钟设备，通过一系列判断决定是否需要用新注册的时钟事件设备代替当前 CPU 核使用的时钟事件设备。如果不需要则函数返回，如果需要则关闭新时钟事件设备，并将当前时钟事件设备移至 clockevents_released 双链表，调用 **tick_setup_device()**函数将新注册的时钟事件设备关联到当前 CPU 核的时钟设备。

●设置时钟设备

tick_setup_device()函数（/kernel/time/tick-common.c）用于将新时钟事件设备关联到指定 CPU 核的时钟设备，函数定义如下：

```
static void tick_setup_device(struct tick_device *td, struct clock_event_device *newdev, int cpu,  \
                                                        const struct cpumask *cpumask)
/*cpumask：只标记 cpu 核的位图*/
{
    ktime_t next_event;     /*产生下一时钟事件的时间间隔，纳秒*/
    void (*handler)(struct clock_event_device *) = NULL;     /*时钟事件处理函数*/

    if (!td->evtdev) {              /*如果当前时钟设备没有关联到时钟事件设备*/
        if (tick_do_timer_cpu == TICK_DO_TIMER_BOOT) {   /*如果尚未设置全局时钟*/
            if (!tick_nohz_full_cpu(cpu))           /*/include/linux/tick.h*/
                tick_do_timer_cpu = cpu;            /*CPU 核局部时钟设为全局时钟*/
            else                    /*不能设为全局时钟*/
                tick_do_timer_cpu = TICK_DO_TIMER_NONE;   /*-1*/
            tick_next_period = ktime_get();       /*当前单调递增时间*/
            tick_period = ktime_set(0, NSEC_PER_SEC / HZ); /*周期时钟事件间隔，节拍纳秒数*/
```

```
        }
        td->mode = TICKDEV_MODE_PERIODIC;     /*时钟设备初始模式设为周期模式*/
    } else {                /*时钟设备当前关联的时钟事件设备不为空*/
        handler = td->evtdev->event_handler;          /*原时钟事件设备处理函数*/
        next_event = td->evtdev->next_event;          /下一时钟事件时间*/
        td->evtdev->event_handler = clockevents_handle_noop;   /*原时钟事件设备处理函数设为空*/
    }

    td->evtdev = newdev;      /*时钟设备关联新时钟事件设备*/

    if (!cpumask_equal(newdev->cpumask, cpumask))       /*将中断关联到当前 CPU 核*/
        irq_set_affinity(newdev->irq, cpumask); /*设置中断与 CPU 亲和性，中断由此 CPU 核响应*/

    if (tick_device_uses_broadcast(newdev, cpu))
                                    /*!GENERIC_CLOCKEVENTS_BROADCAST，返回 0*/
        return;

    /*设置并激活时钟设备*/
    if (td->mode == TICKDEV_MODE_PERIODIC)       /*周期工作模式*/
        tick_setup_periodic(newdev, 0); /*设置时钟事件设备周期模式，/kernel/time/tick-common.c*/
    else                                /*时钟设备原为单触发工作模式*/
        tick_setup_oneshot(newdev, handler, next_event);
                                    /*设置时钟事件设备单触发模式，/kernel/time/tick-oneshot.c*/
}
```

tick_setup_device()函数首先判断 CPU 核当前关联时钟事件设备是否为空，如果是则还需要判断是否要将本时钟设备设为全局时钟，并将时钟事件设备设为周期工作模式，时钟周期为 1/HZ（转为纳秒），将新时钟事件设备关联到时钟设备。

如果 CPU 核原关联时钟事件设备不为空，则将获取原时钟事件设备处理函数和下一个时钟事件时间等信息，然后设置新时钟事件设备关联时钟设备，并保留原时钟设备工作模式不变。

最后根据时钟设备当前的工作模式设置关联的时钟事件设备。

（1）设置周期工作模式

如果时钟设备工作模式为周期模式，则调用 **tick_setup_periodic(newdev, 0)**函数对时钟事件设备进行设置，函数定义如下（/kernel/time/tick-common.c）：

```
void tick_setup_periodic(struct clock_event_device *dev, int broadcast)
/*dev：时钟事件设备指针，broadcast：是否设为广播设备，此处为 0*/
{
    tick_set_periodic_handler(dev, broadcast);   /*设置时钟事件处理函数为tick_handle_periodic()*/
    if (!tick_device_is_functional(dev))     /*x86，/kernel/time/tick-internal.h*/
        return;

    /*设置时钟事件设备工作状态*/
```

```
    if ((dev->features & CLOCK_EVT_FEAT_PERIODIC) && !tick_broadcast_oneshot_active()) {
        clockevents_switch_state(dev, CLOCK_EVT_STATE_PERIODIC);
                                                    /*设置时钟事件设备为周期工作状态*/
    } else {            /*如果时钟事件设备不支持周期工作状态，以单触发模式模拟周期工作模式*/
        unsigned long seq;
        ktime_t next;

        do {
            seq = read_seqbegin(&jiffies_lock);
            next = tick_next_period;          /*下次时钟事件时间*/
        } while (read_seqretry(&jiffies_lock, seq));

        clockevents_switch_state(dev, CLOCK_EVT_STATE_ONESHOT);   /*设为单触发工作状态*/

        for (;;) {                                  /*编程设置下一次时钟事件*/
            if (!clockevents_program_event(dev, next, false)) /*设置失败则增加时间间隔后再设置*/
                return;
            next = ktime_add(next, tick_period);     /*增加一个时钟周期再设置时钟设备*/
        }
    }
}
```

tick_setup_periodic()函数首先调用 tick_set_periodic_handler()函数设置时钟事件设备的处理函数，然后设置时钟事件设备的工作状态。

tick_set_periodic_handler()函数定义在/kernel/time/tick-broadcast.c 文件内：

```
void tick_set_periodic_handler(struct clock_event_device *dev, int broadcast)
{
    if (!broadcast)      /*非广播设备*/
        dev->event_handler = tick_handle_periodic; /*时钟事件设备处理函数，在硬件中断中调用*/
    else                 /*广播设备*/
        dev->event_handler = tick_handle_periodic_broadcast;   /*广播设备处理函数*/
}
```

周期工作模式下时钟事件设备处理函数为 **tick_handle_periodic()**，该函数在时钟硬件中断处理函数中调用，后面将会介绍此函数的实现。

tick_setup_periodic()函数随后需要根据时钟设备工作模式设置时钟事件设备的工作状态，如果时钟事件设备支持周期工作模式，则直接将其切换至周期工作状态即可。如果时钟事件设备不支持周期工作模式（硬件不支持），则将其切换至单触发工作状态，并以 tick_period（1/HZ）的间隔对其进行编程，以设置下一个时钟事件。在事件处理函数 tick_handle_periodic()中，在执行完正常的工作后，如果时钟事件设备工作在单触发状态，最后将以 tick_period 间隔的到期时间对时钟事件设备编程，以便触发下一个事件，以模拟周期工作模式。

（2）设置单触发模式

如果时钟设备工作在单触发模式下，则调用 **tick_setup_oneshot(newdev, handler, next_event)**函数设置时钟事件设备，函数定义在/kernel/time/tick-oneshot.c 文件内：

```
void tick_setup_oneshot(struct clock_event_device *newdev,void (*handler)(struct clock_event_device *), \
                                                                    ktime_t next_event)
/*handler：原时钟事件设备处理函数，next_event：原来时钟事件设备下一个事件时间*/
{
    newdev->event_handler = handler;        /*时钟事件设备处理函数保持不变*/
    clockevents_switch_state(newdev, CLOCK_EVT_STATE_ONESHOT);    /*设为单触发工作状态*/
    clockevents_program_event(newdev, next_event, true);   /*编程设置下一次时钟事件*/
}
```

单触发模式下，时钟事件设备的设置函数比较简单，其处理函数与原处理函数相同，保持不变，然后设置时钟事件设备为单触发工作状态，最后编程设置时钟事件设备，以便产生下一个时钟事件。

时钟设备最初都设为周期工作模式，在周期时钟事件处理函数 tick_handle_periodic()中将会根据需要改变时钟设备的工作模式，并重新设置时钟事件设备处理函数，详见后面小节。

（3）时钟事件设备编程

在前面的介绍中，如果时钟事件设备工作在单触发模式下，在处理函数中需要对其重新编程，以便触发下一个事件（中断），否则将不再产生时钟事件。下面看一下对时钟事件设备编程的接口函数。

clockevents_program_event()函数用于对指定时钟事件设备进行编程，编程成功返回 0，函数定义如下：

```
int clockevents_program_event(struct clock_event_device *dev, ktime_t expires,bool force)
/*/kernel/time/clockevents.c
*expires：下一个事件产生时间，内核的单调递增时间，纳秒值。
*force：是否强制产生时钟事件，非 0 表示如果到期时间已过，则以最小时间间隔产生时钟事件。
*/
{
    unsigned long long clc;
    int64_t delta;
    int rc;

    if (unlikely(expires.tv64 < 0)) {
        ...
    }

    dev->next_event = expires;          /*下一事件时间*/

    if (clockevent_state_shutdown(dev))     /*时钟事件设备是否已关闭*/
        return 0;
    WARN_ONCE(!clockevent_state_oneshot(dev), "Current state: %d\n",
                                            clockevent_get_state(dev));      /*必须处于单触发状态*/

    if (dev->features & CLOCK_EVT_FEAT_KTIME)   /*时钟事件设备支持以纳秒单位表示的时间*/
```

```
            return dev->set_next_ktime(expires, dev);      /*编程设置产生下一事件*/

        /*时钟事件设备不支持以纳秒表示的时间，采用以时钟源计数值表示时间间隔*/
        delta = ktime_to_ns(ktime_sub(expires, ktime_get()));   /*从现在至下次事件的时间间隔，纳秒值*/
        if (delta <= 0)                 /*以最小时间间隔产生时钟事件，force!=0*/
            return force ? clockevents_program_min_delta(dev) : -ETIME;

        /*确定间隔时间，需在 max_delta_ns 与 min_delta_ns 之间*/
        delta = min(delta, (int64_t) dev->max_delta_ns);
        delta = max(delta, (int64_t) dev->min_delta_ns);

        clc = ((unsigned long long) delta * dev->mult) >> dev->shift;   /*间隔时间纳秒值转为周期计数值*/
        rc = dev->set_next_event((unsigned long) clc, dev);      /*编程设置产生下一事件*/

        return (rc && force) ? clockevents_program_min_delta(dev) : rc;      /*成功返回 0*/
    }
```

对当前 CPU 核关联时钟事件设备编程的函数为 tick_program_event()，定义如下：

```
int tick_program_event(ktime_t expires, int force)      /*/kernel/time/tick-oneshot.c*/
{
    struct clock_event_device *dev = __this_cpu_read(tick_cpu_device.evtdev);   /*时钟事件设备*/

    if (unlikely(expires.tv64 == KTIME_MAX)) {      /*到期时间为最大值，不需要再产生时钟事件*/
        clockevents_switch_state(dev, CLOCK_EVT_STATE_ONESHOT_STOPPED);
        return 0;                           /*设为停止工作状态*/
    }

    if (unlikely(clockevent_state_oneshot_stopped(dev))) {      /*时钟事件设备处于停止单触发状态*/
        clockevents_switch_state(dev, CLOCK_EVT_STATE_ONESHOT);   /*切换至单触发状态*/
    }
    return clockevents_program_event(dev, expires, force);
                                                /*编程时钟事件设备，/kernel/time/clockevents.c*/
}
```

由上可知，只有时钟事件设备处于单触发工作状态，才可以对其进行编程设置。

3 时钟硬件初始化

通常一个时钟硬件即可作为时钟源，也可作为时钟事件设备。体系结构（平台）相关代码或时钟硬件驱动程序中需要定义时钟源 clocksource 实例和时钟事件设备 clock_event_device 实例，并向内核注册这两个实例。另外，还需要向内核注册时钟硬件中断，在中断处理函数中调用时钟事件设备的处理函数。

通常体系结构相关代码会将 CPU 核中的时钟计数寄存器注册为时钟源和时钟事件设备。

内核启动函数 start_kernel()中调用 time_init()函数，完成时钟硬件初始化，以及时钟源、时钟事件设

备的注册等工作，代码如下（/arch/mips/kernel/time.c）：

```
void __init time_init(void)
{
    plat_time_init();   /*平台实现函数，/arch/mips/loongson32/common/time.c*/

    /*体系结构注册时钟源和时钟事件设备*/
    if (mips_clockevent_init() != 0 || !cpu_has_mfc0_count_bug())      /*注册时钟事件设备*/
        init_mips_clocksource();        /*注册时钟源，/arch/mips/include/asm/time.h*/
}
```

time_init()函数首先调用平台代码实现的 plat_time_init()函数，完成平台时钟硬件的初始化和相应时钟源、时钟事件设备的注册。随后调用体系结构代码定义的 mips_clockevent_init()函数注册时钟事件设备，注册成功返回 0，然后再调用 init_mips_clocksource()函数注册时钟源。

■平台初始化

龙芯 1B 处理器实现了四路脉冲宽度调节（PWM）/计数控制器，可工作于定时器模式。PWM 控制器中包含一个计数器 CNTR（有效位 24 位），在外部硬件时钟的驱动下不断自加，当计数值等于 HRC 或 LRC 寄存器值时将产生中断，PWM 控制器可注册为系统时钟源和时钟事件设备，更详细的硬件信息请参考处理器手册。

loongson32 平台，时钟硬件初始化函数 plat_time_init()定义如下，假设配置选择了 CEVT_CSRC_LS1X 选项，即采用 PWM 控制器注册时钟源和时钟事件设备。

```
void __init plat_time_init(void)      /*/arch/mips/loongson32/common/time.c*/
{
    struct clk *clk = NULL;

    ls1x_clk_init();    /*初始化 CPU 外设时钟频率，/drivers/clk/clk-ls1x.c*/

#ifdef CONFIG_CEVT_CSRC_LS1X    /*选择 PWM 作为时钟源、时钟事件设备*/
    clk = clk_get(NULL, "ls1x_pwmtimer");
    if (IS_ERR(clk))
        panic("unable to get timer clock, err=%ld", PTR_ERR(clk));

    mips_hpt_frequency = clk_get_rate(clk);    /*硬件时钟频率*/
    ls1x_time_init();      /*注册 PWM 控制器表示的时钟源、时钟事件设备*/
#else      /*没有选择 PWM 作为时钟硬件*/
    clk = clk_get(NULL, "cpu_clk");
    if (IS_ERR(clk))
        panic("unable to get cpu clock, err=%ld", PTR_ERR(clk));

    mips_hpt_frequency = clk_get_rate(clk) / 2;
  #endif      /* CONFIG_CEVT_CSRC_LS1X */
}
```

初始化函数在设置系统时钟频率后，调用 **ls1x_time_init()**函数初始化 PWM 控制器，注册 PWM 控制器表示的时钟源和时钟事件设备，函数定义如下：

```
static void __init ls1x_time_init(void)
{
    struct clock_event_device *cd = &ls1x_clockevent;   /*时钟事件设备实例*/
    int ret;

    if (!mips_hpt_frequency)
        panic("Invalid timer clock rate");

    ls1x_pwmtimer_init();    /*初始化 PWM 控制寄存器，请读者参考龙芯 1B 处理器手册*/

    clockevent_set_clock(cd, mips_hpt_frequency);
                        /*设置时钟事件设备 mult、shift 成员，/arch/mips/include/asm/time.h*/
    cd->max_delta_ns = clockevent_delta2ns(0xffffff, cd);   /*时钟事件最大间隔，纳秒*/
    cd->min_delta_ns = clockevent_delta2ns(0x000300, cd);    /*时钟事件最小间隔，纳秒*/
    cd->cpumask = cpumask_of(smp_processor_id());    /*关联处理器核*/
    clockevents_register_device(cd);          /*注册时钟事件设备*/

    ls1x_clocksource.rating = 200 + mips_hpt_frequency / 10000000;    /*时钟源等级*/
    ret = clocksource_register_hz(&ls1x_clocksource, mips_hpt_frequency);      /*注册时钟源实例*/
    if (ret)
        panic(KERN_ERR "Failed to register clocksource: %d\n", ret);

    setup_irq(LS1X_TIMER_IRQ, &ls1x_pwmtimer_irqaction);       /*注册时钟事件设备中断*/
}
```

PWM 控制器对应的时钟源和时钟事件设备实例定义如下：

```
static struct clocksource ls1x_clocksource = {        /*时钟源实例*/
    .name           = "ls1x-pwmtimer",
    .read       = ls1x_clocksource_read,          /*读计数器值函数*/
    .mask           = CLOCKSOURCE_MASK(24),    /*计数器为 24 位*/
    .flags          = CLOCK_SOURCE_IS_CONTINUOUS,    /*连续时钟源*/
};

static struct clock_event_device ls1x_clockevent = {      /*时钟事件设备实例*/
    .name       = "ls1x-pwmtimer",
    .features   = CLOCK_EVT_FEAT_PERIODIC,      /*周期时钟*/
    .rating     = 300,
    .irq        = LS1X_TIMER_IRQ,     /*中断编号*/
    .set_next_event     = ls1x_clockevent_set_next,   /*设置下一时钟事件（中断），按计数值设置*/
```

```
    .set_mode       = ls1x_clockevent_set_mode,        /*设置时钟事件设备工作模式*/
};
```

PWM 控制器作为时钟硬件时 LS1X_TIMER_IRQ 宏根据配置选择的 PWM 控制器，定义成对应 PWM 控制器的中断编号，例如：选择 PWM1 控制器则中断号为 LS1X_PWM1_IRQ。中断响应结构实例如下：

```
static struct irqaction ls1x_pwmtimer_irqaction = {
    .name           = "ls1x-pwmtimer",
    .handler   = ls1x_clockevent_isr,     /*中断处理函数*/
    .dev_id         = &ls1x_clockevent,     /*指向时钟事件设备*/
    .flags          = IRQF_PERCPU | IRQF_TIMER,
};
```

在中断处理函数 **ls1x_clockevent_isr()**中将调用时钟事件设备中的处理函数，函数定义如下：

```
static irqreturn_t ls1x_clockevent_isr(int irq, void *devid)
{
    struct clock_event_device *cd = devid;     /*时钟事件设备*/

    ls1x_pwmtimer_restart();
    cd->event_handler(cd);          /*时钟事件处理函数*/

    return IRQ_HANDLED;   /*中断已经处理*/
}
```

时钟源计数器值的读取函数，时钟事件设备工作模式的设置函数等，主要是对 PWM 控制寄存器的操作，请读者自行阅读源代码。

■体系结构初始化

在 time_init()函数中除了调用 plat_time_init()函数注册平台（处理器）定义的时钟源、时钟事件设备外，还将调用 **mips_clockevent_init()**和 **init_mips_clocksource()**函数注册体系结构定义的时钟事件设备和时钟源，此处通常利用处理器中 CP0 协处器中的 count 和 compare 寄存器实现时钟源和时钟事件设备。count 寄存器在处理器硬件时钟的驱动下计数，当计数值达到 compare 寄存器值时将向 CPU 核发送硬件中断。

●注册时钟事件设备

mips_clockevent_init()函数用于注册时钟事件设备，定义如下（/arch/mips/include/asm/time.h）：

```
static inline int mips_clockevent_init(void)        /*每个 CPU 核启动时都会调用此函数*/
{
  #ifdef CONFIG_CEVT_R4K       /*龙芯 1B 选择该配置选项*/
    return    r4k_clockevent_init();     /*函数返回 0，/arch/mips/kernel/csrc-r4k.c*/
  #else
    return -ENXIO;
  #endif
}
```

内核在/arch/mips/kernel/cevt-r4k.c 文件内为每个 CPU 核定义了时钟事件设备实例以及中断响应实例：

DEFINE_PER_CPU(struct clock_event_device, **mips_clockevent_device**);

r4k_clockevent_init()函数内初始化当前 CPU 核对应的 clock_event_device 实例（只支持单触发状态），向内核注册此实例，并注册相应中断（中断号为 MIPS_CPU_IRQ_BASE + 7），中断响应结构体实例定义如下（c0_compare_irqaction）。

```
struct irqaction c0_compare_irqaction = {       /*中断响应结构*/
    .handler = c0_compare_interrupt,        /*调用时钟事件设备中的处理函数*/
    .flags = IRQF_PERCPU | IRQF_TIMER | IRQF_SHARED,
    .name = "timer",
};
```

c0_compare_irqaction 实例中中断处理函数为 **c0_compare_interrupt()**，函数内调用 CPU 核对应的时钟事件设备 clock_event_device 中的处理函数 **event_handler(cd)**，处理时钟事件。

●注册时钟源

init_mips_clocksource()用于注册体系结构定义的时钟源，代码如下：

```
static inline int init_mips_clocksource(void)
{
#ifdef CONFIG_CSRC_R4K
    return init_r4k_clocksource();    /*注册时钟源*/
#else
    return 0;
#endif
}
```

内核在/arch/mips/kernel/csrc-r4k.c 文件内定义了由 **count** 寄存器实现的时钟源实例：

```
static struct clocksource clocksource_mips = {
    .name           = "MIPS",
    .read       = c0_hpt_read,      /*直接读取当前 CPU 核 count 寄存器值*/
    .mask           = CLOCKSOURCE_MASK(32),     /*32 位*/
    .flags          = CLOCK_SOURCE_IS_CONTINUOUS,
};
```

init_r4k_clocksource()函数用于注册 clocksource_mips 时钟源实例，代码如下：

```
int __init init_r4k_clocksource(void)
{
    if (!cpu_has_counter || !mips_hpt_frequency)
        return -ENXIO;

    clocksource_mips.rating = 200 + mips_hpt_frequency / 10000000;
    clocksource_register_hz(&clocksource_mips, mips_hpt_frequency);    /*注册时钟源*/
    sched_clock_register(r4k_read_sched_clock, 32, mips_hpt_frequency);    /*注册调度时钟源*/
    return 0;
```

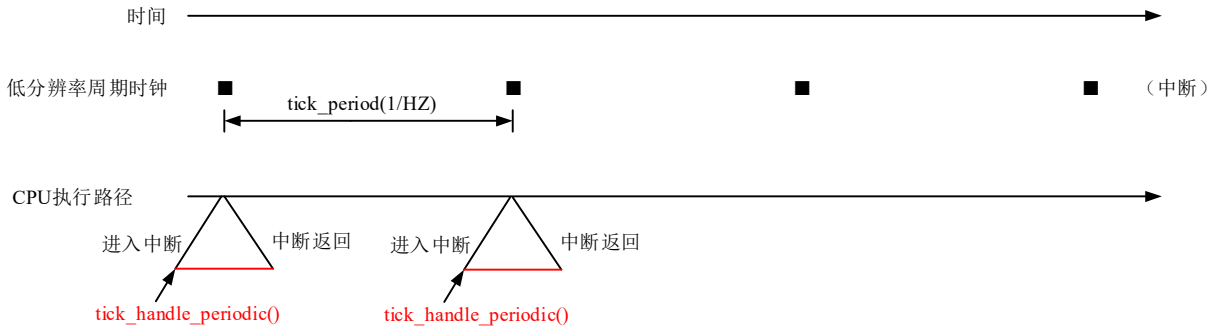
}

这里作者有个问题没有搞明白，在多核处理器中，计时器时间值（计数值）由全局时钟更新，就是由充当全局时钟的 CPU 核更新。但是在获取时间值时，各 CPU 核读取的是自己 count 寄存器值以获取当前计数器值，而不是充当全局时钟那个 CPU 核的 count 寄存器值，这势必造成读取时间的不准确。

另外，内核在/drivers/clocksource/目录下包含了时钟硬件驱动程序，用户也可在此定义和注册由其它时钟硬件实现的时钟源和时钟事件设备。

6.8.4 低分辨率周期时钟

CPU 核时钟设备初始工作模式为周期模式，周期性地产生时钟事件，时钟硬件将以 HZ 的频率产生硬件中断，称为内核节拍（滴答，tick）。时钟事件设备的事件处理函数为 **tick_handle_periodic()**，在中断处理函数中调用此事件处理函数，如下图所示。



tick_handle_periodic()函数内完成内核指定的周期性、例行性的工作，如进程调度、处理定时器等。如果 CPU 核的时钟设备充当了全局时钟还需要更新 jiffies（jiffies64）值和计时器时间值等。

这里需要注意的是 tick_handle_periodic()函数是在中断处理程序中被调用的。

1 周期时钟处理函数

tick_handle_periodic()函数定义在/kernel/time/tick-common.c 文件内：

```
void tick_handle_periodic(struct clock_event_device *dev)
/*dev：时钟事件设备实例指针*/
{
    int cpu = smp_processor_id();   /*当前 CPU 核编号*/
    ktime_t next = dev->next_event;     /*下一个时钟事件（中断）时间*/

    tick_periodic(cpu);     /*完成节拍工作，/kernel/time/tick-common.c*/

  #if defined(CONFIG_HIGH_RES_TIMERS) || defined(CONFIG_NO_HZ_COMMON)
    if (dev->event_handler != tick_handle_periodic) /*如果已启用了高分辨率时钟或动态时钟，返回*/
        return;
  #endif

    if (!clockevent_state_oneshot(dev))      /*时钟事件设备不是单触状态（是周期状态），函数返回*/
```

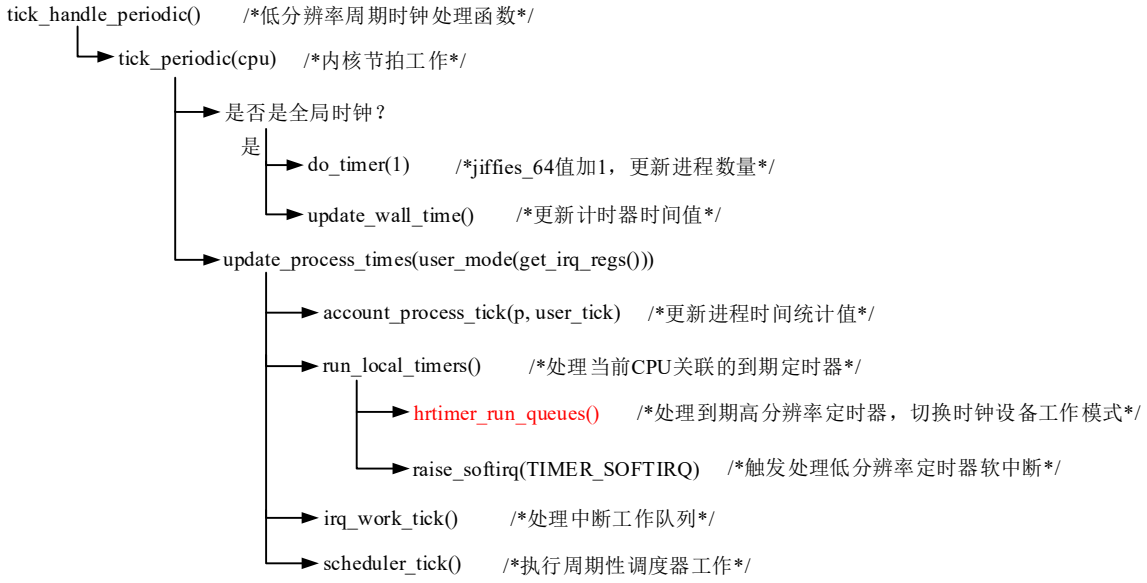
```
            return;

        /*时钟事件设备工作于单触发状态，需要编程设置下一事件*/
        for (;;) {
            next = ktime_add(next, tick_period);     /*下次事件时间加上节拍间隔*/
            if (!clockevents_program_event(dev, next, false))   /*编程设置时钟事件设备*/
                return;
            /*设置不成功表示下一事件已过期，需要再加一个周期*/
            if (timekeeping_valid_for_hres())     /*时钟源支持高分辨率模式，/kernel/time/timekeeping.c*/
                tick_periodic(cpu);                  /*处理已经过期的周期节拍工作*/
        }
    }
```

tick_handle_periodic()函数中调用 **tick_periodic(cpu)**函数完成内核节拍工作（周期性工作），随后判断是否启用了动态时钟或高分辨率时钟，若是则返回，否则往下执行。

随后，判断时钟事件设备是否处于单触发状态，如果不是（即周期状态）则函数返回，否则还需要对时钟事件设备编程，设置下一时钟事件产生的时间，如果下次事件已经过期则增加时间间隔（tick_period）后再设置，直至设置成功。

2 节拍工作

tick_periodic()函数完成内核周期性的例行工作，函数调用关系如下图所示：



tick_periodic()函数判断当前 CPU 核时钟是否是内核的全局时钟，如果是则需要完成 jiffies_64 全局变量的加 1 操作（内核节拍计数加 1）、更新内核进程数量、更新计时器时间值等工作。然后，调用函数 update_process_times()完成本 CPU 核的周期性工作，如处理到期定时器、执行周期性进程调度等。

tick_periodic()函数定义在/kernel/time/tick-common.c 文件内，代码如下：

```
static void tick_periodic(int cpu)
{
    if (tick_do_timer_cpu == cpu) {      /*当前 CPU 时钟是内核全局时钟*/
        write_seqlock(&jiffies_lock);
```

```
            tick_next_period = ktime_add(tick_next_period, tick_period);   /*下一个节拍产生的时间*/

            do_timer(1);        /*jiffies_64 计数加 1，更新进程数量等，/kernel/time/timekeeping.c*/
            write_sequnlock(&jiffies_lock);
            update_wall_time();   /*更计时器时间值，见本节前文，/kernel/time/timekeeping.c*/
        }

        update_process_times(user_mode(get_irq_regs()));   /*处理本 CPU 核工作，/kernel/time/timer.c*/
        profile_tick(CPU_PROFILING);       /*/kernel/profile.c*/
    }
```

■处理全局工作

全局时钟需要调用 do_timer()函数执行更新 jiffies_64 值的工作等，然后还要调用 update_wall_time()函数更新计时器时间值（前文介绍过了），下面主要看一下 do_timer()函数的实现。

do_timer()函数定义如下（/kernel/time/timekeeping.c）：

```
void do_timer(unsigned long ticks)
{
    jiffies_64 += ticks;   /*jiffies_64 值加 1*/
    calc_global_load(ticks);   /*更新内核进程数量等，见第 5 章*/
}
```

■处理本 CPU 核工作

update_process_times()函数完成当前 CPU 核的周期性工作，参数 user_mode(get_irq_regs())表示时钟中断发生前 CPU 是处于用户态还是内核态，内核态时此值为 0（假），用户态时为非 0（真），时钟硬件中断可能发生在用户态，也可以发生在内核态。

```
void update_process_times(int user_tick)
{
    struct task_struct *p = current;
    account_process_tick(p, user_tick);   /*更新进程时间统计值，/kernel/sched/cputime.c*/
    run_local_timers();            /*处理到期定时器，切换时钟设备工作模式等，/kernel/time/timer.c*/
    rcu_check_callbacks(user_tick);      /*RCU 机制，见下一节*/
#ifdef CONFIG_IRQ_WORK
    if (in_irq())
        irq_work_tick();   /*处理中断工作队列，见本章前文，/kernel/irq_work.c*/
#endif
    scheduler_tick();   /*周期性调度器执行函数，见第 5 章，/kernel/sched/core.c*/
    run_posix_cpu_timers(p);
}
```

account_process_tick()函数用于更新进程的时间统计值，如在用户态运行的时长、在内核态运行的时

长等。除了 run_local_timers()函数外，其它函数前面都介绍过了。

run_local_timers()函数定义在/kernel/time/timer.c 文件内：

```
void run_local_timers(void)
{
    hrtimer_run_queues();          /*切换时钟设备工作模式以及处理高分辨率定时器等*/
    raise_softirq(TIMER_SOFTIRQ);  /*触发处理低分辨率定时器的软中断*/
}
```

run_local_timers()函数内调用 hrtimer_run_queues()函数处理到期高分辨率定时器以及判断是否需要切换时钟设备工作模式，需要则切换，不需要则不切换（保持周期时钟），后面小节将详细介绍此函数的实现。raise_softirq(TIMER_SOFTIRQ)函数用于触发处理低分辨率定时器的软中断，见下文。

3 低分辨率定时器

前面介绍的延时工作实现，如软中断、工作队列等，并没有指定工作延迟执行的具体时间。内核定义了定时器，用于将某一工作延期指定的时间后执行。内核中有两种类型的定时器，一种是低分辨率定时器，另一种是高分辨率定时器。低分辨率定时器以内核节拍数为单位计时，计时精度为 1/HZ(s)。在内核节拍工作中将处理在当前节拍到期或过期的低分辨率定时器。

低分辨率定时器不能满足一些对时间精度要求较高的场合，为此内核实现了高分辨定时器，它以纳秒的时间值表示到期时间（单调递增时间）。高分辨定时器意味着需要更高精度的时钟硬件中断，在中断处理函数中处理到期的高分辨率定时器。若启用了高分辨率定时器，将用定时器的到期时间去编程时钟事件设备，以便能及时地产生时钟事件处理高分辨率定时器。

简单地说，低分辨率定时器始终是在内核节拍中处理，由内核掌握节奏，低分辨率定时器是被动地被处理的。而高分辨率定时器（如果启用了高分辨率时钟），它会主动去设置时钟设备，迫使其在定时器到期时产生时钟事件，在事件处理函数中处理高分辨率定时器。

本小节先介绍低分辨率定时器的实现，后面小节将介绍高分辨定时器的实现。

■添加定时器

低分辨率定时器由 timer_list 结构体表示，结构体定义如下（/include/linux/timer.h）：

```
struct timer_list {
    struct hlist_node    entry;         /*散列表节点成员，将实例添加到定时器管理结构中*/
    unsigned long    expires;       /*到期时间，以节拍数为单位*/
    void        (*function)(unsigned long);   /*定时器执行函数，定时器需要执行的工作*/
    unsigned long        data;      /*传递给执行函数的参数*/
    u32         flags;                /*标记成员*/
    int         slack;
  #ifdef CONFIG_TIMER_STATS
    int         start_pid;
    void        *start_site;
    char        start_comm[16];
  #endif
  #ifdef CONFIG_LOCKDEP
    ...
```

```
    #endif
};
```

timer_list 结构体主要成员简介如下：

●**entry**：散列表节点成员，用于将低分辨率定时器实例加入散列表管理结构。

●**expires**：定时器到期时间，保存的是定时器到期时的内核节拍数。

●**function**、**data**：定时器执行函数指针及参数，定时器到期时将调用此函数。

●**flags**：标记成员，标记位定义如下（/include/linux/timer.h）：

```
#define   TIMER_CPUMASK          0x0007FFFF      /*CPU 核掩码，表示可执行定时器的 CPU 核*/
#define   TIMER_MIGRATING        0x00080000      /*定时器正在迁移*/
#define   TIMER_BASEMASK         (TIMER_CPUMASK | TIMER_MIGRATING)
#define   TIMER_DEFERRABLE       0x00100000
#define   TIMER_IRQSAFE          0x00200000    /**/
```

定义并初始化定时器实例的宏如下（/include/linux/timer.h）：

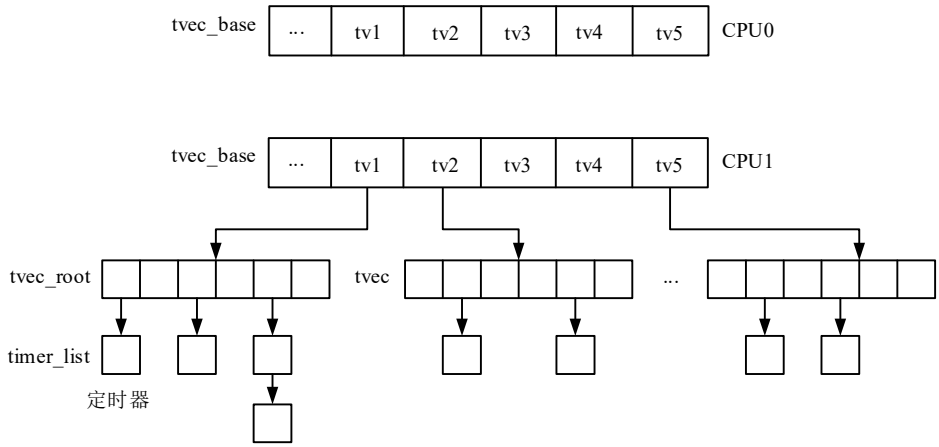
```
#define   DEFINE_TIMER(_name, _function, _expires, _data)      \
      struct timer_list    _name =TIMER_INITIALIZER(_function, _expires, _data)

#define TIMER_INITIALIZER(_function, _expires, _data)       \
      __TIMER_INITIALIZER((_function), (_expires), (_data), 0)

#define __TIMER_INITIALIZER(_function, _expires, _data, _flags) { \
            .entry = { .next = TIMER_ENTRY_STATIC },   \
            .function = (_function),              \
            .expires = (_expires),                     \
            .data = (_data),                      \
            .flags = (_flags),                    \
            .slack = -1,                          \
            __TIMER_LOCKDEP_MAP_INITIALIZER(          \
                  __FILE__ ":" __stringify(__LINE__))  \
      }
```

向内核添加定时器的函数为 **add_timer(struct timer_list *timer)**，函数定义如下（/kernel/time/timer.c）：

```
void add_timer(struct timer_list *timer)
{
      BUG_ON(timer_pending(timer));     /*定时器是否被挂起，/include/linux/timer.h*/
      mod_timer(timer, timer->expires);     /*将定时器插入到管理结构中，/kernel/time/timer.c*/
}
```

mod_timer()函数用于修改定时器到期时间并将其添加到当前 CPU 核的低分辨率定时器管理结构中。

内核定义了如下数据结构用于管理低分辨率定时器：



每个 CPU 核对应一个 tvec_base 结构体实例，其中包含了五个数组 tv1 至 tv5。定时器按到期时间的远近分配到各个数组中，例如：tv1 管理的是在 0-255 个内核节拍内到期的定时器，tv1 数组中共包含 256 项，每一项管理在某一节拍到期的定时器，tv2-tv5 数组项与此类似按定时器到期时间进行分组，只不过后面的分组到期时间粒度比较粗。内核节拍工作中处理低分辨率定时器时，只需要执行 tv1 数组中第一项管理的定时器，然后将后面的定时器往前移即可。

tv1 至 tv5 为内嵌在 tvec_base 结构中的五组散列表实例，散列表链接的对象为定时器结构 timer_list 实例。定时器实例在链表中以到期时间从左至右依次排序，例如 tv1 列表中共有 256 项，每个表项分别链接还有 0-255 个时钟周期到期的定时器，后面的散列表类似，只不过划分的粒度越越粗。周期时钟中断内只需要处理 tv1 表项中第一项链接的定时器，并将后续定时器往前移即可。

内核为系统内每个 CPU 核定义了各自的 tvec_base 结构体实例，用于管理各自关联的定时器：

static DEFINE_PER_CPU(struct tvec_base, **tvec_bases**);　　/*/kernel/time/timer.c*/

下面回过头来看一下向管理结构添加定时器实例的 mod_timer()函数：

```
int mod_timer(struct timer_list *timer, unsigned long expires)
/*expires：到期时间相对于当前 jiffies64 值的偏移量*/
{
    expires = apply_slack(timer, expires);    /*定时器需要添加到 tvec_base 实例哪个数组项*/

    if (timer_pending(timer) && timer->expires == expires)
        return 1;

    return __mod_timer(timer, expires, false, TIMER_NOT_PINNED); /*将定时器添加到管理结构*/
}
```

mod_timer()根据定时器到期时间计算出定时器需要添加到 tvec_base 实例（当前 CPU 核关联的实例）哪个数组项，__mod_timer()函数将定时器添加到管理结构中。

删除定时器的函数为 del_timer(struct timer_list * timer)，源代码请读者自行阅读。

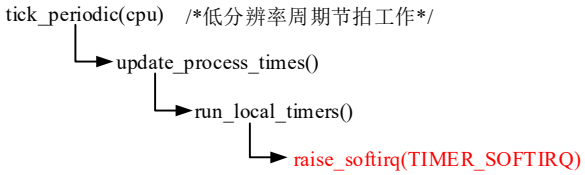
■处理到期定时器

内核通过 TIMER_SOFTIRQ 软中断来处理到期的低分辨率定时器。内核在启动阶段调用 **init_timers()** 函数，初始化低分辨定时器管理结构和开启低分辨率定时器处理软中断，函数定义在/kernel/time/timer.c

文件内：

```
void __init init_timers(void)
{
    init_timer_cpus();        /*初始化各 CPU 核 tvec_bases 结构体实例，/kernel/time/timer.c*/
    init_timer_stats();       /*更新统计量，需配置 TIMER_STATS，/kernel/time/timer_stats.c*/
    timer_register_cpu_notifier();  /*没有配置 HOTPLUG_CPU 为空操作，/kernel/time/timer.c*/
    open_softirq(TIMER_SOFTIRQ, run_timer_softirq);  /*开启定时器处理软中断*/
}
```

内核在周期节拍工作中触发 TIMER_SOFTIRQ 软中断来处理到期的低分辨率定时器，函数调用关系如下图所示：



TIMER_SOFTIRQ 软中断的处理函数为 **run_timer_softirq()**，函数定义在/kernel/time/timer.c 文件内，代码如下：

```
static void run_timer_softirq(struct softirq_action *h)
{
    struct tvec_base *base = this_cpu_ptr(&tvec_bases);    /*当前 CPU 核对应的 tvec_bases 实例*/

    if (time_after_eq(jiffies, base->timer_jiffies))
        __run_timers(base);      /*处理到期（或过期）的低分辨率定时器，/kernel/time/timer.c*/
}
```

软中断处理函数内调用__run_timers(base)函数处理当前 CPU 核所有到期（过期）的低分辨率定时器，对每个到期定时器调用其执行函数，并对 tvec_bases 实例中的定时器在管理结构中执行相应的移动操作。

6.8.5 低分辨率动态时钟

时钟设备工作于低分辨率周期时钟模式时，时钟硬件总是以 1/HZ 为周期产生时钟中断，触发时钟事件。对于一些功耗要求敏感的设备，如电池供电的手持设备，如果在 CPU 核无事可做时仍然周期性地产生时钟中断，这将浪费系统电力。在 CPU 核无事可做时，具体就是运行 idle 进程时，可暂停周期时钟（不产生时钟中断），在离开 idle 进程时再恢复周期时钟，以节省电力。可动态关闭和打开的周期时钟称之为（低分辨率）动态时钟。有时也称为无时钟系统，但是这个称呼不太准确，动态时钟依然有周期时钟，只不过有时会暂停。

内核提供了以下几个配置选项（单选项），以选择时钟模式：

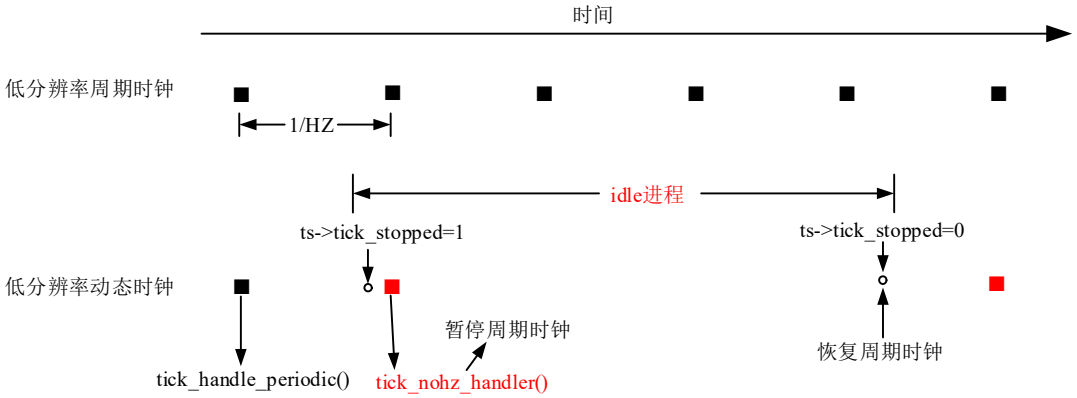
●**HZ_PERIODIC**：周期时钟。

●**NO_HZ_IDLE**：空闲动态时钟，即 CPU 运行空闲进程时暂停周期时钟，选择此选择项后会自动选择 NO_HZ_COMMON 和 TICK_ONESHOT 选项（NO_HZ 是较老的选项，等同 NO_HZ_IDLE）。

●**NO_HZ_FULL**：完全动态时钟，尽最大可能暂停周期时钟，即使 CPU 核在运行用户进程时也可能暂停周期时钟。

本节以选择了 **NO_HZ_IDLE** 选项为例，说明动态时钟的实现。如下图所示，在 CPU 核进入 idle 进

程运行时将启用动态时钟，退出 idle 进程时将退出动态时钟。



当 CPU 核运行 idle 进程时，将检测是否可暂停周期时钟，如果是则设置 ts->tick_stopped 标记（详见下文）。在下一个周期时钟事件中，处理函数 tick_nohz_handler()中将检测是否可切换至动态时钟模式，并检测 ts->tick_stopped 标记位是否置位，若是则切换至动态时钟模式，将时钟事件设备设置为单触发状态，处理函数设为 tick_nohz_handler()，但不是对时钟设备编程，即暂停周期时钟。

CPU 核在运行 idle 进程时，是禁止内核抢占的，但没有关闭中断。idle 进程中将循环检测是否需要重调度，如果是则清零 ts->tick_stopped 标记位，对时钟设备进行编程，恢复周期时钟，最后执行进程调度。

动态时钟事件处理函数 tick_nohz_handler()与周期时钟处理函数执行的工作类似，也要完成周期性的节拍工作，但是增加了对 ts->tick_stopped 标记的检测，标记置位则不对时钟设备编程（暂停周期时钟），如果标记位为 0，则对时钟设备编程，以便产生下一个周期时钟事件。

1 数据结构

tick_sched 结构体用于低分辨率动态时钟模式时控制周期时钟，在高分辨率时钟模式时模拟周期时钟。tick_sched 结构体定义在/kernel/time/tick-sched.h 头文件：

```
struct tick_sched {
    struct hrtimer    sched_timer; /*高分辨率定时器，用于仿真周期时钟（启用高分辨率时钟时）*/
    unsigned long     check_clocks;
    enum tick_nohz_mode   nohz_mode;     /*动态时钟模式*/
    ktime_t           last_tick;         /*下次时钟事件发生的时间，纳秒*/
    int               inidle;
    int               tick_stopped;      /*指示周期时钟是否已暂停*/
    unsigned long     idle_jiffies;      /*进入空闲进程时的 jiffies 计数值*/
    unsigned long     idle_calls;        /*空闲进程总的调用次数*/
    unsigned long     idle_sleeps;       /*空闲进程的调用次数，暂停了周期时钟的情况*/
    int               idle_active;       /*是否在运行 idle 进程*/
    ktime_t           idle_entrytime;
    ktime_t           idle_waketime;
    ktime_t           idle_exittime;
    ktime_t           idle_sleeptime;    /*周期时钟上一次禁用的准确时间*/
    ktime_t           iowait_sleeptime;
    ktime_t           sleep_length;      /*周期时钟禁用的时间长度*/
```

```
    unsigned long       last_jiffies;       /*下次时钟事件到期时间，jiffies*/
    u64                 next_timer;
    ktime_t             idle_expires;    /*下一个低分辨率定时器的到期时间*/
    int                 do_timer_last;   /*本 CPU 是最近一个执行 do_timer()的 CPU，在进入 idle 前*/
};
```

tick_sched 结构体主要成员简介如下：

●**sched_timer**：高分辨率定时器 hrtimer 实例，启用高分辨率时钟时此定时器用于模拟周期时钟，定时器执行函数中完成低分辨率周期时钟内的节拍工作。低分辨率时钟模式时，其到期时间记录了下一次时钟事件的时间。

●**nohz_mode**：动态时钟模式，枚举类型定义如下（/kernel/time/tick-sched.h）：

```
enum tick_nohz_mode {
    NOHZ_MODE_INACTIVE,         /*未激活动态时钟*/
    NOHZ_MODE_LOWRES,           /*低分辨率动态时钟*/
    NOHZ_MODE_HIGHRES,          /*高分辨率动态时钟*/
};
```

●**tick_stopped**：标记是否已暂停周期时钟，在 idle 进程内设置和清零。

内核在/kernel/time/tick-sched.c 文件内为每个 CPU 核定义了 tick_sched 结构体实例：

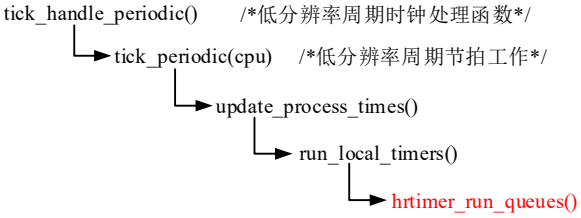
```
static DEFINE_PER_CPU(struct tick_sched, tick_cpu_sched);
```

内核在/kernel/time/tick-common.c 文件内定义了内核节拍的初始化函数，不过一般为空操作，如下所示：

```
void __init tick_init(void)
{
    tick_broadcast_init();      /*没有选择 GENERIC_CLOCKEVENTS_BROADCAST 选项为空操作*/
    tick_nohz_init();       /*没有选择 NO_HZ_FULL 为空操作，/kernel/time/tick-sched.c*/
}
```

2 切换至动态时钟

CPU 核时钟设备初始模式被设为低分辨率周期时钟模式，在周期时钟（节拍）处理函数 tick_periodic()中将调用 hrtimer_run_queues()函数判断时钟设备是否可切换为低分辨率动态时钟或高分辨率时钟模式，如果可以则执行切换，如果不能切换则保持周期时钟模式，函数调用关系如下图所示。



hrtimer_run_queues()函数定义在/kernel/time/hrtimer.c 文件内，主要用于执行时钟模式切换和处理高分辨率定时器，函数代码如下：

```
void hrtimer_run_queues(void)
```

```
{
    struct hrtimer_cpu_base *cpu_base = this_cpu_ptr(&hrtimer_bases); /*管理高分辨率定时器的结构*/
    ktime_t now;

    if (__hrtimer_hres_active(cpu_base))    /*如果已经启用了高分辨率时钟，函数返回*/
        return;

    /*切换至低分辨率动态时钟或高分辨率时钟，或者不切换*/
    if (tick_check_oneshot_change(!hrtimer_is_hres_enabled())) {
                /*切换至低分辨率动态时钟或不切换返回 0，可切换至高分辨率时钟返回 1*/
        hrtimer_switch_to_hres();      /*切换至高分辨率时钟，见下一小节*/
        return;         /*函数返回*/
    }

    /*低分辨率时钟模式（含周期时钟和动态时钟）情况下处理高分辨率定时器*/
    raw_spin_lock(&cpu_base->lock);
    now = hrtimer_update_base(cpu_base);   /*当前时间值，/kernel/time/hrtimer.c*/
    __hrtimer_run_queues(cpu_base, now);   /*处理高分辨率定时器，见下一小节*/
    raw_spin_unlock(&cpu_base->lock);
}
```

hrtimer_run_queues()函数内判断是否需要将当前 CPU 核时钟设备切换至低分辨率动态时钟模式或高分辨率时钟模式，如果是则执行切换。

如是时钟设备仍处于低分辨率时钟模式（周期或动态时钟模式），则还需要处理到期的高分辨率定时器。高分辨率定时器的定义和处理下一小节再做介绍。

tick_check_oneshot_change()函数用于确定是否可切换时钟设备工作模式，若可切换至高分辨率模式，返回 1，维持低分辨率模式返回 0。如果可以切换至低分辨率动态时钟模式，函数内会执行切换。

下面主要介绍 tick_check_oneshot_change()函数的实现。

■判断是否可切换模式

内核配置选项 HIGH_RES_TIMERS 用于确定内核是否支持高分辨率时钟模式，hrtimer_hres_enabled 全局变量表示内核是否支持高分辨率时钟。

如果选择了 HIGH_RES_TIMERS 配置选项，则 hrtimer_hres_enabled 变量初始值为 1，表示支持高分辨率时钟，否则为 0。用户可通过命令行参数“highres”修改此变量值，“highres=on”表示支持高分辨率时钟（hrtimer_hres_enabled=1），“highres=off”表示不支持（hrtimer_hres_enabled=0）。

hrtimer_is_hres_enabled()函数定义在/kernel/hrtimer.c 文件内，若没有选择 HIGH_RES_TIMERS 配置选项则函数返回 0，表示不支持高分辨率时钟。如果选择了 HIGH_RES_TIMERS 配置选项，函数返回全局变量 hrtimer_hres_enabled 值（默认为 1），由此变量值决定内核是否支持高分辨率时钟。

如果内核配置没有选择 TICK_ONESHOT 选项，tick_check_oneshot_change()函数直接返回 0，表示时钟模式不可切换，时钟设备只能处于低分辨率周期时钟状态。也就是说 TICK_ONESHOT 选项，是支持动态时钟和高分辨率时钟的必要条件。

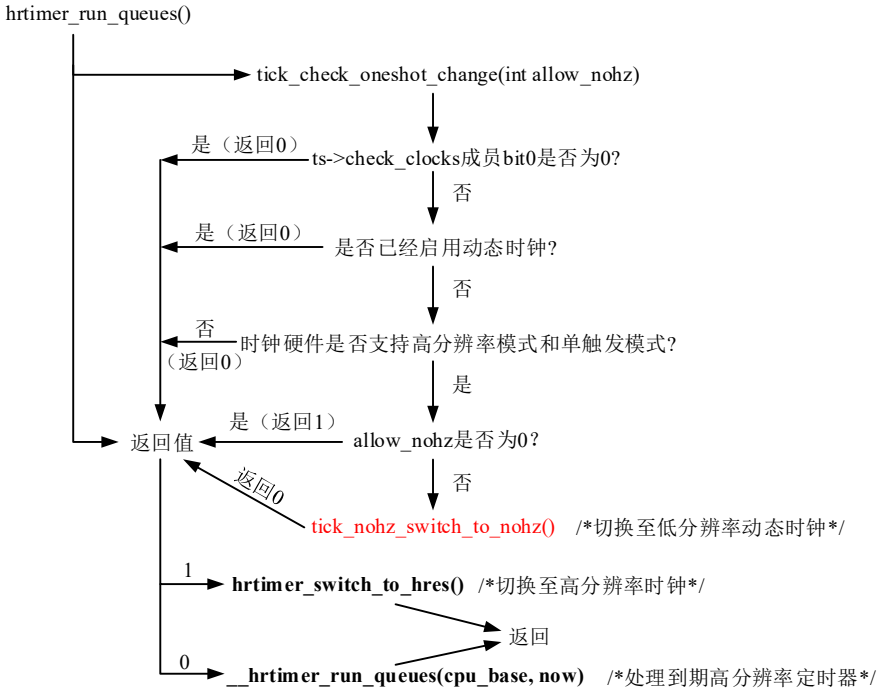
如果选择了配置选项 TICK_ONESHOT（选择 NO_HZ_IDLE 或 HIGH_RES_TIMERS 选项时默认选择

此选项），tick_check_oneshot_change()函数以 hrtimer_is_hres_enabled()函数返回值取反作为参数，如果参数值为 0（支持高分辨率时钟）且时钟设备可工作于高分辨率时钟模式，则函数 tick_check_oneshot_change()返回 1，随后 hrtimer_run_queues()函数将调用 **hrtimer_switch_to_hres()**函数将时钟设备切换至高分辨率时钟模式，hrtimer_run_queues()函数返回。

如果函数参数值为非 0 或时钟设备不能切换至高分辨率时钟模式，则 tick_check_oneshot_change()函数内将尝试将时钟设备切换至低分辨率动态时钟模式，如果可以则切换，不能切换则维持周期模式，函数返回 0。若仍维持低分辨率模式，hrtimer_run_queues()函数随后还要处理到期的高分辨率定时器。

由以上分析可知，时钟设备切换至高分辨率时钟模式优先于切换至低分辨率动态时钟模式。

tick_check_oneshot_change()函数定义在/kernel/time/tick-sched.c 文件内，函数返回 1 表示时钟设备可以切换至高分辨率时钟模式，如果切换至低分辨率动态时钟模式或者没有切换，函数返回 0，函数调用关系如下图所示。



tick_check_oneshot_change()函数代码如下：

```
int tick_check_oneshot_change(int allow_nohz)
/*allow_nohz：是否不支持高分辨率时钟模式，1 表示不支持，0 表示支持*/
{
    struct tick_sched *ts = this_cpu_ptr(&tick_cpu_sched);    /*当前 CPU 核的 tick_sched 实例*/

    if (!test_and_clear_bit(0, &ts->check_clocks))    /*清零 bit0 位并返回原 bit0 值*/
        return 0;      /*bit0 表示时钟设备是否支持单触发模式，为 0，函数返回 0*/
                       /*若时钟设备关联的时钟事件设备支持单触发模式会设置此位*/
    if (ts->nohz_mode != NOHZ_MODE_INACTIVE)    /*已经启用动态时钟，返回 0*/
        return 0;

    if (!timekeeping_valid_for_hres() || !tick_is_oneshot_available())
        return 0;      /*需计时器支持高分辨率特性且时钟事件设备支持单触发模式，否则返回 0*/
```

```
    if (!allow_nohz)    /*如果支持高分辨率动态时钟，返回 1，尝试切换至高分辨率时钟模式*/
        return 1;

    tick_nohz_switch_to_nohz();    /*尝试切换至低分辨率动态时钟模式，/kernel/time/tick-sched.c*/
    return 0;
}
```

tick_check_oneshot_change()函数调用 timekeeping_valid_for_hres()函数用于判断计时器 tk_core 关联时钟源是否支持高分辨率特性，即是否设置 CLOCK_SOURCE_VALID_FOR_HRES 标记位。因为若支持高分辨率时钟，高分辨率定时器确定到期时间时需要从计时器中获取时间值，因此也要求计时器支持高分辨率特性。

tick_is_oneshot_available()函数用于判断当前使用时钟事件设备是否支持单触发特性，即是否设置特性标记位 CLOCK_EVT_FEAT_ONESHOT。

内核计时器关联时钟源支持高分辨率特性和当前使用时钟事件设备支持单触发模式是启用高分辨率时钟或低分辨率动态时钟的必要条件。

这里我们假设内核配置没有选择 HIGH_RES_TIMERS 选项，时钟源支持高分辨率特性，时钟事件设备支持单触发模式，tick_check_oneshot_change()函数将调用 **tick_nohz_switch_to_nohz()**函数尝试将时钟设备切换至低分辨率动态时钟模式。

●切换至低分辨率动态时钟

如果内核配置选择了 NO_HZ_IDLE 选项（默认选择 NO_HZ_COMMON 选项），表示内核支持动态时钟。内核在/kernel/time/tick-sched.c 文件内将定义全局变量表示是否支持动态时钟：

```
static int tick_nohz_enabled __read_mostly  = 1;    /*内核是否使能动态时钟，默认值为 1*/
unsigned long tick_nohz_active  __read_mostly;      /*是否已启用动态时钟*/
```

另外，命令行参数"nohz="可设置 **tick_nohz_enabled** 变量值，“nohz=on”表示使能动态时钟，“nohz=off”表示禁止动态时钟。

tick_nohz_switch_to_nohz()函数定义在/kernel/time/tick-sched.c 文件内，用于尝试将时钟设备切换至低分辨率动态时钟模式，函数代码如下（如果没有选择 NO_HZ_COMMON 选项，函数为空操作）：

```
static void tick_nohz_switch_to_nohz(void)
{
    struct tick_sched *ts = this_cpu_ptr(&tick_cpu_sched);  /*当前 CPU 核对应 tick_sched 实例*/
    ktime_t next;

    if (!tick_nohz_enabled)  /*是否使能动态时钟*/
        return;

    if (tick_switch_to_oneshot(tick_nohz_handler))    /*/kernel/time/tick-oneshot.c*/
        /*切换至单触发模式，处理函数为 tick_nohz_handler()，成功返回 0，否则返回错误码*/
        return;
```

```
        hrtimer_init(&ts->sched_timer, CLOCK_MONOTONIC, HRTIMER_MODE_ABS);
                                    /*初始化仿真周期时钟高分辨率定时器，使用单调递增时间*/
        /*设置下一个时钟事件*/
        next = tick_init_jiffy_update();     /*下一个周期时钟事件时间*/

        hrtimer_forward_now(&ts->sched_timer, tick_period);   /*下一个事件时间写入仿真定时器*/
        hrtimer_set_expires(&ts->sched_timer, next);          /*设置到期时间，定时器没有激活*/
        tick_program_event(next, 1);   /*编程设置时钟设备，以便触发下一个时钟事件*/
        tick_nohz_activate(ts, NOHZ_MODE_LOWRES);
                                        /*设置启用动态时钟标记，/kernel/time/tick-sched.c*/
    }
```

tick_nohz_switch_to_nohz()函数内调用 **tick_switch_to_oneshot()**函数将时钟事件设备设置为单触发工作状态，时钟事件处理函数设为 **tick_nohz_handler()**。随后，用仿真定时器 ts->sched_timer 的到期时间记录下一个时钟事件的时间（时间间隔为 tick_period），并以此对时钟事件设备编程。

也就是说，时钟设备在切换至单触发模式后，在 tick_period 时间之后，将触发时钟事件，不过那时的事件处理函数为 **tick_nohz_handler()**。

最后，tick_nohz_switch_to_nohz()函数调用 tick_nohz_activate()函数设置 tick_sched 实例中与动态时钟相关的成员。

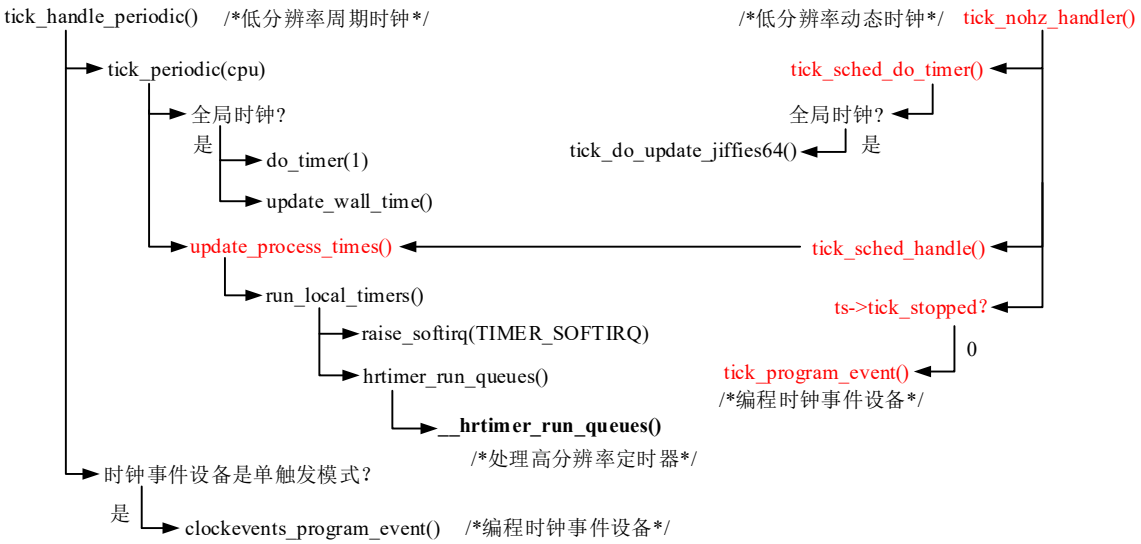
tick_nohz_activate()函数定义在/kernel/time/tick-sched.c 文件内：

```
    static inline void tick_nohz_activate(struct tick_sched *ts, int mode)
    /*mode：NOHZ_MODE_LOWRES，低分辨率动态时钟*/
    {
        if (!tick_nohz_enabled)
            return;
        ts->nohz_mode = mode;     /*设置模式为 NOHZ_MODE_LOWRES（低分辨率动态时钟）*/
        if (!test_and_set_bit(0, &tick_nohz_active))        /*测试并置位 bit0*/
                                    /*设置 tick_nohz_active 全局变量，bit0 原值为 0 则设置定时器迁移属性*/
            timers_update_migration(true);
          /*设置低/高分辨率定时器管理结构支持动态时钟和定时器可迁移属性，/kernel/time/timer.c*/
    }
```

3 动态时钟处理函数

时钟设备设为动态时钟模式后，依然会在 tick_period 时间之后产生时钟事件，时钟事件处理函数改为 tick_nohz_handler()，不再是周期时钟采用的 tick_handle_periodic()函数。

tick_nohz_handler()函数调用关系如下图所示（注意与周期时钟处理函数的对比）：



tick_nohz_handler()函数定义在/kernel/time/tick-sched.c 文件内，代码如下：

```
static void tick_nohz_handler(struct clock_event_device *dev)
{
    struct tick_sched *ts = this_cpu_ptr(&tick_cpu_sched);
    struct pt_regs *regs = get_irq_regs();
    ktime_t now = ktime_get();        /*获取当前单调递增时间*/

    dev->next_event.tv64 = KTIME_MAX;    /*下次事件时间设为最大值，暂停周期时钟*/

    tick_sched_do_timer(now);
            /*若是全局时钟，更新 jiffies64 值，计时器时间值等，/kernel/time/tick-sched.c*/
    tick_sched_handle(ts, regs);    /*完成内核节拍工作，/kernel/time/tick-sched.c*/

    if (unlikely(ts->tick_stopped))   /*是否暂停周期时钟*/
        return;        /*暂停周期时钟，不设置下一个时钟事件，函数返回*/

    /*不暂停周期时钟，编程时钟事件设备，触发下一个事件*/
    hrtimer_forward(&ts->sched_timer, now, tick_period);   /*设置仿真定时器到期时间，并未激活*/
    tick_program_event(hrtimer_get_expires(&ts->sched_timer), 1);   /*编程设置下一个时钟事件*/
}
```

tick_nohz_handler()函数首先调用 tick_sched_do_timer(now)函数判断当前时钟是否是全局时钟，如果是则调用 tick_do_update_jiffies64()函数更新 jiffies（jiffies64）值，并更新计时器时间值。需要注意的是这里并不是直接对 jiffies（jiffies64）值加 1，而通过时间间隔来计算经历的节拍数。因为如果周期时钟暂停后又恢复，可能与上次时钟事件间隔了多个节拍周期。

tick_sched_handle()函数执行内核周期节拍工作（同周期时钟处理函数），函数代码如下：

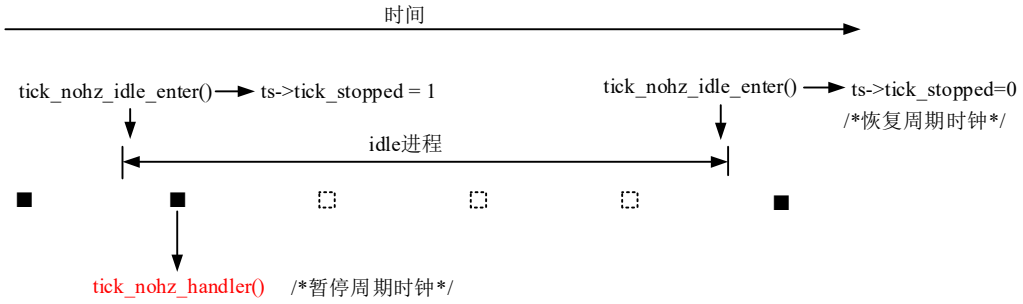
```
static void tick_sched_handle(struct tick_sched *ts, struct pt_regs *regs)
{
```

```
#ifdef CONFIG_NO_HZ_COMMON
    if (ts->tick_stopped) {            /*是否暂停周期时钟*/
        touch_softlockup_watchdog();
        if (is_idle_task(current))     /*当前是 idle 进程*/
            ts->idle_jiffies++;        /*idle 中时钟事件数量加 1*/
    }
#endif
    update_process_times(user_mode(regs));  /*完成 CPU 核节拍工作，同周期时钟，见上一小节*/
    profile_tick(CPU_PROFILING);
}
```

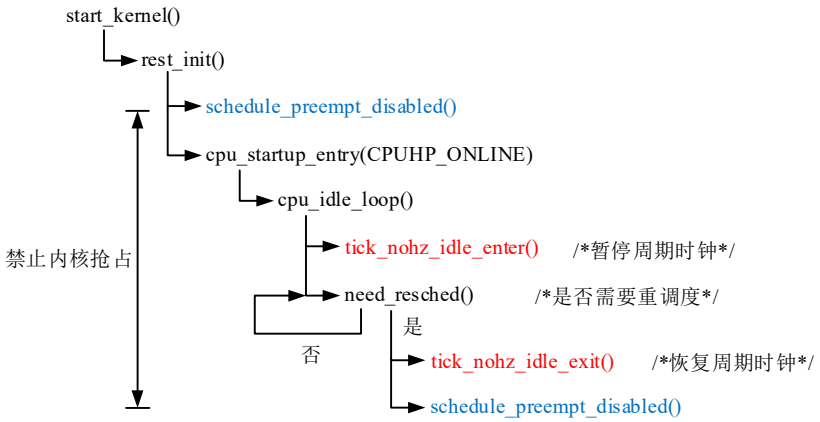
tick_nohz_handler()函数最后判断 **ts->tick_stopped** 成员是否非 0，是则表示可以暂停周期时钟，函数返回，不对时钟设备编程，暂停时钟事件。如果为 0，表示不能暂停周期时钟，则调用 tick_program_event()函数对时钟设备编程，以便触发下一个时钟事件（时间间隔为 **tick_period**，同周期时钟间隔）。

4 暂停和恢复周期时钟

由前面的介绍可知，所谓的低分辨率动态时钟只不过是在 CPU 核运行 idle 进程期间暂停周期时钟，在离开 idle 进程前恢复周期时钟。在进入 idle 进程后，将检测是否要暂停周期时钟，如果是则设置标记成员 ts->tick_stopped 值为 1。在下一个动态时钟事件处理函数中检测到此成员值非 0，则暂停周期时钟。在退出 idle 进程前将清零 ts->tick_stopped 成员值，对时钟设备编程，恢复周期时钟，如下图所示。



内核自身在初始化各子系统后，转换成空闲进程，更准确地说是内核线程，函数调用关系如下图所示：



idle 进程首先调用 tick_nohz_idle_enter()函数判断是否要暂停周期时钟，是则设置 ts->tick_stopped 标记；然后循环检测是否需要执行重调度，是则跳出循环，调用 tick_nohz_idle_exit()函数清 ts->tick_stopped

标记，恢复周期时钟。

idle 进程的前后都调用了 schedule_preempt_disabled()函数。schedule_preempt_disabled()函数首先使能内核抢占（抢占计数减 1），然后执行进程调度，最后禁止内核抢占（抢占计数减 1）。

schedule_preempt_disabled()函数定义如下（/kernel/sched/core.c）：

```
void __sched schedule_preempt_disabled(void)
{
    sched_preempt_enable_no_resched();      /*使能内核抢占，抢占计数减 1*/
    schedule();                             /*进程调度*/
    preempt_disable();                      /*禁止内核抢占，抢占计数加 1*/
}
```

由上可知，在 idle 进程中是禁止内核抢占的，不会发生进程切换。idle 进程自己会检测是否要重调度，是则主动执行进程调度，也就是说暂停、恢复周期时钟都在 idle 进程中完成。

还需要说明的一点是，这里的暂停和恢复周期时钟还要适用高分辨率时钟模式。

cpu_idle_loop()函数（/kernel/sched/idle.c）是 idle 进程的执行函数（其它 CPU 核的空闲进程也是一样），函数代码简列如下：

```
static void cpu_idle_loop(void)
{
    while (1) {             /*无限循环*/

        __current_set_polling();
        tick_nohz_idle_enter();     /*暂停周期时钟，/kernel/time/tick-sched.c*/

        while (!need_resched()) {           /*是否需要执行重调度*/
            ...                             /*空闲进程执行代码*/
        }       /*while 循环结束*/

        /*运行至此，表示需要执行重调度，要退出 idle 进程了*/
        preempt_set_need_resched();         /*空操作*/
        tick_nohz_idle_exit();      /*恢复周期时钟，/kernel/time/tick-sched.c*/
        __current_clr_polling();
        smp_mb__after_atomic();
        sched_ttwu_pending();
        schedule_preempt_disabled();        /*进程调度，/kernel/sched/core.c*/
    }
}
```

下面看一下 tick_nohz_idle_enter()和 tick_nohz_idle_exit()函数如何暂停和恢复周期时钟。

■暂停周期时钟

CPU 核进入 idle 进程后将调用 tick_nohz_idle_enter()函数判断是否可暂停周期时钟（全局时钟不能暂停），如果是则设置 ts->tick_stopped=1，在随后产生的时钟事件中（时钟中断），tick_nohz_handler()处理

函数将暂停周期时钟。

tick_nohz_idle_enter()函数定义如下（/kernel/time/tick-sched.c）：

```
void tick_nohz_idle_enter(void)
{
    struct tick_sched *ts;

    WARN_ON_ONCE(irqs_disabled());
    set_cpu_sd_state_idle();
    local_irq_disable();        /*关闭本地中断*/

    ts = this_cpu_ptr(&tick_cpu_sched);        /*CPU 核对应 tick_sched 实例*/
    ts->inidle = 1;
    __tick_nohz_idle_enter(ts);    /*/kernel/time/tick-sched.c*/

    local_irq_enable();        /*开本地中断*/
}
```

__tick_nohz_idle_enter()函数定义如下：

```
static void __tick_nohz_idle_enter(struct tick_sched *ts)
{
    ktime_t now, expires;
    int cpu = smp_processor_id();        /*当前 CPU 核编号*/

    now = tick_nohz_start_idle(ts);        /*当前单调递增时间值，ts->idle_active=1 等*/

    if (can_stop_idle_tick(cpu, ts)) {        /*是否能暂停周期时钟，/kernel/time/tick-sched.c*/
        int was_stopped = ts->tick_stopped;        /*原 ts->tick_stopped 值*/
        ts->idle_calls++;
        expires = tick_nohz_stop_sched_tick(ts, now, cpu);        /*/kernel/time/tick-sched.c*/
                                /*根据情况对时钟设备编程，设置 ts->tick_stopped 标记等*/
        if (expires.tv64 > 0LL) {
            ts->idle_sleeps++;
            ts->idle_expires = expires;
        }

        if (!was_stopped && ts->tick_stopped)        /*原 ts->tick_stopped 值由 0 改为 1*/
            ts->idle_jiffies = ts->last_jiffies;
    }
}
```

__tick_nohz_idle_enter()函数首先调用 can_stop_idle_tick()函数判断是否可暂停周期时钟，如果可以再调用 tick_nohz_stop_sched_tick()函数设置 tick_sched 实例、修改 ts->tick_stopped 值（需要暂停则置 1），另外还要根据情况对时钟设备编程，以触发下一个时钟事件。

can_stop_idle_tick()函数用于判断是否能暂停周期时钟，能暂停的条件如下（需同时满足）：

（1）ts->nohz_mode 不为 NOHZ_MODE_INACTIVE（切换为动态时钟时会修改 ts->nohz_mode 值）。

（2）不需要重调度。

（3）没有挂起的软中断。

tick_nohz_stop_sched_tick()函数判断如果需要暂停周期时钟则置位 ts->tick_stopped 值，还需要对时钟设备编程。例如，如果有高分辨率定时器到期，则要以此时间设置时钟设备；又如时钟事件暂停时长不能超过计时器关联时钟源最大计时间隔，否则会造成计时模糊等。

如果 tick_nohz_stop_sched_tick()函数置位了 ts->tick_stopped 标记，则不论情况如何都会对时钟设备编程，在下一个时钟事件的处理函数 tick_nohz_handler()中就不需要对时钟设备编程了。

can_stop_idle_tick()和 tick_nohz_stop_sched_tick()函数源代码请读者自行阅读。

■恢复周期时钟

CPU 核退出 idle 进程前将调用 tick_nohz_idle_exit()函数恢复周期时钟，函数定义如下：

```
void tick_nohz_idle_exit(void)                  /*/kernel/time/tick-sched.c*/
{
    struct tick_sched *ts = this_cpu_ptr(&tick_cpu_sched);        /*tick_sched 实例*/
    ktime_t now;

    local_irq_disable();        /*关本地中断*/
    WARN_ON_ONCE(!ts->inidle);
    ts->inidle = 0;

    if (ts->idle_active || ts->tick_stopped)
        now = ktime_get();              /*当前单调递增时间值*/

    if (ts->idle_active)            /*tick_nohz_idle_enter()函数中置 1*/
        tick_nohz_stop_idle(ts, now);    /*ts->idle_active 清 0，更新统计量等，/kernel/time/tick-sched.c*/

    if (ts->tick_stopped) {             /*已暂停了周期时钟*/
        tick_nohz_restart_sched_tick(ts, now);     /*ts->tick_stopped 清 0，时钟设备编程等*/
        tick_nohz_account_idle_ticks(ts);
    }

    local_irq_enable();       /*开本地中断*/
}
```

tick_nohz_idle_exit()函数最后调用函数 tick_nohz_restart_sched_tick()恢复周期时钟，函数代码如下。

```
static void tick_nohz_restart_sched_tick(struct tick_sched *ts, ktime_t now)
{
    tick_do_update_jiffies64(now);        /*更新 jiffies64 值和计时器值等*/
```

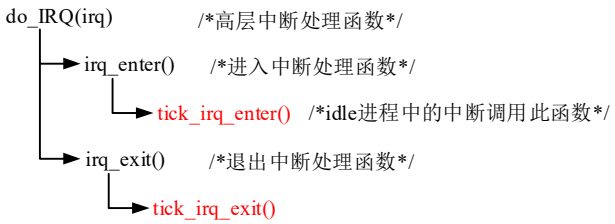
```
        update_cpu_load_nohz();

        calc_load_exit_idle();
        touch_softlockup_watchdog();
        ts->tick_stopped  = 0;
        ts->idle_exittime = now;

        tick_nohz_restart(ts, now);     /*编程时钟设备，恢复周期时钟，/kernel/time/tick-sched.c*/
    }
```

tick_nohz_restart()函数对时钟设备编程，设置产生下一次时钟事件，恢复周期时钟，源代码请读者自行阅读。

5 进出中断处理函数

前面提到在 idle 进程中禁止内核抢占，并可能暂停了周期时钟，但是并没有关闭中断，那么在 idle 进程中产生的中断是否要做特殊处理呢？

请看下面中断处理函数的调用关系，在进入退出中断处理函数时，需要处理启用动态时钟的情况：



在进入中断处理函数的 irq_enter()函数中，若当前 CPU 核在运行 idle 进程，将调用 tick_irq_enter()函数，在退出中断处理函数的 irq_exit()函数中将调用 tick_irq_exit()函数（不管是不是 idle 进程都会调用）。

■进入中断

tick_irq_enter()函数定义如下（/kernel/time/tick-sched.c）：

```
void tick_irq_enter(void)
{
    tick_check_oneshot_broadcast_this_cpu();
    tick_nohz_irq_enter();              /*/kernel/time/tick-sched.c*/
}
```

tick_nohz_irq_enter()函数定义如下：

```
static inline void tick_nohz_irq_enter(void)
{
    struct tick_sched *ts = this_cpu_ptr(&tick_cpu_sched);      /*当前 CPU 核 tick_sched 实例*/
    ktime_t now;

    if (!ts->idle_active && !ts->tick_stopped)      /*idle 进程且暂停了周期时钟才会往下执行*/
        return;
```

```
        now = ktime_get();      /*当前单调递增时间*/
        if (ts->idle_active)      /*idle 进程*/
            tick_nohz_stop_idle(ts, now);        /*ts->idle_active 清 0 等/
        if (ts->tick_stopped) {            /*暂停了周期时钟*/
            tick_nohz_update_jiffies(now);   /*更新 jiffies64、计时器时间值等，/kernel/time/tick-sched.c*/
            tick_nohz_kick_tick(ts, now);        /*空操作*/
        }
    }
```

tick_irq_enter()函数的主要工作是调用 tick_nohz_update_jiffies(now)函数更新 jiffies64 和计时器时间值等，注意在这里 CPU 核并不要求是充当全局时钟的 CPU 核，任意 CPU 核都会执行此操作。

■退出中断

在退出中断处理的 irq_exit()函数中，始终会调用 tick_irq_exit()函数，函数定义如下（/kernel/softirq.c）：

```
static inline void tick_irq_exit(void)
{
#ifdef CONFIG_NO_HZ_COMMON        /*支持动态时钟*/
    int cpu = smp_processor_id();

    if ((idle_cpu(cpu) && !need_resched()) || tick_nohz_full_cpu(cpu)) {
        if (!in_interrupt())        /*CPU 空闲，不需要重调度，退出最外层中断处理函数*/
            tick_nohz_irq_exit();      /*/kernel/time/tick-sched.c*/
    }
#endif
}
```

tick_nohz_irq_exit()函数定义如下：

```
void tick_nohz_irq_exit(void)
{
    struct tick_sched *ts = this_cpu_ptr(&tick_cpu_sched);

    if (ts->inidle)            /*tick_nohz_idle_enter()函数中置 1*/
        __tick_nohz_idle_enter(ts);          /*是否可暂停周期时钟，/kernel/time/tick-sched.c*/
    else
        tick_nohz_full_stop_tick(ts);
}
```

tick_nohz_irq_exit()函数主要是调用__tick_nohz_idle_enter()函数执行类似刚进入 idle 进程时的工作，即判断是否可暂停周期时钟，并对时钟设备编程等。

6.8.6 高分辨率时钟

时钟设备的高分辨率时钟模式是为了满足高分辨率定时器的要求，而高分辨率定时器是为了解决低分辨率定时器定时精度不高的问题。

低分辨率定时器只在内核节拍工作中处理（触发软中断处理），也就是说定时精度是一个节拍（1/HZ 秒），它不能满足某些对定时精度较高的场合，因此内核定义了高分辨率定时器。

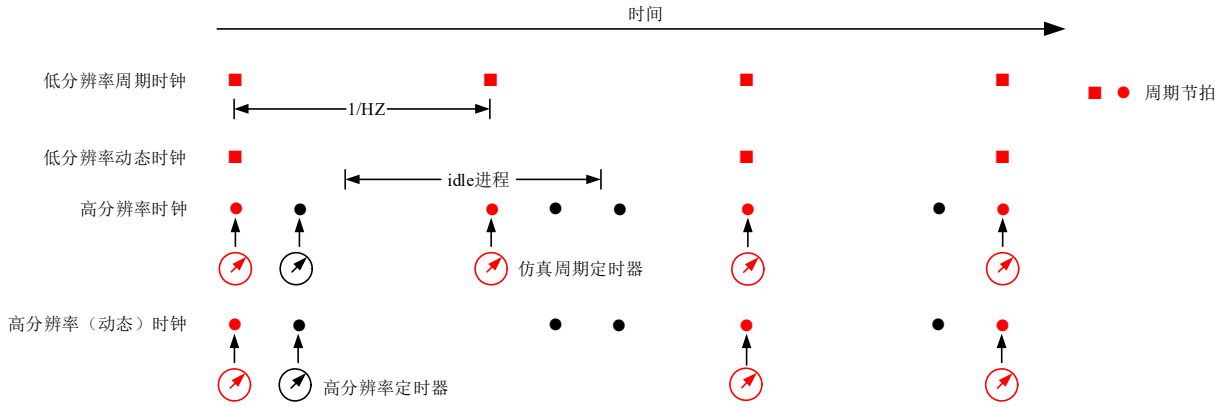
高分辨率定时器以纳秒为单位进行定时，而不是像低分辨率定时器用节拍数定时。高分辨率定时器的支持始终编译入内核，不受 HIGH_RES_TIMERS 配置选项和全局变量 hrtimer_hres_enabled 的控制。当时钟设备工作于低分辨率时钟模式时，在周期节拍工作中处理高分辨率定时器，只不过时间延迟会比较大（最大为 1/HZ 秒）。这时高辨率定时器是在周期时钟中断处理函数中处理，低分辨率定时器由软中断中处理。

如果时钟设备工作于高分辨率时钟模式（必须是单触发状态），在时钟事件处理函数中将处理到期高分辨率定时器，并利用下一个到期高分辨率定时器的到期时间去编程时钟设备，以便在下一个高分辨率定时器到期时触发时钟事件，以及时处理高分辨率定时器。

低分辨率定时器是被动地等待在内核节拍中被处理，而高分辨率定时器会主动地去编程时钟设备，使其按高分辨率定时器的到期时间产生时钟事件，在事件处理函数中处理到期定时器。

为了实现内核节拍，内核为每个 CPU 核定义了一个高分辨率定时器，定时器以 1/HZ 秒为周期到期，在定时器处理函数中执行内核节拍工作（支持动态时钟），这个定时器称它为仿真周期定时器。

高分辨率时钟事件（中断）如下图所示：



内核为每个 CPU 核定义了管理高分辨率定时器的结构，高分辨率时钟事件处理函数处理完到期定时器后，从管理结构中获取下一个到期定时器的到期时间，对时钟设备进行编程。

周期节拍在高分辨率时钟模式下由仿真周期定时器来实现，在定时器处理函数中执行周期工作（含触发处理低分辨率定时器软中断），并设置下一次到期时间，这包括启用动态时钟的情形。

在高分辨率时钟模式下，时钟事件处理函数并不需要去区分周期时钟还是动态时钟，这由仿真周期定时器执行函数处理，时钟事件处理函数把仿真周期定时器视为普通的高分辨定时器即可，不需要特别处理。因此在高分辨率时钟模式下，其事件处理函数比较简单，就是处理到期定时器，根据下一个到期高分辨率定时器到期时间编程设置时钟设备即可。

1 高分辨率定时器

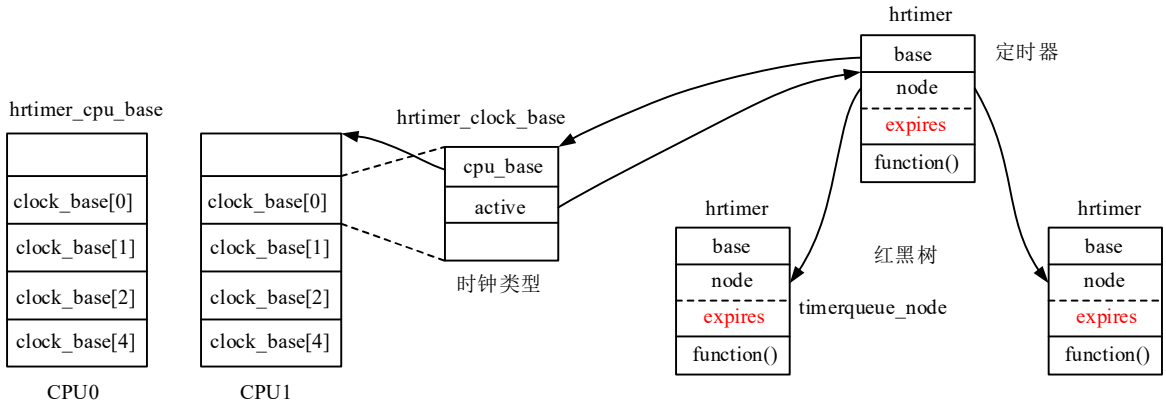
高分辨率定时器与低分辨率定时器本质上是一样的，只不过它能提供更高的定时精度，并且到期时间采用的是人类可识别的时间（单位纳秒），而不是周期节拍计数值。

内核为每个 CPU 核定义了管理高分辨率定时器的红黑树，定时器以到期时间从小到大在红黑树中从左至右排序。

处理定时器时，就是扫描红黑树中已经到期或过期的定时器，调用其执行函数。

■数据结构

高分辨率定时器管理结构如下图所示：



内核为每个 CPU 核定义了 hrtimer_cpu_base 结构体实例，用于管理高分辨率定时器。hrtimer_cpu_base 结构体中包含一个 hrtimer_clock_base 结构体数组，每个数组项对应一个时钟类型，如单调递增时间，真实时间等。hrtimer_clock_base 结构体中包含一个红黑树用于管理定时器 hrtimer 实例。

内核可以按不同的时钟类型定义定时器到期时间，定时器将添加到时钟类型对应 hrtimer_clock_base 实例中的红黑树。

高分辨率定时器由 hrtimer 结构体表示，通过 timerqueue_node 结构体成员添加到红黑树，此结构体中包含定时器的到期时间 expires。定时器按到期时间从小到大在红黑树中从左至右排列。

●管理结构

hrtimer_cpu_base 结构体用于管理某个 CPU 核的定时器，定义如下（/include/linux/hrtimer.h）：

```
struct hrtimer_cpu_base {
    raw_spinlock_t      lock;           /*保护自旋锁*/
    seqcount_t          seq;            /*顺序锁*/
    struct hrtimer      *running;       /*当前正在执行的定时器*/
    unsigned int        cpu;            /*关联 CPU 核编号*/
    unsigned int        active_bases;   /*clock_base[]数组项中具有活跃定时器的数组项位掩码*/
    unsigned int        clock_was_set_seq;
    bool                migration_enabled;  /*允许定时器在 CPU 核间迁移*/
    bool                nohz_active;        /*是否启用了动态时钟*/
#ifdef CONFIG_HIGH_RES_TIMERS       /*内核支持高分辨率定时器*/
    unsigned int            in_hrtirq  : 1,     /*处于时钟中断处理函数中*/
                            hres_active: 1,     /*时钟设备是否启用高分辨率时钟模式*/
                            hang_detected : 1;
    ktime_t                 expires_next;       /*下一到期定时器的时间*/
    struct hrtimer          *next_timer;        /*下一个到期的定时器*/
    unsigned int            nr_events;      /*定时器引发中断总数*/
    unsigned int            nr_retries;
    unsigned int            nr_hangs;       /*挂起中断总数*/
    unsigned int            max_hang_time;  /*事件处理函数花费的最长时间*/
```

```
#endif
    struct hrtimer_clock_base    clock_base[HRTIMER_MAX_CLOCK_BASES];
                                        /*时钟类型实例数组，数组项中包含管理定时器的红黑树*/
} ____cacheline_aligned;
```

hrtimer_cpu_base 结构体主要成员简介如下：

●**active_bases**：位掩码，每个比特位对应后面的 clock_base[]数组项，置位表示该数组项中管理红黑树包含定时器，不包含定时器该位清 0，在处理定时器时可跳过该数组项。

●**clock_base[]**：hrtimer_clock_base 结构体数组，每个数组项对应一个时钟类型，高分辨定时器到期时间需要采用某一时钟类型，hrtimer_clock_base 结构体用于管理基于同一时钟类型的高分辨率定时器。

clock_base[]数组项数为 **HRTIMER_MAX_CLOCK_BASES**，由枚举类型定义：

```
enum    hrtimer_base_type {
    HRTIMER_BASE_MONOTONIC,      /*单调递增时钟*/
    HRTIMER_BASE_REALTIME,       /*真实时间时钟*/
    HRTIMER_BASE_BOOTTIME,       /*系统启动时钟*/
    HRTIMER_BASE_TAI,
    HRTIMER_MAX_CLOCK_BASES,     /*时钟类型数量*/
};
```

hrtimer_clock_base 结构体定义如下（/include/linux/hrtimer.h）：

```
struct hrtimer_clock_base {
    struct hrtimer_cpu_base *cpu_base;   /*CPU 核所属 hrtimer_cpu_base 实例*/
    int             index;          /*时钟类型索引值（枚举类型）*/
    clockid_t       clockid;        /*时钟类型 id，/include/uapi/linux/time.h*/
    struct timerqueue_head  active;      /*定时器红黑树根节点*/
    ktime_t         (*get_time)(void);       /*获取时钟类型当前时间值函数*/
    ktime_t         offset;                  /*相对于单调递增时间的偏移量*/
} __attribute__((__aligned__(HRTIMER_CLOCK_BASE_ALIGN)));
```

hrtimer_clock_base 结构体 active 成员为 timerqueue_head 结构体实例，表示定时器红黑树根节点，结构体定义如下（/include/linux/timerqueue.h）：

```
struct timerqueue_head {
    struct rb_root head;      /*红黑树根节点，定时器按到期时间在红黑树中从左至右排序*/
    struct timerqueue_node *next;       /*指向红黑树中节点*/
};
```

内核在/kernel/time/hrtimer.c 文件内为每个 CPU 核定义了静态定义了 hrtimer_cpu_base 结构体实例并初始化：

```
DEFINE_PER_CPU(struct hrtimer_cpu_base, hrtimer_bases) =
{
    .lock = __RAW_SPIN_LOCK_UNLOCKED(hrtimer_bases.lock),
    .seq = SEQCNT_ZERO(hrtimer_bases.seq),
```

```
    .clock_base =
    {
        {
            .index = HRTIMER_BASE_MONOTONIC,  /*单调递增时间，定时器比较常用的时间*/
            .clockid = CLOCK_MONOTONIC,
            .get_time = &ktime_get,      /*获取单调递增时间函数*/
        },
        {
            .index = HRTIMER_BASE_REALTIME,      /*真实时间时钟类型*/
            .clockid = CLOCK_REALTIME,
            .get_time = &ktime_get_real,  /*获取真实时间值函数，单调递增时间加偏移量*/
        },
        {
            .index = HRTIMER_BASE_BOOTTIME,
            .clockid = CLOCK_BOOTTIME,
            .get_time = &ktime_get_boottime,  /*获取启动时间值函数，单调递增时间加偏移量*/
        },
        {
            .index = HRTIMER_BASE_TAI,
            .clockid = CLOCK_TAI,
            .get_time = &ktime_get_clocktai,  /*单调递增时间加偏移量*/
        },
    }
};
```

以上时间类型的时间值其实都是在单调递增时间的基础上加上一个偏移量获得。

●定时器

高分辨率定时器由 **hrtimer** 结构体表示，定义如下（/include/linux/hrtimer.h）：

```
struct hrtimer {
    struct timerqueue_node  node;        /*红黑树节点成员，插入到定时器红黑树中*/
    ktime_t          _softexpires;              /*定时器下次到期时间*/
    enum hrtimer_restart     (*function)(struct hrtimer *);      /*定时器到期时的执行函数*/
    struct hrtimer_clock_base     *base;        /*时钟类型结构体指针*/
    unsigned long              state;              /*定时器状态*/
  #ifdef CONFIG_TIMER_STATS       /*统计量*/
    int                    start_pid;
    void                   *start_site;
    char                   start_comm[16];
  #endif
};
```

hrtimer 结构体主要成员简介如下：

●node：timerqueue_node 结构体实例，包含红黑树节点成员，定义如下（/include/linux/timerqueue.h）：

```
struct timerqueue_node {
    struct rb_node node;        /*红黑树节点，插入到管理定时器红黑树*/
    ktime_t expires;            /*定时器到期时间*/
};
```

●**function**：定时器到期执行函数指针，执行函数返回值必须是 hrtimer_restart 枚举类型，定义如下：

```
enum hrtimer_restart {
    HRTIMER_NORESTART,  /*定时器不需要重启*/
    HRTIMER_RESTART,    /*定时器需要重启，如周期定时器*/
};
```

●**base**：所属时钟类型 hrtimer_clock_base 实例指针，每个定时器必须关联到某一时钟类型。

●**state**：定时器状态，取值定义如下：

```
#define  HRTIMER_STATE_INACTIVE    0x00    /*定时器未激活*/
#define  HRTIMER_STATE_ENQUEUED    0x01    /*定时器激活，在红黑树中*/
```

■定时器操作

高分辨率定时器操作函数定义在/include/linux/hrtimer.h 头文件或/kernel/time/hrtimer.c 文件内，例如：

●void **hrtimer_init**(struct hrtimer *timer, clockid_t which_clock,enum hrtimer_mode **mode**)：初始化高分辨率定时器实例。

which_clock 参数表示时钟类型如：CLOCK_REALTIME、CLOCK_MONOTONIC 等。

mode 参数为 hrtimer_mode 枚举类型，定义在/include/linux/hrtimer.h 头文件，表示定时器模式：

```
enum hrtimer_mode {
    HRTIMER_MODE_ABS = 0x0,         /*使用绝对时间值*/
    HRTIMER_MODE_REL = 0x1,         /*相对于当前的时间值*/
    HRTIMER_MODE_PINNED = 0x02,     /*定时器绑定到 CPU */
    HRTIMER_MODE_ABS_PINNED = 0x02,     /*定时器绑定到 CPU，使用绝对时间*/
    HRTIMER_MODE_REL_PINNED = 0x03,     /*定时器绑定到 CPU，使用相对时间*/
};
```

which_clock 和 mode 参数决定了定时器关联到 hrtimer_cpu_base 实例中哪个 hrtimer_cpu_base 实例数组项，即采用的时钟类型。

hrtimer_init()函数只是初始化 hrtimer 实例部分成员，尚未设置到期时间，未设置到期执行函数，也没有将实例添加到红黑树结构中。

●void **hrtimer_set_expires**(struct hrtimer *timer, ktime_t time)：设置定时器到期时间为 time。

●u64 **hrtimer_forward**(struct hrtimer *timer, ktime_t now, ktime_t interval)：设置定时器下一次到期时间在 now 之后（now 一般为当前时间），interval 表示在旧的到期时间（上一次到期时间）基础上延长到期时间的步长，即下一个到期时间为 old+n*interval>now（n 为满足此不等式的最小整数）。例如：定时器旧到期时间为 5，now 为 12，interval 为 2，则下一个到期时间为 13，从旧到期时间开始最少增加 4 个步长为 2 的时间间隔才能超过 12，即 5+4*2>12。

●void **hrtimer_start**(struct hrtimer *timer, ktime_t tim,const enum hrtimer_mode mode)：设置定时器到期时间并将定时器添加到管理结构中（红黑树），tim 表示时间值，mode 参数表示 tim 时间是绝对时间还是相对时间。如果新添加定时器在时钟设备下次事件之前到期，则需要对时钟设备编程，修改下次事件时间。

●int hrtimer_cancel(struct hrtimer *timer)：注销定时器并等待其执行完。

■处理定时器

内核在启动函数 start_kernel()内调用 hrtimers_init()函数完成高分辨率定时器初始化，初始化负责将可阻塞通知注册到 CPU 通知链，函数定义如下（/kernel/time/hrtimer.c）：

```
static struct notifier_block    hrtimers_nb = {     /*可阻塞通知实例*/
        .notifier_call = hrtimer_cpu_notify,        /*通知处理函数*/
};

void __init hrtimers_init(void)
{
        hrtimer_cpu_notify(&hrtimers_nb, (unsigned long)CPU_UP_PREPARE, \
                                                (void *)(long)smp_processor_id());   /*调用通知执行函数*/
                                                  /*初始化 hrtimer_cpu_base 实例，/kernel/time/hrtimer.c*/
        register_cpu_notifier(&hrtimers_nb);        /*将通知实例注册到 cpu_chain 通知链，/kernel/cpu.c*/
}
```

通知执行函数 hrtimer_cpu_notify()完成 hrtimer_cpu_base 实例的进一步初始化，如 hrtimer_cpu_base 成员的初始化。register_cpu_notifier()函数将 hrtimers_nb 通知实例注册到 cpu_chain 通知链，当 CPU 支持热插拔时（SMP），在再次激活 CPU 时将再次执行 hrtimer_cpu_notify()函数。

除启动 CPU 核外，其它 CPU 核启动时也会执行 cpu_chain 通知链中通知。

不管时钟设备是低分辨率时钟还是高分辨率时钟，事件处理函数都会调用**__hrtimer_run_queues()**函数处理到期高分辨率定时器。高分辨率定时器的处理直接在时钟事件处理函数中（中断处理函数中）执行，不像低分辨率定时器一样在软中断中处理。

__hrtimer_run_queues()函数定义在/kernel/time/hrtimer.c 文件内：

```
static void __hrtimer_run_queues(struct hrtimer_cpu_base *cpu_base, ktime_t now)
/*now：当前时间，处到到期时间在 now 及之前的定时器*/
{
        struct hrtimer_clock_base *base = cpu_base->clock_base;
        unsigned int active = cpu_base->active_bases;

        for (; active; base++, active >>= 1) { /*遍历 hrtimer_cpu_base 实例中 hrtimer_clock_base 实例数组*/
                struct timerqueue_node *node;
                ktime_t basenow;

                if (!(active & 0x01))   /*跳过不含定时器的数组项*/
                        continue;
```

```
            basenow = ktime_add(now, base->offset);    /*基准时间*/

            while ((node = timerqueue_getnext(&base->active))) {    /*红黑树中下一定时器实例*/
                struct hrtimer *timer;
                timer = container_of(node, struct hrtimer, node);      /*从红黑树节点获取定时器实例*/
                if (basenow.tv64 < hrtimer_get_softexpires_tv64(timer))     /*到期时间大于基准时间*/
                    break;     /*退出循环*/
                __run_hrtimer(cpu_base, base, timer, &basenow);
                                                    /*处理到期定时器，/kernel/time/hrtimer.c*/
            }
        }
    }
```

__hrtimer_run_queues()函数不难理解，函数通过 active 变量遍历包含定时器的 hrtimer_clock_base 实例，从每个 hrtimer_clock_base 实例的红黑树中逐个取出到期时间在 basenow（基准时间）之前的定时器实例，调用__run_hrtimer()函数处理定时器。

__run_hrtimer()函数首先将定时器从红黑树中移出，然后调用定时器注册的执行函数 function()，并根据执行函返回值（HRTIMER_RESTART 或 HRTIMER_NORESTART）确定是否将定时器重新插入到管理红黑树中。__run_hrtimer()函数没有修改定时器到期时间，因此需要定时器执行函数修改其下次到期时间，插入定时器时不会对时钟设备进行编程（不同于 hrtimer_start()函数）。__run_hrtimer()函数源代码请读者自行阅读。

2 仿真周期定时器

若时钟设备工作于高分辨时钟模式时，内核周期节拍工作依然要周期地产生。不过此时节拍工作由仿真周期定时器（高分辨率定时器）完成，它以 1/HZ 的周期到期。仿真周期定时器执行函数中将执行节拍工作，并重新设置下一次到期时间，以便周期性地到期。

如果启用了动态时钟，在 idle 进程内，仿真周期定时器将暂停，在 CPU 核退出 idle 进程前，将重新激活仿真周期定时器。

tick_sched 结构体中 sched_timer 成员是高分辨率定时器实例，即仿真周期定时器，在其执行函数中完成周期节拍工作。

```
struct tick_sched {                    /*每个 CPU 核对应一个 tick_sched 结构体实例*/
    struct hrtimer      sched_timer;                    /*高分辨率定时器*/
    ...
}
```

在切换高分辨率时钟模式的 hrtimer_switch_to_hres()函数中（见下文），将调用 **tick_setup_sched_timer()** 函数初始化、激活仿真周期定时器，函数定义如下（/kernel/time/tick-sched.c）：

```
void tick_setup_sched_timer(void)
{
    struct tick_sched *ts = this_cpu_ptr(&tick_cpu_sched);    /*当前 CPU 核 tick_sched 实例*/
    ktime_t now = ktime_get();      /*当前单调递增时间值*/
```

```
    /*初始化仿真周期定时器*/
    hrtimer_init(&ts->sched_timer, CLOCK_MONOTONIC, HRTIMER_MODE_ABS);   /*初始化*/
    ts->sched_timer.function = tick_sched_timer;      /*定时器执行函数，见下文*/

    hrtimer_set_expires(&ts->sched_timer, tick_init_jiffy_update());   /*设置定时器到期时间*/

    /* Offset the tick to avert jiffies_lock contention. */
    if (sched_skew_tick) {          /*初始值为 0，可通过"skew_tick"命令行参数设置*/
        u64 offset = ktime_to_ns(tick_period) >> 1;
        do_div(offset, num_possible_cpus());
        offset *= smp_processor_id();
        hrtimer_add_expires_ns(&ts->sched_timer, offset);     /*调整定时器到期时间*/
    }

    hrtimer_forward(&ts->sched_timer, now, tick_period);   /*设置定时器到期时间，步长 tick_period*/
    hrtimer_start_expires(&ts->sched_timer, HRTIMER_MODE_ABS_PINNED);   /*激活定时器*/
    tick_nohz_activate(ts, NOHZ_MODE_HIGHRES);            /*设置 tick_sched 模式*/
        /*设置 ts->nohz_mode 成员（需选择 NO_HZ_COMMON 选项），/kernel/time/tick-sched.c*/
}
```

tick_setup_sched_timer()函数内设置仿真周期定时器的执行函数为 **tick_sched_timer()**，设置定时器下一个到期时间为当前时间加上周期时长 tick_period，并激活定时器。最后如果选择了 NO_HZ_COMMON 选项，还需要设置 tick_sched 实例中与动态时钟相关的成员。

■定时器执行函数

仿真周期定时器到期时，时钟事件处理函数中将调用其执行函数 **tick_sched_timer()**，执行周期节拍工作，函数定义如下（/kernel/time/tick-sched.c）：

```
static enum hrtimer_restart tick_sched_timer(struct hrtimer *timer)
{
    struct tick_sched *ts =container_of(timer, struct tick_sched, sched_timer);      /*tick_sched 实例*/
    struct pt_regs *regs = get_irq_regs();
    ktime_t now = ktime_get();          /*当前单调递增时间*/

    tick_sched_do_timer(now);
                        /*如果是全局时钟，更新 jiffies64 和计时器值，/kernel/time/tick-sched.c*/
    if (regs)       /*是否在中断上下文中，高分辨率定时器在中断处理函数中处理，条件成立*/
        tick_sched_handle(ts, regs);    /*完成周期节拍工作，同低分辨率动态时钟处理函数*/

    if (unlikely(ts->tick_stopped))      /*是否要暂停周期时钟，是则无需重启仿真周期定时器*/
        return    HRTIMER_NORESTART;        /*不需要重启*/
```

```
    /*需要重启仿真周期定时器*/
    hrtimer_forward(timer, now, tick_period);   /*设置仿真周期定时器下一次到期时间*/

    return HRTIMER_RESTART;     /*需要重新激活定时器*/
}
```

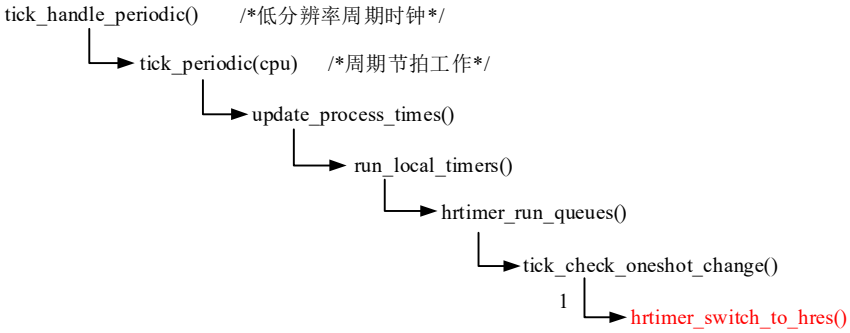
tick_sched_timer()函数与低分辨率动态时钟的 tick_nohz_handler()处理函数非常相似。

tick_sched_timer()函数内判断当前时钟是否是全局时钟，如果是则更新全局变量 jiffies64 和计时器时间值，然后调用 **tick_sched_handle(ts, regs)**函数（同低分辨率动态时钟）完成周期节拍工作，最后判断是否需要暂停周期时钟（动态时钟）。如果是则返回 HRTIMER_NORESTART，表示不需要重启仿真周期定时器，如果不需要暂停周期时钟则设置仿真周期定时器下一次到期时间（间隔 tick_period），函数返回值为 HRTIMER_RESTART，表示需要重启定时器。

内核在处理该定时器时，如果返回值是 HRTIMER_RESTART，会将定时器重新添加到定时器管理结构中，以便下次到期时被处理。

3 启用高分辨时钟模式

前面介绍的低分辨率周期时钟事件处理函数的节拍工作函数 tick_periodic()中将判断是否需要将时钟设备切换至高分辨时钟模式（优先于切换至低分辨率动态时钟模式），函数调用关系如下：



tick_check_oneshot_change()函数用于判断是否可切换至高分辨率时钟模式，是则返回 1，否则返回 0。hrtimer_switch_to_hres()函数执行切换操作。

如果内核配置选择了 HIGH_RES_TIMERS 选项（没有设置 highres=off”命令行参数）、计时器关联时钟源支持高分辨率特性且时钟事件设备支持单触发模式，tick_check_oneshot_change()函数将返回 1，随后将调用 **hrtimer_switch_to_hres()**函数启用高分辨率时钟。

hrtimer_switch_to_hres()函数定义在/kernel/time/hrtimer.c 文件内，代码如下：

```
static int hrtimer_switch_to_hres(void)
{
    struct hrtimer_cpu_base *base = this_cpu_ptr(&hrtimer_bases);
                        /*当前 CPU 核 hrtimer_cpu_base 实例，用于管理高分辨率定时器*/
    if (tick_init_highres()) {      /*修改时钟设备模式，成功返回 0，/kernel/tick-oneshot.c*/
        ...             /*输出信息*/
        return 0;
    }
    base->hres_active = 1;      /*启用了高分辨率时钟模式*/
    hrtimer_resolution = HIGH_RES_NSEC;   /*1 纳秒，最高分辨率*/
```

```
        tick_setup_sched_timer();
                                /*初始化、激活仿真周期时钟定时器，见上文，/kernel/time/tick-sched.c*/
        retrigger_next_event(NULL);  /*编程时钟设备，/kernel/time/hrtimer.c*/
        return 1;
    }
```

hrtimer_switch_to_hres()函数调用 tick_init_highres()函数（**tick_switch_to_oneshot(hrtimer_interrupt)**）将当前 CPU 时钟设备设为单触发模式，时钟事件设备处理函数设为 **hrtimer_interrupt()**，然后调用函数 **tick_setup_sched_timer()**函数初始化、激活仿真周期时钟定时器，见上文。

hrtimer_switch_to_hres()函数最后调用 retrigger_next_event()函数对时钟设备编程，设置下一个时钟事件产生的时间为最近一个高分辨率定时器到期的时间，请读者自行阅读函数源代码。

4 高分辨率时钟处理函数

时钟设备切换至高分辨率时钟模式后，其事件处理函数设为 **hrtimer_interrupt()**，此函数将在时钟硬件中断处理函数中被调用，主要用于处理到期的高分辨率定时器。

hrtimer_interrupt()函数定义在/kernel/time/hrtimer.c 文件内，代码如下：

```
void hrtimer_interrupt(struct clock_event_device *dev)
{
    struct    hrtimer_cpu_base *cpu_base = this_cpu_ptr(&hrtimer_bases);
                                                    /*CPU 核 hrtimer_cpu_base 实例*/
    ktime_t expires_next, now, entry_time, delta;
    int retries = 0;

    BUG_ON(!cpu_base->hres_active);
    cpu_base->nr_events++;
    dev->next_event.tv64 = KTIME_MAX;   /*最大时间值*/

    raw_spin_lock(&cpu_base->lock);
    entry_time = now = hrtimer_update_base(cpu_base);      /*当前时间值，开始处理定时器的时间*/
retry:
    cpu_base->in_hrtirq = 1;
    cpu_base->expires_next.tv64 = KTIME_MAX;

    __hrtimer_run_queues(cpu_base, now);        /*处理到期、过期的高分辨率定时器，见上文*/
    expires_next = __hrtimer_get_next_event(cpu_base);   /*下一个最近到期定时器的到期时间*/
    cpu_base->expires_next = expires_next;
    cpu_base->in_hrtirq = 0;
    raw_spin_unlock(&cpu_base->lock);

    if (!tick_program_event(expires_next, 0)) { /*编程设置下一时钟事件，/kernel/time/tick-oneshot.c*/
                                                /*编程成功返回 0*/
```

```
            cpu_base->hang_detected = 0;
            return;
        }

        /*如果以下一个定时器到期时间编程时钟设备不成功，执行以下代码*/
        raw_spin_lock(&cpu_base->lock);
        now = hrtimer_update_base(cpu_base);       /*更新当前时间*/
        cpu_base->nr_retries++;
        if (++retries < 3)     /*执行三次处理到期定时器*/
            goto retry;     /*再次扫描到期的定时器进行处理*/

    /*执行三次处理到期定时器后，对时钟设备编程还不成功，执行以下代码*/
    cpu_base->nr_hangs++;
    cpu_base->hang_detected = 1;
    raw_spin_unlock(&cpu_base->lock);
    delta = ktime_sub(now, entry_time);     /*本次处理定时器所花的时间，纳秒*/
    if ((unsigned int)delta.tv64 > cpu_base->max_hang_time)       /*处理定时器花费的最大时间*/
            cpu_base->max_hang_time = (unsigned int) delta.tv64;

    if (delta.tv64 > 100 * NSEC_PER_MSEC)    /*处理定时器时间超过 100 毫秒*/
        expires_next = ktime_add_ns(now, 100 * NSEC_PER_MSEC);
                                                /*下次时钟事件延期 100 毫秒*/
    else
        expires_next = ktime_add(now, delta);       /*下次时钟事件延期 delta 时间*/
   tick_program_event(expires_next, 1);              /*编程设置时钟设备*/
   printk_once(KERN_WARNING "hrtimer: interrupt took %llu ns\n",ktime_to_ns(delta));
}
```

hrtimer_interrupt()函数不难理解，函数调用__hrtimer_run_queues()函数处理到期、过期的定时器，随后从定时器红黑树中查找下一个到期的定时器，以其到期时间编程时钟设备，成功则返回，即在下一个定时器到期时再产生时钟事件。

若以下一定时器到期时间对时钟设备编程不成功，则可能需要多次执行__hrtimer_run_queues()函数，然后再对时钟设备编程，还不成功则以更长的到期时间对时钟设备编程。

前面介绍的仿真周期定时器与其它定时器一样，也由红黑树管理，在其到期执行函数中执行周期节拍工作。

6.8.7 时间统计与定时

本小节介绍的时间统计是指内核对进程运行时长的统计（可由用户读取），定时指的是时间框架提供给用户进程的定时器功能，用户进程可设置定时器。定时器到期将向进程发送信号，默认是会终止进程，但进程可设置信号处理函数。内核基于进程定时功能还实现了进程的休眠功能。

1 时间统计

进程结构中与时间相关成员如下所示：

```
struct task_struct {
    ...
    cputime_t utime, stime, utimescaled, stimescaled;
    cputime_t gtime;
    ...
}
```

task_struct 结构体中时间相关成员语义如下：

●**utime**：进程在用户态运行的时长。

●**utimescaled**：utime 根据 CPU 频率缩放的时长，等于 utime。

●**stime**：进程在内核态运行的时长。

●**stimescaled**：stime 根据 CPU 频率缩放的时长，等于 stime。

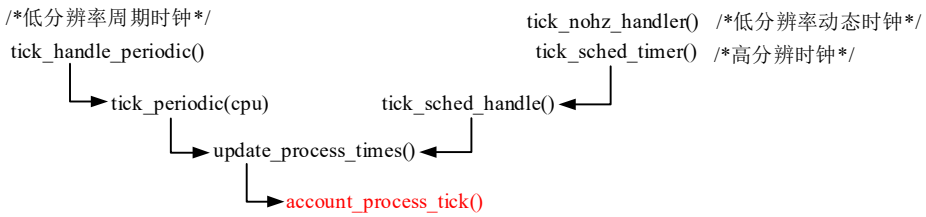
以上成员在进程创建时初始化为 0。

表示时长的 cputime_t 结构体定义在/include/asm-generic/cputime-nsecs.h 头文件：

```
typedef   u64 __nocast   cputime_t;        /*64 位无符号整数，纳秒数表示的时间*/
typedef   u64 __nocast   cputime64_t;

#define   cputime_one_jiffy  jiffies_to_cputime(1)      /*一个节拍内的纳秒数*/
```

在/include/asm-generic/cputime-nsecs.h 头文件还定义了 cputime_t 与 jiffies 之间的相互转换函数，请读者自行阅读。

内核在周期节拍工作中将统计进程时间，函数调用关系如下图所示：



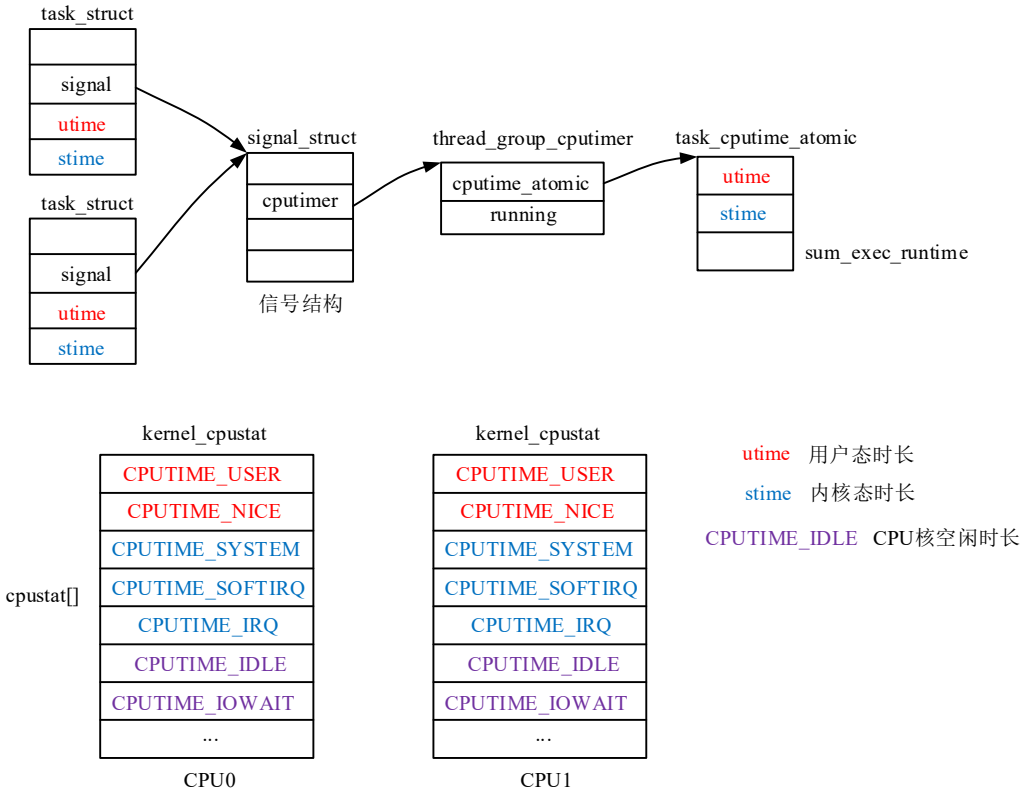
account_process_tick()函数定义如下（/kernel/sched/cputime.c）：

```
void account_process_tick(struct task_struct *p, int user_tick)
/*p：当前进程结构，user_tick：周期节拍发生在用户空间还是内核空间*/
{
    cputime_t one_jiffy_scaled = cputime_to_scaled(cputime_one_jiffy);   /*一个节拍所含纳秒数*/
    struct rq *rq = this_rq();

    if (vtime_accounting_enabled())      /*没有选择 VIRT_CPU_ACCOUNTING，返回 false*/
        return;
```

```
    if (sched_clock_irqtime) {     /*没有选择 IRQ_TIME_ACCOUNTING，sched_clock_irqtime 为 0*/
        irqtime_account_process_tick(p, user_tick, rq, 1);
        return;
    }

    if (steal_account_process_tick())
                        /*没有选择 PARAVIRT 选项，返回 false，/kernel/sched/cputime.c*/
        return;

    if (user_tick)
        account_user_time(p, cputime_one_jiffy, one_jiffy_scaled);      /*统计用户态运行时长*/
    else if ((p != rq->idle) || (irq_count() != HARDIRQ_OFFSET))
        account_system_time(p, HARDIRQ_OFFSET, cputime_one_jiffy,one_jiffy_scaled);
                                                                    /*统计内核态运行时长*/
    else
        account_idle_time(cputime_one_jiffy);       /*统计 CPU 核空闲时长*/
}
```

account_process_tick()函数在内核节拍中采样当前进程是在用户态运行还是内核态运行，又或者是 CPU 核处于空闲状态，将节拍周期时长累加到不同的统计项上。

除了 task_struct 结构体中统计了进程的运行时长，在进程关联的信号结构中统计了线程组内所有线程的运行时长，内核还为每个 CPU 核定义了 kernel_cpustat 结构体实例，用于统计 CPU 核在各状态下运行的时长，如下图所示。



信号 signal_struct 结构体中关联的数据结构在/include/linux/sched.h 头文件内定义，主要是统计线程组

内所有线程的时长。

kernel_cpustat 结构体定义在/include/linux/kernel_stat.h 头文件，结构体中是一个 64 位无符号整数，用于统计 CPU 核在各状态运行的时长。内核在/kernel/sched/core.c 文件内为每个 CPU 核定义了此结构体实例（percpu 变量）。

account_process_tick()函数中调用的 account_user_time()等函数定义在/kernel/sched/cputime.c 文件内，函数内将时长累加到信号关联结构体中成员（调用函数在/kernel/sched/stats.h 头文件）和 kernel_cpustat 实例数组项。

kernel_cpustat 结构体中的数组项对 CPU 核运行状态进行了更细的划分，例如：CPUTIME_NICE 数组项统计 nice 值大于 0 的进程在用户态运行时长，CPUTIME_USER 数组项统计 nice 值小等于 0 的进程在用户态运行时长，CPUTIME_SYSTEM 数组项表示系统调用时长，CPUTIME_IOWAIT 和 CPUTIME_IDLE 数组项统计的是 CPU 核空闲时长。

用户进程可通过系统调用获取当前进程运行时长的统计量（/kernel/sys.c），例如：

●**times**(struct tms __user * tbuf)：tbuf 参数指向 tms 结构体，保存进程运行时长信息，这里是线程组内所有线程时长的统计量，系统调用返回当前 jiffies64 值。

tms 结构体定义如下（/include/uapi/linux/times.h）：

```
struct tms {
    __kernel_clock_t tms_utime;      /*用户态运行时长*/
    __kernel_clock_t tms_stime;      /*内核态运行时长*/
    __kernel_clock_t tms_cutime;     /*执 wait()系统调用的所有已终止子进程在用户态运行时长*/
    __kernel_clock_t tms_cstime;     /*执 wait()系统调用的所有已终止子进程在内核态运行时长*/
};
```

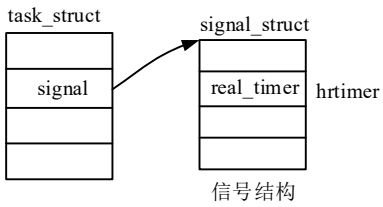
__kernel_clock_t 以是内核节拍计数的值，除以 HZ 才是秒。

2 定时器与休眠

定时器是进程规划自己在未来某一时刻接获通知的一种机制。休眠则能使进程暂停执行一段时间，在 Linux 中休眠也是通过定时器来实现。

■定时器

线程组关联的信号 signal_struct 结构体中包含一个高分辨率定时器成员，用于实现定时功能，注意这个定时器是线程组内所有线程共用的。



在复制信号结构的 copy_signal()函数中将对 real_timer 定时器初始化，相关代码如下：

```
static int copy_signal(unsigned long clone_flags, struct task_struct *tsk)
{
    ...
    hrtimer_init(&sig->real_timer, CLOCK_MONOTONIC, HRTIMER_MODE_REL);
    sig->real_timer.function = it_real_fn;      /*定时器到期执行函数为 it_real_fn()*/
```

```
    ...
}
```

real_timer 到期执行函数为 **it_real_fn()**，定义如下（/kernel/time/itimer.c）：

```
enum hrtimer_restart it_real_fn(struct hrtimer *timer)
{
    struct signal_struct *sig =container_of(timer, struct signal_struct, real_timer);

    trace_itimer_expire(ITIMER_REAL, sig->leader_pid, 0);
    kill_pid_info(SIGALRM, SEND_SIG_PRIV, sig->leader_pid);

    return HRTIMER_NORESTART;
}
```

it_real_fn()函数内将向信号结构中 sig->leader_pid 指向的线程组长发送 SIGALRM 信号。SIGALRM 信号默认会终止线程组（致命信号）。若不想线程组终止，则需要为信号设置信号处理函数。

用户进程可通过 **setitimer()**系统调用（/kernel/time/itimer.c）设置并激活 real_timer 定时器：

```
int setitimer(int which, struct itimerval __user * value,struct itimerval __user * ovalue)
```

which 参数值和 itimerval 结构体定义在/include/uapi/linux/time.h：

which 参数取值如下：

```
#define   ITIMER_REAL       0    /*以真实时间倒计时，到期产生 SIGALARM 信号*/
#define   ITIMER_VIRTUAL    1
                        /*以进程虚拟时间（用户态下运行时长）倒计时，到期产生 SIGVTALRM*/
#define   ITIMER_PROF       2
                        /*以进程时间（用户态和内核态）运行时长倒计时，到期产生 SIGPROF 信号*/
```

若为 ITIMER_VIRTUAL 和 ITIMER_PROF 参数，将使用后面介绍的 POSIX 定时器，以上信号默认都会使线程组终止。

value 和 ovalue 指向 itimerval 结构体，前者用于设置新时间，后者用于返回原时间，结构体定义如下：

```
struct itimerval {
    struct timeval    it_interval;    /*秒和微秒表示的时间，赋予 signal->it_real_incr 成员*/
    struct timeval    it_value;       /*到期时间*/
};
```

it_value 表示定时器到期时间。若 it_interval 为 0，定时器将是一次性的，只到期一次，若 it_interval 非 0，在到期一次之后，以 it_interval 为间隔周期到期。

在处理信号时，在取出信号的 dequeue_signal()函数中，若是 SIGALARM 信号，且 signal->it_real_incr 不为 0，则再次设置并激活 real_timer 定时器（到期时间再加上 signal->it_real_incr）。

int getitimer(int which, struct itimerval __user * value)系统调用用于读取定时器信息，源代码请读者自行阅读。

另外，**alarm**(unsigned int seconds)系统调用用于按秒设置 real_timer 定时器，这是一个老式的接口，可由库函数实现。

■休眠

内核在/kernel/time/hrtimer.c 定义了 **nanosleep()**系统调用，原型如下：

int nanosleep(struct timespec __user * rqtp,struct timespec __user * rmtp);

rqtp 表示休眠时长（进程进入睡眠，纳秒级的），rmtp 表示唤醒后的剩余时长，因为可能还没到睡眠时长进程就被信号唤醒了。

nanosleep()系统调用实现原理是在进程栈中定义、设置并激活一个高分辨率定时器（不是 real_timer 定时器），进程进入睡眠，定时器到期后在其执行函数中唤醒进程。

库函数 sleep()等就是基于 nanosleep()系统调用实现的。

另外，内核代码中也可以使当前进程进入睡眠，到期将进程唤醒，接口函数如下（/kernel/time/timer.c）：

●**schedule_timeout**(signed long timeout)：timeout 参数以节拍数计时，不改变进程状态，定时 timeout 后唤醒进程（由低分辨率定时器实现）。若 timeout 参数为 MAX_SCHEDULE_TIMEOUT，进程将无限期睡眠，不能由定时器唤醒，须由其它内核代码唤醒。timeout 大于 0 可由定时器唤醒，也可被其它代码唤醒，函数返回剩余睡眠时长。

●**schedule_timeout_interruptible**(signed long timeout)：进入可中断睡眠，内部调用 schedule_timeout()函数。

●**schedule_timeout_uninterruptible**(signed long timeout)：进入不可中断睡眠，调用 schedule_timeout()函数实现。

●**schedule_timeout_killable**(signed long timeout)：进入中度睡眠（可被致命信号唤醒），内部调用函数 schedule_timeout()实现。

●**msleep**(unsigned int msecs)：毫秒级的睡眠，调用 schedule_timeout_uninterruptible()函数实现，必须睡眠指定时长才会重新运行原进程。

6.8.8 POSIX 时钟与定时器

Linux 内核中还支持 POSIX 标准时钟 API，主要包括时间值的设置和获取，以及定时器的应用。POSIX 时钟和定时器的主要实现代码位于/kernel/time/posix-timers.c 文件内。

1 数据结构

POSIX 标准定义了一些标准的时钟类型，Linux 内核还定义了两个非标准的时钟类型，下面分别做简要介绍。

■标准时钟类型

内核中 POSIX 标准时钟类型如下（/include/uapi/linux/time.h）

```
#define   CLOCK_REALTIME                  0       /*真实时间*/
#define   CLOCK_MONOTONIC                 1       /*单调递增时间*/
#define   CLOCK_PROCESS_CPUTIME_ID        2       /*当前进程时间*/
#define   CLOCK_THREAD_CPUTIME_ID         3       /*当前线程运行时间*/
#define   CLOCK_MONOTONIC_RAW             4       /*原始单调递增时间*/
```

```
#define   CLOCK_REALTIME_COARSE      5
#define   CLOCK_MONOTONIC_COARSE     6
#define   CLOCK_BOOTTIME             7      /*启动时间*/
#define   CLOCK_REALTIME_ALARM       8
#define   CLOCK_BOOTTIME_ALARM       9
#define   CLOCK_SGI_CYCLE            10          /* Hardware specific */
#define   CLOCK_TAI                  11

#define   MAX_CLOCKS                 16       /*时钟类型数量*/
```

内核中的 POSIX 时钟和定时器需基于某一时钟类型，每种时钟类型对应一个 k_clock 结构体实例，用于设置/获取时间，创建和设置定时器等。

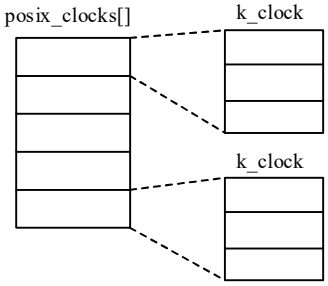
k_clock 结构体定义如下（/include/linux/posix-timers.h）：

```
struct k_clock {
    int (*clock_getres) (const clockid_t which_clock, struct timespec *tp);     /*获取分辨率，必须定义*/
    int (*clock_set) (const clockid_t which_clock,const struct timespec *tp);      /*设置时间*/
    int (*clock_get) (const clockid_t which_clock, struct timespec * tp);        /*获取时间，必须定义*/
    int (*clock_adj) (const clockid_t which_clock, struct timex *tx);
    int (*timer_create) (struct k_itimer *timer);       /*初始化 POSIX 定时器*/
    int (*nsleep) (const clockid_t which_clock, int flags,struct timespec *, struct timespec __user *);
    long (*nsleep_restart) (struct restart_block *restart_block);
    int (*timer_set) (struct k_itimer * timr, int flags,
                      struct itimerspec * new_setting,struct itimerspec * old_setting);
                                                          /*设置并激活 POSIX 定时器*/
    int (*timer_del) (struct k_itimer * timr);        /*删除 POSIX 定时器*/
#define TIMER_RETRY 1
    void (*timer_get) (struct k_itimer * timr,struct itimerspec * cur_setting);     /*获取 POSIX 定时器*/
};
```

内核在/kernel/time/posix-timers.c 文件内定义了 k_clock 结构体数组，每个时钟类型对应一项：

```
static struct k_clock    posix_clocks[MAX_CLOCKS];
```

k_clock 结构体数组如下图所示：



注册 k_clock 实例的函数声明如下（/kernel/time/posix-timers.c）：

```
void posix_timers_register_clock(const clockid_t clock_id,struct k_clock *new_clock);
```

clock_id 参数表示时钟类型，new_clock 指向 k_clock 实例，函数实现很简单就是将 new_clock 指向 k_clock 实例的数据复制到 posix_clocks[clock_id]数组项，也就是说后面注册的实例会覆盖前面的注册的实例（注册工作由内核完成）。

内核在/kernel/time/posix-timers.c 文件内注册了部分 k_clock 实例，初始化函数定义如下：

```
static __init int init_posix_timers(void)
{
    struct k_clock clock_realtime = {
        ...
    };
    struct k_clock clock_monotonic = {
        ...
    };
    struct k_clock clock_monotonic_raw = {
        ...
    };
    struct k_clock clock_realtime_coarse = {
        ...
    };
    struct k_clock clock_monotonic_coarse = {
        ...
    };
    struct k_clock clock_tai = {
        ...
    };
    struct k_clock clock_boottime = {
        ...
    };
    /*注册 k_clock 实例*/
    posix_timers_register_clock(CLOCK_REALTIME, &clock_realtime);
    posix_timers_register_clock(CLOCK_MONOTONIC, &clock_monotonic);
    posix_timers_register_clock(CLOCK_MONOTONIC_RAW, &clock_monotonic_raw);
    posix_timers_register_clock(CLOCK_REALTIME_COARSE, &clock_realtime_coarse);
    posix_timers_register_clock(CLOCK_MONOTONIC_COARSE, &clock_monotonic_coarse);
    posix_timers_register_clock(CLOCK_BOOTTIME, &clock_boottime);
    posix_timers_register_clock(CLOCK_TAI, &clock_tai);

    posix_timers_cache = kmem_cache_create("posix_timers_cache",
                                sizeof (struct k_itimer), 0, SLAB_PANIC,NULL);
                                  /*创建 POSIX 定时器 k_itimer 结构体 slab 缓存*/
    return 0;
}
__initcall(init_posix_timers);
```

内核在/kernel/time/posix-cpu-timers.c 文件内还注册了两个 k_clock 实例，如下所示：

```
static __init int init_posix_cpu_timers(void)
{
    struct k_clock process = {          /*进程时钟*/
        ...
    };
    struct k_clock thread = {           /*线程时钟*/
        ...
    };
    struct timespec ts;
    posix_timers_register_clock(CLOCK_PROCESS_CPUTIME_ID, &process);
    posix_timers_register_clock(CLOCK_THREAD_CPUTIME_ID, &thread);

    cputime_to_timespec(cputime_one_jiffy, &ts);
    onecputick = ts.tv_nsec;
    WARN_ON(ts.tv_sec != 0);
    return 0;
}
__initcall(init_posix_cpu_timers);
```

另外，内核还在/kernel/time/alarmtimer.c 文件内注册了时钟类型 CLOCK_REALTIME_ALARM 和 CLOCK_BOOTTIME_ALARM 对应的 k_clock 实例，这两个时钟类型主要用于设置 alarm 定时器，后面将介绍。

■非标准时钟类型

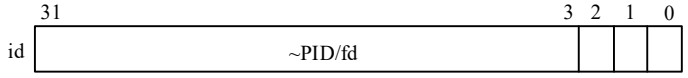
POSIX 标准时钟类型对应的 k_clock 实例都由内核注册，其实它们都是基于内核单调递增时间、真实时间或原始单调递增时间实现的。

Linux 内核还定义了两种非标准的时钟类型暂且称它们为动态时钟和 CPU 时钟类型。

动态时钟是指可由外部设备驱动程序注册的时钟类型，内核视它为外设，通过设备文件访问，通过文件可获取/设置时间和设置定时器。

CPU 时钟是指可获取本线程组内任意线程（线程组）的时间，并设置定时器的时钟类型。前面介绍标准的 CLOCK_PROCESS_CPUTIME_ID 和 CLOCK_THREAD_CPUTIME_ID 时钟类型只适用于当前进程/线程。

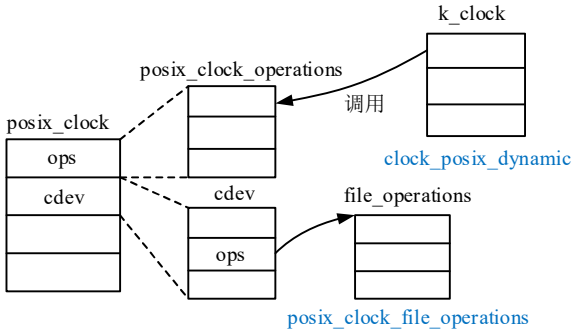
POSIX 标准时钟类型的标识取值大于 0，非标准时钟类型的标识为负值，如下图所示：



高 29 位为线程 PID（CPU 时钟）或设备文件描述符 fd（动态时钟）取反，bit2 表示 CPU 时钟是适用于进程（线程组）还是线程，最低两位取值语义为：0 进程总运行时长，1 用户空间运行时长，2 调度器计算的运行时长，3 表示是动态时钟类型（高 29 位为~fd）。

动态时钟相关代码位于/kernel/time/posix-clock.c 文件内，外设驱动程序需定义 posix_clock 结构体实例，并向内核注册。注册此实例会同时注册字符设备，进程通过设备文件（文件描述符）访问操作此设备，源代码请读者自行阅读。

在/kernel/time/posix-clock.c 文件内定义的动态时钟对应的 k_clock 实例 clock_posix_dynamic，实例与注册的 posix_clock 实例的关系如下图所示：



用户进程通过参数传递表示动态时钟 posix_clock 实例的文件描述符 fd，由 fd 查找到 posix_clock 实例，k_clock 实例中的函数调用 posix_clock_operations 实例中的函数，用于获取/设置时间，设置定时器等。

CPU 时钟 k_clock 实例 clock_posix_cpu 定义在/kernel/time/posix-cpu-timers.c 文件内，文件内初始化函数 init_posix_cpu_timers()还注册了 CLOCK_PROCESS_CPUTIME_ID 和 CLOCK_THREAD_CPUTIME_ID 时钟类型对应的 k_clock 实例，源代码请读者自行阅读。

非标准时钟类型对应的 clock_posix_dynamic 和 clock_posix_cpu 实例只定义不注册。

2 设置/获取时间

POSIX 标准获取时间的系统调用为 clock_gettime()，设置时间的系统调用为 clock_settime()，调整时间的系统调用为 clock_adjtime()，获取时钟分辨率的系统调用为 clock_getres()。下面主要介绍获取时间系统调用的实现。

获取时间值的 clock_gettime()系统调用实现代码如下：

```
SYSCALL_DEFINE2(clock_gettime, const clockid_t, which_clock,struct timespec __user *,tp)
/*which_clock：本小节开始处介绍的时钟类型*/
{
    struct k_clock *kc = clockid_to_kclock(which_clock);      /*时钟类型转 k_clock 实例*/
    struct timespec kernel_tp;
    int error;

    if (!kc)
        return -EINVAL;
    error = kc->clock_get(which_clock, &kernel_tp);        /*获取时间值*/
    if (!error && copy_to_user(tp, &kernel_tp, sizeof (kernel_tp)))      /*复制到用户空间*/
        error = -EFAULT;
    return error;
}
```

clock_gettime()系统调用实现比较简单，由时钟类型查找 k_clock 实例，然后调用其中的 clock_get()

函数获取时间值，返回给用户空间。

时钟类型 which_clock 在 Linux 中可以是负数，表示动态时钟或 CPU 时钟类型。由时钟类型转 k_clock 实例的 clockid_to_kclock()函数定义如下：

```
static struct k_clock *clockid_to_kclock(const clockid_t id)
{
    if (id < 0)
        return (id & CLOCKFD_MASK) == CLOCKFD ?       /*bit[1,0]=11*/
            &clock_posix_dynamic : &clock_posix_cpu;        /*动态时钟或 CPU 时钟类型*/

    if (id >= MAX_CLOCKS || !posix_clocks[id].clock_getres)
        return NULL;
    return &posix_clocks[id];       /*id>0，直接返回 posix_clocks[id]实例指针*/
}
```

设置时间的**clock_settime()**系统调用与 clock_gettime()系统调用类似，调用 k_clock 实例中的 clock_set()函数设置时间，源代码请读者自行阅读。

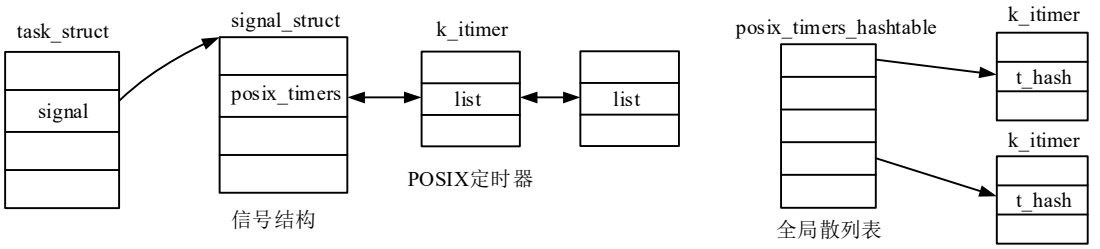
3 定时器

POSIX 时钟也提供了定时器和休眠功能，休眠功能通常是基于普通的高分辨率定时器实现的，与系统调用 nanosleep()实现相同。POSIX 休眠系统调用为 clock_nanosleep()，请读者自行阅读实现代码，下面主要介绍 POSIX 定时器的操作。

k_clock 实例中的 timer_create()函数用于初始化 POSIX 定时器，timer_set()函数用于设置并激活定时器，timer_get()函数用于获取定时器信息，timer_del()函数用于删除定时器。POSIX 定时器相关系统调用中调用 k_clock 实例中的相关函数实现。

■数据结构

POSIX 定时器结构如下图所示：



POSIX 定时器（间隔定时器）由 k_itimer 结构体表示，定时器实例由全局散列表管理，线程创建的定时器由关联的信号结构通过双链表管理。

k_itimer 结构体定义如下（/include/linux/posix-timers.h）：

```
struct k_itimer {
    struct list_head list;      /*双链表成员*/
    struct hlist_node t_hash;   /*散列表*/
    spinlock_t it_lock;
    clockid_t it_clock;         /*时钟类型*/
```

```
        timer_t it_id;              /* timer id */
        int it_overrun;             /* overrun on pending signal */
        int it_overrun_last;        /* overrun on last delivered signal */
        int it_requeue_pending;     /* waiting to requeue this timer */
    #define REQUEUE_PENDING 1
        int it_sigev_notify;            /*定时器到期通知用户进程的方式*/
        struct signal_struct *it_signal;
        union {
            struct pid *it_pid;     /* pid of process to send signal to */
            struct task_struct *it_process;     /* for clock_nanosleep */
        };
        struct sigqueue *sigq;          /*指向 sigqueue 实例，分配定时器时分配*/
        union {
            struct {
                struct hrtimer timer;       /*高分辨率定时器，通用的定时器*/
                ktime_t interval;
            } real;
            struct cpu_timer_list cpu;
            struct {
                unsigned int clock;
                unsigned int node;
                unsigned long incr;
                unsigned long expires;
            } mmtimer;
            struct {
                struct alarm alarmtimer;        /*alarm 定时器*/
                ktime_t interval;
            } alarm;
            struct rcu_head rcu;
        } it;
    };
```

对于大部分的 POSIX 时钟类型，POSIX 定时器都是使用 timer 成员表示的高分辨率定时器，定时器添加到前面介绍的基于 CPU 核的红黑树中。

对于 alarm 定时器（时钟类型为 CLOCK_REALTIME_ALARM 或 CLOCK_BOOTTIME_ALARM）使用 alarmtimer 成员表示的 alarm 定时器。alarm 定时器相关代码位于/kernel/time/alarmtimer.c 文件内。

alarm 定时器由 alarm 结构体表示，定义如下（/include/linux/alarmtimer.h）：

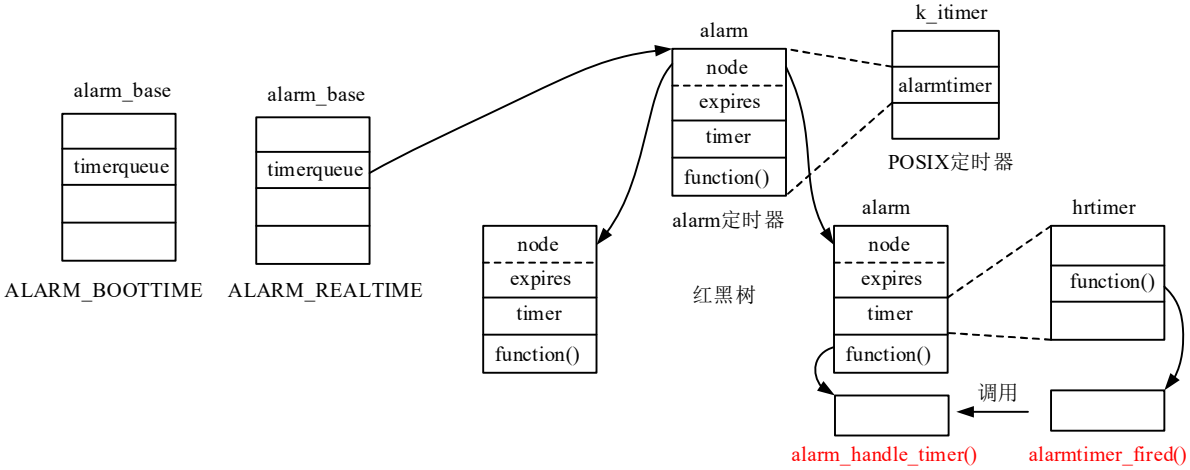
```
struct alarm {
    struct timerqueue_node  node;     /*添加到 alarm_base 结构体中红黑树*/
    struct hrtimer          timer;          /*高分辨率定时器*/
    enum alarmtimer_restart (*function)(struct alarm *, ktime_t now);       /*定时器到期处理函数*/
    enum alarmtimer_type    type;     /*类型（BOOTTIME/REALTIME）*/
```

```
    int             state;
    void            *data;
};
```

alarm 定时器可以周期到期的，到期执行函数 function()返回值为枚举类型如下所示：

```
enum alarmtimer_restart {
    ALARMTIMER_NORESTART,       /*不需要重启*/
    ALARMTIMER_RESTART,         /*需要重启*/
};
```

alarm 结构体中的 timer 高分辨率定时器添加到基于 CPU 核的红黑树中，另外 alarm 实例还会添加到 alarm_base 结构体的红黑树中，如下图所示：



内核定义了两个 alarm_base 结构体实例，分别基于真实时间和启动时间。alarm 内嵌高分辨率定时器的到期执行函数调用 alarm 实例中的到期执行函数。

■创建定时器

内核提供了 POSIX 定时器的相关系统调用，如下所示（/kernel/time/posix-timers.c）：

- **timer_create()**：创建 POSIX 定时器，返回定时器 ID。
- **timer_settime()**：启动或停止定时器。
- **timer_delete()**：删除定时器。
- **timer_gettime()**：通过定时器 ID 获取定时器剩余时间。
- **timer_getoverrun()**：获取定时器溢出计数，到期发送了信号，但是信号没有被处理的次数。

下面以创建 POSIX 定时器的 timer_create()系统调用为例，介绍其实现，其它系统调用实现函数请读者自行阅读。

在介绍 timer_create()系统调用前，先看 sigevent 结构体的定义，它用于传递定时器到期时内核向用户进程的通知方式。

sigevent 结构体定义如下（/include/uapi/asm-generic/siginfo.h）：

```
typedef struct sigevent {
    sigval_t sigev_value;
    int sigev_signo;            /*到期发送的信号*/
```

```
    int sigev_notify;     /*通知方式，信号或调用函数等*/

    union {
        int _pad[SIGEV_PAD_SIZE];
        int _tid;         /*将信号发送给线程（当前同组线程）*/
        struct {
            void (*_function)(sigval_t);         /**/
            void *_attribute;     /* really pthread_attr_t */
        } _sigev_thread;
    } _sigev_un;
} sigevent_t;
```

sigevent 结构体主要成员简介如下：

●**sigev_notify**：通知方式，取值如下：

```
  #define   SIGEV_SIGNAL            0     /*通过信号通知*/
  #define   SIGEV_NONE              1     /*不通知*/
  #define   SIGEV_THREAD            2
                                  /*在线程中执行_function()函数，好像没有这功能了，有待确认！*/
  #define   SIGEV_THREAD_ID         4     /*发送信号给_tid 线程*/
```

●**sigev_value**：sigval 联合体，定义如下：

```
  typedef union sigval {
      int sival_int;
      void __user *sival_ptr;
  } sigval_t;
```

当通知方式为信号时，且为实时信号，sigev_value 成员表示信号的伴随数据。

创建 POSIX 定时器的 timer_create()系统调用实现函数简列如下：

```
SYSCALL_DEFINE3(timer_create, const clockid_t, which_clock,
                struct sigevent __user *, timer_event_spec,timer_t __user *, created_timer_id)
/*which_clock：时钟类型，timer_event_spec：sigevent 实例指针，created_timer_id：保存定时器 ID*/
{
    struct k_clock *kc = clockid_to_kclock(which_clock);       /*时钟类型转 k_clock 实例*/
    struct k_itimer *new_timer;          /*POSIX 定时器指针*/
    int error, new_timer_id;
    sigevent_t event;
    int it_id_set = IT_ID_NOT_SET;
    ...
    new_timer = alloc_posix_timer();         /*分配 k_itimer 实例*/
    ...
    spin_lock_init(&new_timer->it_lock);
    new_timer_id = posix_timer_add(new_timer);     /*添加到散列表，返回 ID 值*/
    ...
    it_id_set = IT_ID_SET;
```

```
    new_timer->it_id = (timer_t) new_timer_id;
    new_timer->it_clock = which_clock;
    new_timer->it_overrun = -1;

    if (timer_event_spec) {
        if (copy_from_user(&event, timer_event_spec, sizeof (event))) {     /*复制用户数据*/
            ...
        }
        rcu_read_lock();
        new_timer->it_pid = get_pid(good_sigevent(&event));       /*目标线程 PID 或线程组长*/
        rcu_read_unlock();
        ...
    } else {
        memset(&event.sigev_value, 0, sizeof(event.sigev_value));
        event.sigev_notify = SIGEV_SIGNAL;
        event.sigev_signo = SIGALRM;              /*默认发送 SIGALRM 信号*/
        event.sigev_value.sival_int = new_timer->it_id;
        new_timer->it_pid = get_pid(task_tgid(current));
    }
    /*设置定时器*/
    new_timer->it_sigev_notify         = event.sigev_notify;
    new_timer->sigq->info.si_signo = event.sigev_signo;        /*信号编号*/
    new_timer->sigq->info.si_value = event.sigev_value;       /*信号私有数据*/
    new_timer->sigq->info.si_tid      = new_timer->it_id;
    new_timer->sigq->info.si_code    = SI_TIMER;           /*信号代码*/

    if (copy_to_user(created_timer_id,&new_timer_id, sizeof (new_timer_id))) {    /*复制定时器 ID*/
        ...
    }

    error = kc->timer_create(new_timer);   /*调用 k_clock 实例中 timer_create()函数，初始化定时器*/
    ...
    spin_lock_irq(&current->sighand->siglock);
    new_timer->it_signal = current->signal;
    list_add(&new_timer->list, &current->signal->posix_timers);
                                                  /*添加到 current->signal->posix_timers 双链表*/
    spin_unlock_irq(&current->sighand->siglock);
    return 0;          /*成功返回 0*/
    ...
}
```

■处理定时器

在设置/激活定时器的 timer_settime()系统调用中将调用 k_clock 实例中 **timer_set()**函数为定时器中的高分辨率定时器设置执行函数，一般为 **posix_timer_fn()**函数（除了 alarm 定时器）。

alarm 定时器在创建时为其内嵌高分辨率定时器设置执行函数为 alarmtimer_fired()，它将调用 alarm 定时器的执行函数 alarm_handle_timer()。

下面以 posix_timer_fn()函数为例说明定时器到期的处理，函数代码简列如下：

```
static enum hrtimer_restart posix_timer_fn(struct hrtimer *timer)
{
    struct k_itimer *timr;
    unsigned long flags;
    int si_private = 0;
    enum hrtimer_restart ret = HRTIMER_NORESTART;

    timr = container_of(timer, struct k_itimer, it.real.timer);
    spin_lock_irqsave(&timr->it_lock, flags);

    if (timr->it.real.interval.tv64 != 0)
        si_private = ++timr->it_requeue_pending;

    if (posix_timer_event(timr, si_private)) {        /*向进程发送信号，成功返回 0*/
        if (timr->it.real.interval.tv64 != 0) {       /*发送不成功，重启定时器*/
            ktime_t now = hrtimer_cb_get_time(timer);
    #ifdef CONFIG_HIGH_RES_TIMERS
            {
                ktime_t kj = ktime_set(0, NSEC_PER_SEC / HZ);
                if (timr->it.real.interval.tv64 < kj.tv64)
                    now = ktime_add(now, kj);
            }
    #endif
            timr->it_overrun += (unsigned int)hrtimer_forward(timer, now,timr->it.real.interval);
            ret = HRTIMER_RESTART;
            ++timr->it_requeue_pending;
        }
    }
    unlock_timer(timr, flags);
    return ret;
}
```

POSIX 定时器到期时将向线程组或指定组内线程发送信号，如果进程没有为信号设置处理函数，默认会终止线程组。进程可为信号设置处理函数，改变信号的默认处理。

6.9 并发与同步

在编写内核代码和驱动程序时，需要注意对共享资源的保护。共享资源就是内核中公共的变量、数据结构等，例如：全局变量、管理某种数据结构实例的双链表、红黑树等等。共享资源是数据而不是代码，访问共享资源的代码段称为临界区。

所谓并发就是指多个内核路径同时进入临界区，访问和操作同一共享数据，就有可能出现相互覆盖共享数据的情况，造成被访问数据的不一致。

举个简单的例子，假设有两个内核路径都需要对变量 a 执行加 1 操作，每个路径的执行步骤是将 a 的值载入寄存器，然后在寄存器中执行加 1 操作，最后将寄存器中结果写回内存。如果路径 A 在载入变量 a 的值后被路径 B 中断了，B 对 a 加了 1 并写入了内存，然后 A 路径继续运行，对寄存器中的值加 1（最开始 a 的值加 1），再写回内存，此时 A 写回的值会覆盖 B 写回的值，a 的值最终只加了 1，造成结果的不正确。

对共享数据的一致性状态控制称为同步控制。并发可能会引发数据的不同步，对并发的控制就是为了维护共享数据的同步（一致性）。

为了维护数据的一致性，使并发串行化是比较常用的手段，即某个时刻只让一个内核路径进入访问同一个共享数据的临界区，其它路径在临界区外等待，等到前一个路径退出临界区后，下一个路径再进入临界区访问共享数据。

内核通过各种类型的锁机制来保证并发的串行化。锁有很多种，各有不同的特征和适用场景，如原子变量、自旋锁、顺序锁、信号量、互斥量、RCU 机制等。但是锁并不是绝对禁止并发，例如读写自旋锁、读写信号量、RCU 机制等允许多个读路径同时访问共享数据（写路径不行）。

最好的并发控制就是避免不必要的数据共享，不需要加锁，允许并发发生。percpu 变量就是无锁化的一个例子，每个 CPU 核具有私有的数据结构实例，不与其它 CPU 核共享。

除了并发外，编译器在优化过程中会对指令重新排序，高性能处理器的指令乱序执行，也可能会造成数据的不一致。可以通过人工添加屏障的方法来保证指令的顺序执行，以得到正确的结果。对抗编译器的指令重排的屏障叫编译器屏障，对抗处理器乱序执行的屏障叫内存屏障。屏障就像一条界线，禁止屏障前的操作屏障后的操作乱序。

6.9.1 屏障

屏障是一种保证内存访问顺序的方法，用来解决下列内存访问乱序的问题。

（1）编译器编译代码时可能重新排列汇编指令，使编译出来的程序在处理器上运行更快，但是有时候优化的结果可能不符合程序员的意图。

（2）现代的处理器采用超标量体系结构和乱序执行技术，能够在一个时钟周期并行执行多条指令。处理器按照程序顺序取出一批指令，分析找出没有依赖关系的指令，发给多个独立的执行单元并行执行，最后按照程序顺序提交执行结果。用一句话总结就是“顺序取指，乱序执行，顺序提交执行结果”。有些情况不允许乱序执行，必须严格按照顺序，可是处理器不能识别出依赖关系。常见的情况是处理器访问外围设备控制器的寄存器，例如查询有些外围设备的状态值，需要先向控制寄存器写入数值，然后再从状态寄存器读取状态值，顺序不能乱。

（3）在多处理器系统中，硬件工程师使用存储缓冲区、使用无效队列协助缓存和缓存一致性协议实现高性能，引入了处理器之间的内存访问乱序问题。一个处理器修改数据，可能不会把数据立即同步到自己的缓存或者其他处理器缓存，导致其处理器不能立即看到最新的数据。

内核主要支持 2 种屏障：（1）编译器屏障，（2）内存屏障。

1 编译器屏障

为了提高程序的执行速度，编译器会优化代码，对于不存在数据依赖或控制依赖的汇编指令，可能重新排列它们的顺序。但是有时候优化产生的指令顺序不符合程序员的真实意图，程序员需要使用编译器屏障指导编译器。编译器屏障是：

barrier();

它阻止编译器把屏障一侧的指令移动到另一侧，既不能把屏障前面的指令移动到屏障后面，也不能把屏障后的指令移动到屏障前面。编译器屏障也称为编译器优化屏障。

barrier()定义如下（/include/linux/compiler-gcc.h）：

#define barrier() __asm__ __volatile__("": : :"memory")

关键字"__volatile__"告诉编译器：禁止优化代码，不改变 barrier()前面的代码块、barrier()和后面代码块的顺序。

嵌入式汇编代码中的破坏列表"memory"告诉编译器：内存中变量的值可能变化，不要继续使用加载到寄存器中的值，应该重新从内存中加载变量的值。

内核定义了 READ_ONCE()、WRITE_ONCE()和 ACCESS_ONCE()（计划淘汰），它们可以看作 barrier()的弱化形式，只阻止编译器对单个变量的优化。

C 语言的关键字“volatile”也可以阻止编译器对单个变量的优化。读写外设寄存器的__raw_readl()和__raw_writel()等函数就使用了这个关键字，详见/include/asm-generic/io.h 头文件。

2 内存屏障

内存屏障用于解决内存时序一致性问题。CPU 的内存时序一致性模型有严格一致性、顺序一致性、处理器一致性、松散一致性（弱一致性）等模型。现代高性能 CPU 大多使用松散一致性模型，访存指令存在乱序执行情况。对抗访存指令在处理器上乱序执行的内存屏障有多种，主要举列如下。

●**mb()**：全屏障，可以对抗读内存操作和写内存操作的乱序执行。

●**rmb()**：读屏障，可以对抗读内存操作的乱序执行，不干预写内存操作。

●**wmb()**：写屏障，可以对抗写内存操作的乱序执行，不干预读内存操作。

●**smp_mb()**：多处理器版全屏障。

●**smp_rmb()**：多处理器版读屏障。

●**smp_wmb()**：多处理器版写屏障。

一般来说，内存屏障仅是一条界线，能够保证界线前后的访存指令不会交错执行。

处理器与外设寄存器之间也存在一致性问题，也需要以上强制性内存屏障来解决。还有另外一些内存屏障用来解决处理器与外设之间内存一致性，例如：

●**dma_rmb()**：DMA 读屏障。

●**dam_wmb()**：DMA 写屏障。

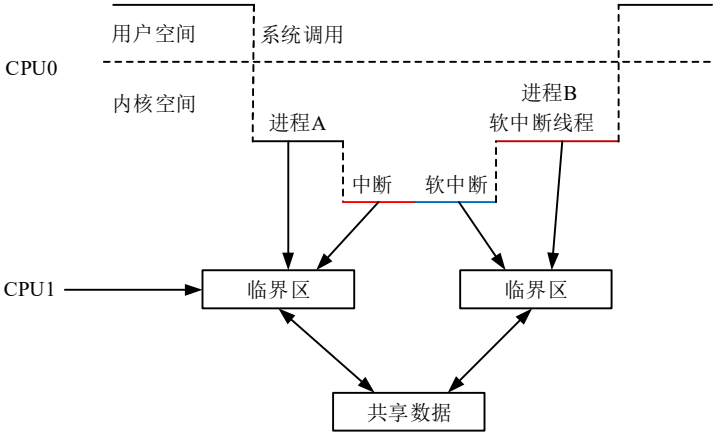
以上内存屏障的实现都是特定于体系结构的，通常由特定的指令实现，这里就不展开介绍了。

6.9.2 单核处理器并发控制

本小节主要介绍在单核处理器上的并发控制。单核处理器上不存在真正的并发执行，宏观上的并发实际上只是交错执行。

临界区是指访问和操作共享数据的代码段，共享数据无法同时被多个执行路径访问（修改），访问临界区的执行线程或代码路径称为并发源。

对于单核处理器，并发源可能来自异常（系统调用）、中断、软中断等，如下图所示：



并发源的位置有以下几种：

（1）进程系统调用（异常处理函数）。

（2）中断处理函数。

（3）软中断处理函数和软中断处理线程。

由于在单核处理器中只有一个执行路径，因此只有一个并发源在临界区被抢占时，才可能与下一个并发源同时访问共享数据。例如：进程 A 正在临界区访问共享数据，发生了中断，在中断处理函数或软中断中可能会进入同一个临界区或访问同一个共享数据；又或者进程 A 在临界区时发生了中断，中断返回前执行了进程调度（内核抢占），运行进程 B，进程 B 正好也要访问共享数据（进程 B 可能是软中断线程）。

在单核处理器中为了保护共享数据，可以在进入临界区时，通过禁止内核抢占、禁止软中断或关闭中断等方式来屏蔽其它的并发源，在离开临界区时再打开，以保证临界区能够原子性（不被中断）地执行完。

单核处理器中的并发控制包括以下方法：

（1）禁止内核抢占：可防止不同进程（含软中断线程）同时进入临界区。

（2）禁止软中断（同时禁止了内核抢占）：可防止不同进程和软中断同时进入临界区。

（3）禁止本地硬中断：可防止不同进程、软中断和硬中断同时进入临界区。

CPU 核在访问本地数据时，如 percpu 变量，可通过以上方法来保护共享数据。

在多核处理器中，每个 CPU 核都有自己的执行路径，不同的执行路径可能同时访问共享数据，因此存在真正的并发，并且其它 CPU 核的并发是不受本 CPU 核控制的，这时就需要通过下面介绍的锁机制来保护共享数据。

6.9.3 原子操作

如果共享数据只是一个变量，如整型数，则可以直接使用原子操作，而不需要重量级的加锁/解锁操作。内核保证原子操作的原子性，排它性，操作不会被打断，即使多个路径同时对变量执行操作也可保证顺序执行。

原子操作包括通用原子操作（原子变量）、本地原子操作和位掩码原子操作等。原子操作通常是由体系结构实现的。

1 原子变量

原子变量数据结构定义在/include/linux/types.h 头文件：

```
typedef struct {
    int counter;        /*整型数*/
} atomic_t;
```

对原子变量的操作不能直接进行，而需要调用内核提供的接口函数，以保证操作的原子性。原子变量操作的接口函数定义在/arch/mips/include/asm/atomic.h 头文件，例如：

●**ATOMIC_INIT(i)**：初始化原子变量值为 i。

●**atomic_read(v)**：读取原子变量值，v 为原子变量指针。

●**atomic_set(v, i)**：设置原子变量 v 的值为 i，v 为原子变量指针。

●**atomic_inc(v)**：v（指针）指向原子变量值加 1。

●**atomic_dec(v)**：v（指针）指向原子变量值减 1。

●**atomic_inc_and_test(v)**：v（指针）指向原子变量值加 1 后并测试是否为 0。

●**atomic_dec_and_test(v)**：v（指针）指向原子变量值减 1 后并测试是否为 0。

原子变量操作函数的实现是特定于体系结构的，请读者自行阅读相关代码，这里就不展开介绍了。

2 本地原子操作

除通用原子变量外，内核还定义了本地原子变量类型 local_t 和 local64_t。本地原子变量通常是某个 CPU 核的本地变量，允许所有 CPU 核读，但只允许本地 CPU 核写。

本地原子变量 local_t 类型定义如下（/arch/mips/include/asm/local.h）：

```
typedef struct
{
    atomic_long_t a;      /*原子变量*/
} local_t;
```

local_t 类型本地原子变量操作接口函数如下（举例，/arch/mips/include/asm/local.h）：

●**LOCAL_INIT(i)**：初始化值为 i。

●**local_read(l)**：读本地原子变量值（l 为指针，下同）。

●**local_set(l, i)**：设置本地原子变量值为 i。

●**local_add(i, l)**：本地原子变量值加 i。

●**local_sub(i, l)**：本地原子变量值减 i。

●**local_inc(l)**：本地原子变量值加 1。

●**local_dec(l)**：本地原子变量值减 1。

本地原子变量 local64_t 类型定义如下（/include/asm-generic/local64.h）：

```
typedef struct {
```

```
    atomic64_t a;
} local64_t;
```

local64_t 类型本地原子变量操作函数如下（举例，/include/asm-generic/local64.h）：

●**LOCAL64_INIT(i)**：设置初始值为 i。

●**local64_read(l)**：读本地原子变量值（l 为指针，下同）。

●**local64_set(l,i)**：设置本地原子变量值为 i。

●**local64_inc(l)**：本地原子变量值加 1。

●**local64_dec(l)**：本地原子变量值减 1。

●**local64_add(i,l)**：本地原子变量值加 i。

●**local64_sub(i,l)**：本地原子变量值减 i。

3 位掩码原子操作

位掩码操作是第三类原子操作，可以给定一个长整型类型的位掩码地址，原子性地设置、清除或改变指定位值，接口函数如下（/arch/mips/include/asm/bitops.h）：

●void **set_bit**(unsigned long nr, volatile unsigned long *addr)：置位（addr 地址中的 nr 比特位，下同）。

●void **clear_bit**(unsigned long nr, volatile unsigned long *addr)：清零。

●void **change_bit**(unsigned long nr, volatile unsigned long *addr)：转变比特位值。

●int **test_and_set_bit**(unsigned long nr,volatile unsigned long *addr)：置位并返回原值。

●int **test_and_clear_bit**(unsigned long nr,volatile unsigned long *addr)：清零并返回原值。

●int **test_and_change_bit**(unsigned long nr,volatile unsigned long *addr)：转变位值，并返回原值。

检测指定位值的函数如下（/include/asm-generic/bitops/non-atomic.h）：

●int **test_bit**(int nr, const volatile unsigned long *addr)：返回 addr 地址 nr 比特位的值。

6.9.4 自旋锁

如果共享数据只是一个变量，那么使用原子变量就可以解决问题。但是共享数据大多是某种数据结构，如双链表、红黑树等。

自旋锁 spinlock 是内核使用最广泛的同步原语。自旋锁同一时刻只能被一个内核代码路径持有，如果另一个内核代码路径试图获取一个已经被持有的自旋锁，那么此路径需要一直自旋忙等待，直到锁持有者释放了该锁。如果该锁没有被持有，那么可以立即获得该锁。

简单地说，自旋锁好比门锁，锁上只有一把钥匙，只允许一个人进入，共享数据位于房间里。第一个操作共享数据的路径拿钥匙开锁进入房间，把门反锁。当有其它路径到达时发现门被锁上且没有钥匙，于是在门口等待，直到第一个操作者完成操作，退出房间放回钥匙，门外等待的下一个路径才能持钥匙开锁进入房间，如此循环。

自旋锁具有以下基本特征：

（1）等待锁的过程是自旋的，不会睡眠和调度。

（2）持有自旋锁的临界区中不允许调度（含内核抢占）和睡眠，因为一旦发生调度，临界区什么时候能够继续运行是不确定的，这会导致其它竞争者死锁。因此，自旋锁的加锁操作会禁止内核抢占，解锁时恢复抢占。编写设备驱动程序时需特别注意，在临界区不能主动调度（睡眠），也不能调用可能引起睡眠的内核函数。

自旋锁主要用于多核处理器之间的并发控制，适用于锁竞争不太激烈的场景。如果锁竞争非常激烈，那么大量的时间会浪费在获取锁自旋上，导致整体性能下降。

在功能上，自旋锁分普通自旋锁和读写自旋锁两种。

1 普通自旋锁

普通自旋锁同一时刻只允许一个路径持有锁进入临界区，不区分读者和写者，只能有一个进入临界区。

■数据结构

自旋锁由 spinlock_t 结构体表示，定义在/include/linux/spinlock_types.h 头文件：

```
typedef struct spinlock {
    union {
        struct raw_spinlock    rlock;     /*原始自旋锁*/
        ...
    };
} spinlock_t;
```

spinlock_t 结构体中主要成员是 raw_spinlock 结构体，定义在/include/linux/spinlock_types.h 头文件：

```
typedef struct raw_spinlock {     /*原始自旋锁，相应的操作接口函数与 spinlock 类似*/
    arch_spinlock_t raw_lock;   /*体系结构自旋锁，由体系结构实现（SMP）或为空（UP）*/
    ...
} raw_spinlock_t;
```

arch_spinlock_t 结构体对于单核处理器和多核处理器具有不同的实现。

对于单核处理器，arch_spinlock_t 结构体为空（/include/linux/spinlock_types_up.h）。获取锁时只需禁止内核抢占，释放锁时恢复内核抢占即可。

对于多核处理器（SMP 选项），arch_spinlock_t 结构体是一个体系结构相关的数据结构，定义在头文件/arch/mips/include/asm/spinlock_types.h：

```
typedef union {     /*联合体*/
    u32 lock;       /*经典实现*/
    struct {         /*排队自旋锁实现*/
  #ifdef __BIG_ENDIAN     /*大端体系结构*/
        ...
  #else                          /*小端体系结构*/
        u16 serving_now;   /*当前持有锁进程编号，低 16 位，初始为 0*/
        u16 ticket;        /*分配给最近一个申请锁进程的编号，高 16 位，初始为 0*/
  #endif
    } h;
} arch_spinlock_t;
```

经典自旋锁 arch_spinlock_t 中只是一个无符号整数 lock，初始值为 0，持有锁的路径将其置 1，等待锁的路径不停地检测其值是否为 0，不为 0 则一直循环检测，持有锁的路径释放锁时将 lock 清 0，随后第

一个检测到 lock 为 0 的路径将会持有锁。经典自旋锁的一个问题就是如果有很多竞争者在竞争锁，谁将获得锁是随机的，就有可能导致有的竞争者饿死，永远竞争不到锁。

排队自旋锁保证了公平性，它给每个竞争锁的路径按顺序分配一张票据（ticket，编号）。ticket 成员值是最近分配的一个路径票据（下次分配前对其加 1）。当前持有锁的路径票据为 serving_now，释放锁时 serving_now 值加 1，随后只有票据值是 serving_now+1 的路径才能获得锁，以此实现排队的功能。

内核在/include/linux/spinlock_types.h 头文件实现了定义并初始化自旋锁的宏，主要是设置自旋锁成员值为 0：

```
#define DEFINE_SPINLOCK(x)  spinlock_t x = __SPIN_LOCK_UNLOCKED(x)

#define __SPIN_LOCK_UNLOCKED(lockname) \
     (spinlock_t ) __SPIN_LOCK_INITIALIZER(lockname)

#define __SPIN_LOCK_INITIALIZER(lockname) \
     { { .rlock = __RAW_SPIN_LOCK_INITIALIZER(lockname) } }    /*lock=0*/
```

spin_lock_init(_lock)函数定义在/include/linux/spinlock.h 头文件（_lock 为 spinlock_t 结构体指针），用于初始化已定义的自旋锁。

自旋锁操作主要是获取锁和释放锁。获取锁的接口函数为 spin_lock(spinlock_t *lock)，释放锁的接口函数为 spin_unlock(spinlock_t *lock)。

在单核处理器（UP）系统中，获取锁的 spin_lock()函数内主要是调用禁止 preempt_disable()函数内核抢占，再加编译器屏障 barrier()。释放锁的 spin_unlock()函数主要就是调用 preempt_enable()函数使能内核抢占。

下面主要讨论 SMP 系统中自旋锁的获取和释放操作。

■获取/释放锁

内核路径在进入临界区前需要调用 **spin_lock**(spinlock_t *lock)函数获取自旋锁，在操作完成后需要调用 **spin_unlock**(spinlock_t *lock)函数释放自旋锁，离开临界区。在临界区中不能调度，不能睡眠，但允许中断（包含处理软中断）。

●获取自旋锁

获取自旋锁的函数 spin_lock(spinlock_t *lock)定义在/include/linux/spinlock.h 头文件：

```
static inline void spin_lock(spinlock_t *lock)
{
     raw_spin_lock(&lock->rlock);     /*/include/linux/spinlock.h*/
}
#define raw_spin_lock(lock)  _raw_spin_lock(lock)       /*获取原始自旋锁*/
```

_raw_spin_lock(lock)函数定义成__raw_spin_lock(lock)函数（如果选择了 INLINE_SPIN_LOCK 选项），函数定义如下（/include/linux/spinlock_api_smp.h）：

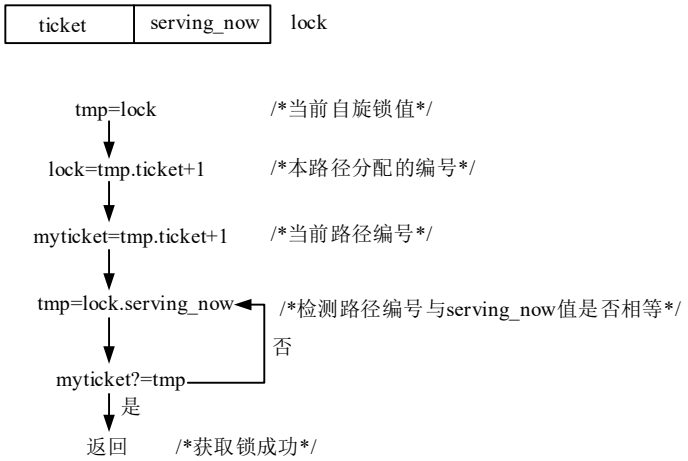
```
static inline void __raw_spin_lock(raw_spinlock_t *lock)
```

```
{
    preempt_disable();    /*禁止内核抢占*/
    spin_acquire(&lock->dep_map, 0, 0, _RET_IP_);  /*未选择 CONFIG_LOCKDEP 选项为空操作*/
    LOCK_CONTENDED(lock, do_raw_spin_trylock, do_raw_spin_lock);
            /*未选择 CONFIG_LOCK_STAT，调用 do_raw_spin_lock()函数，/include/linux/lockdep.h*/
}
```

在未选择 LOCK_STAT 配置选项时，LOCK_CONTENDED()只是调用 do_raw_spin_lock()函数，lock 为其参数。do_raw_spin_lock()函数定义如下（/include/linux/spinlock.h）：

```
static inline void do_raw_spin_lock(raw_spinlock_t *lock) __acquires(lock)
{
    __acquire(lock);
    arch_spin_lock(&lock->raw_lock);   /*/arch/mips/include/asm/spinlock.h*/
}
```

arch_spin_lock()函数是一个由体系结构实现的函数，由汇编代码实现，执行流程如下图所示：



arch_spin_lock()函数首先读取当前自旋锁 lock 成员值，对其 ticket 成员值加 1，作为当前路径的编号赋予 myticket，并写回 lock.ticket 成员。然后，读取 lock.serving_now 成员值，将其与 myticket 比较，如果相等则表示本路径是下一个持有锁的路径，函数返回，获取锁成功。如果不相等，则不断循环读取 lock.serving_now 成员值并与 myticket 比较，直至两者相等函数返回。

在获取自旋锁的操作中，如果当前路径暂时不能获取该锁，执行路径将在原地循环，不停地检测是否可获取自旋锁而不进入睡眠，这也是自旋锁名称的由来。执行路径在等待锁的过程中没有做实质性的工作，在空耗 CPU 时间，因此临界区代码应尽量短，尽快退出临界区释放自旋锁，以便其它路径获得自旋锁。

●释放自旋锁

释放自旋锁的函数 spin_unlock(spinlock_t *lock)定义如下（/include/linux/spinlock.h）：

```
static inline void spin_unlock(spinlock_t *lock)
{
    raw_spin_unlock(&lock->rlock);    /*/include/linux/spinlock.h*/
}
#define raw_spin_unlock(lock)    _raw_spin_unlock(lock)    /*释放原始自旋锁*/
```

_raw_spin_unlock()函数定义成__raw_spin_unlock()函数，定义如下（/include/linux/spinlock_api_smp.h）：

```
static inline void __raw_spin_unlock(raw_spinlock_t *lock)
{
    spin_release(&lock->dep_map, 1, _RET_IP_);
    do_raw_spin_unlock(lock);    /*/include/linux/spinlock.h*/
    preempt_enable();    /*使能内核抢占*/
}
```

do_raw_spin_unlock(lock)函数内调用 arch_spin_unlock(&lock->raw_lock)函数释放自旋锁：

```
static inline void arch_spin_unlock(arch_spinlock_t *lock)    /* /arch/mips/include/asm/spinlock.h*/
{
    unsigned int serving_now = lock->h.serving_now + 1;    /*下一个获得锁路径编号*/
    wmb();            /*写内存屏障*/
    lock->h.serving_now = (u16)serving_now;    /*下一个获得锁路径编号写入 arch_spinlock_t 实例*/
    nudge_writes();        /*内存屏障*/
}
```

释放自旋锁的操作比较简单，即将当前持有锁路径编号加 1，写入到 arch_spinlock_t 实例 serving_now 成员中，表示下一个获得锁路径的编号。下一个路径在获取锁的操作中检测到其编号与 serving_now 值相等时将会获得该锁。

内核还定义了自旋锁其它的操作接口函数，例如：

- int **spin_trylock**(spinlock_t *lock)：尝试获取锁，成功返回 1，否则返回 0，不自旋等待。
- int **spin_is_locked**(spinlock_t *lock)：锁是否已被持有（上锁），未被持有返回 0，已被持有返回 1。

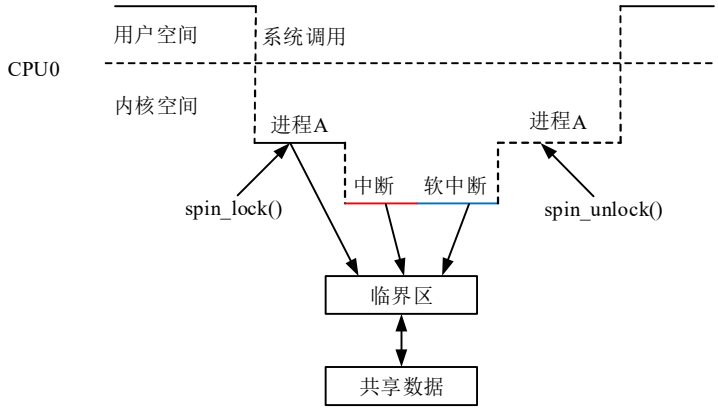
内核在/include/linux/spinlock.h 头文件还定义了原始自旋锁的操作接口函数，其实自旋锁、原始自旋锁之间是包含关系，相关函数代码请读者自行阅读。

当前主线内核还没有打上实时补丁（实时内核官网：https://rt.wiki.kernel.org/），自旋锁与原始自旋锁等价。原始自旋锁的语义是在临界区不可抢占，不可睡眠。如果打上了实时补丁，自旋锁与原始自旋锁语义就不同了，自旋锁在临界区就变成可抢占，可睡眠了，原始自旋锁语义保持不变。

因此为了适用将来实时补丁的加入，在绝对不允许抢占和睡眠的临界区应使用原始自旋锁，其它地方用自旋锁。

■自旋锁变种

上面介绍的自旋锁解决了多处理器核之间的并发问题，使之串行化。但是没有解决本 CPU 核由于中断而导致的并发问题（交错执行问题）。如下图所示，假设进程 A 在持有自旋锁进入临界区后，在临界区中发生了中断，如果在中断处理函数或软中断中也要获取自旋锁，对相同的共享数据进行操作。此时，锁被进程 A 持有，中断处理函数或软中断将一直自旋，无法获得锁（进程 A 也无法继续运行，无法释放锁），造成死锁问题。



为了解决以上问题，需要结合自旋锁和开关中断（软中断），如果进程访问的共享数据会在中断处理函数中被访问则需要在临界区关闭中断（同时禁止了软中断），如果共享数据不会被中断处理函数访问，但是会被软中断访问，则在临界区需要禁止软中断。

普通自旋锁和开关中断（软中断）结合的接口函数如下（/include/linux/spinlock.h）：

●**spin_lock_irq**(spinlock_t *lock)：获取自旋锁并关闭本地中断。

●**spin_unlock_irq**(spinlock_t *lock)：释放自旋锁并开启本地中断。

●spin_lock_irqsave(lock, flags)：获取自旋锁、关闭本地中断并将中断状态保存至 flags 参数。

●spin_unlock_irqrestore(spinlock_t *lock, unsigned long flags)：释放自旋锁、并从 flags 参数中恢复中断状态。

●**spin_lock_bh**(spinlock_t *lock)：获取自旋锁并禁止软中断。

●**spin_unlock_bh**(spinlock_t *lock)：释放自旋锁并使能软中断。

在中断/软中断中可调用 spin_lock(spinlock_t *lock)和 spin_unlock(spinlock_t *lock)函数获取和释放锁。

2 读写自旋锁

普通自旋锁并没有区分路径对共享数据的访问是读还是写，对于读操作来说可以允许多个路径同时进行，而对于写操作来说应当是排它性的。

读写自旋锁是对普通自旋锁的改进，区分读者和写者，允许多个读者同时持有读锁，进入临界区读取共享数据，但写者需独占地持有锁（写者持有锁时其它写者和读者不能持有锁）。读写自旋锁适用于读多写少的应用场景。

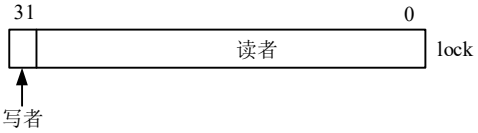
读写自旋锁由 rwlock_t 结构体表示，结构体定义在/include/linux/rwlock_types.h 头文件：

```
typedef struct {
    arch_rwlock_t raw_lock;  /*体系结构定义，/arch/mips/include/asm/spinlock_types.h*/
    ...
} rwlock_t;
```

rwlock_t 结构体中主要包含 arch_rwlock_t 结构体成员，定义在/arch/mips/include/asm/spinlock_types.h 头文件：

```
typedef struct {
    volatile unsigned int lock;
} arch_rwlock_t;
```

arch_rwlock_t 读写自旋锁包含一个无符号整型数成员，布局如下：



写者使用最高位，因为某一时刻只能有一个写者，因此一个比特位就够了，其余位用于读者。获取/释放锁的操作如下：

（1）申请写锁时，如果 lock 为 0，那么置位最高位，进入临界区。lock 不为 0，自旋等待。

（2）释放写锁时，lock 清 0。

（3）申请读锁时，如果最高位是 0，则 lock 值加 1，进入临界区，如果最高位不为 0，自旋等待。

（4）释放读锁时，lock 值减 1。

arch_rwlock_t 结构体中 lock 成员值与读写锁状态如下表所示：

读写锁操作	条件（lock）	lock 值变化
申请读者锁	大于等于 0（最高位为 0）	加 1
释放读者锁	---	减 1
申请写者锁	等于 0	0x8000_0000
释放写者锁	---	0x0

内核定义和初始化 rwlock_t 结构体实例的宏如下（/include/linux/rwlock_types.h）：

```
#define DEFINE_RWLOCK(x)    rwlock_t x = __RW_LOCK_UNLOCKED(x)

#define __RW_LOCK_UNLOCKED(lockname) \
    (rwlock_t)    {    .raw_lock = __ARCH_RW_LOCK_UNLOCKED,  \
                  RW_DEP_MAP_INIT(lockname) }
```

__ARCH_RW_LOCK_UNLOCKED 宏初始化 raw_lock 成员 lock 成员值为 0。

读写自旋锁操作函数声明在/include/linux/rwlock.h 头文件，例如（源代码类似普通自旋锁）：

●**rwlock_init**(lock)：初始化已定义读写自旋锁。

●**read_lock**(lock)：申请读者锁。

●**read_unlock**(lock)：释放读者锁。

●**read_trylock(lock)**：尝试获取读者自旋锁，成功返回 1，不成功返回 0，不等待。

●**write_lock**(lock)：申请写者锁。

●**write_unlock**(lock)：释放写者锁。

●**write_trylock(lock)**：尝试获取写者自旋锁，成功返回 1，不成功返回 0，不等待。

●**read_lock_irq**(lock)：申请读者锁并关中断。

●**read_unlock_irq**(lock)：释放读者锁并开中断

●**read_lock_bh**(lock)：申请读者锁并禁止软中断。

●**read_unlock_bh**(lock)：释放读者锁并使能软中断。

●**write_lock_irq**(lock)：申请写者锁并关中断。

●**write_unlock_irq**(lock）：释放写者锁并开中断。

●**write_lock_bh**(lock)：申请写者锁并禁止软中断。

●**write_unlock_bh**(lock)：释放写者锁并使能软中断。

读写自旋锁的操作与普通自旋锁类似，只在底层实现上有所区别（/arch/mips/include/asm/spinlock.h），请读者自行阅读源代码。

读写自旋锁的缺点是：如果读者很多，写者很难获取写锁，可能饿死。针对这个缺点，内核实现了排队读写锁，主要改进是：如果写者正在等待写锁，那么读者申请读锁时自旋等待，写者在锁被释放以后先得到写锁。排队读写锁的配置选项为 QUEUED_RWLOCKS，源文件是/kernel/locking/qrwlock.c。

6.9.5 顺序锁

读写自旋锁是对普通自旋锁的改进，它对锁的竞争者进行了区分，允许多个读者同时进入临界区。但是读写自旋锁没有区分读者和写者的重要性（优先权），因此带来一个问题：大量读者持有读锁的时候，写者可能需要等待很长的时间，直到所有读者都退出临界区以后才能得到写锁。

顺序锁就是为了解决这个问题而设计的，它同样适合读者多而写者少的场景，但是赋予写者比读者高的优先权。

顺序锁读者不会阻塞写者，也就是说读者和写者可同时操作共享数据（写者之间还是互斥的）。顺序锁中包含一个顺序号，写者在进入和退出临界区时都会将顺序号加 1。读者在操作共享数据前读取顺序号，然后对共享数据执行读操作，然后再次读取顺序号与读操作前读取的顺序号进行比较，如果相等说明读操作过程中没有写者操作过共享数据，读操作成功，如果不相等，则重新执行读操作。

顺序锁支持两种类型的读者（读操作方法）：

（1）顺序读者：不会阻塞写者，在读操作前检测顺序号，读操作完成后再次检测顺序号，如果相等，读操作成功，不相等则再次执行读操作。多个读者可以同时进入临界区。

（2）持锁读者：如果写者或另一个持锁读者正位于临界区，持锁读者将等待，持锁读者也会阻塞写者，此时顺序锁与自旋锁功能相同。

如果使用顺序读者，那么互斥访问的资源不能是指针，因为写者可能使指针无效，读者访问无效的指针会出现致命的错误。读者和写者在不能获取顺序锁时与自旋锁一样，在原地等待，不睡眠。

顺序锁有两个版本：（1）完整版的顺序锁，带自旋锁和顺序号，（2）只提供顺序号的顺序锁，自旋锁由调用者提供。

顺序锁数据结构及接口函数定义在/include/linux/seqlock.h 头文件内。

1 完整版顺序锁

完整版顺序锁数据结构定义如下：

```
typedef struct {
    struct seqcount seqcount;      /*包含顺序号成员*/
    spinlock_t lock;          /*保护 seqcount 成员的自旋锁*/
} seqlock_t;
```

seqcount 成员为 seqcount 结构体，定义如下（只含顺序号的顺序锁）：

```
typedef struct seqcount {
```

```
    unsigned sequence;     /*顺序号，无符号整数*/
    ...
} seqcount_t;
```

内核定义并初始化顺序锁的宏定义如下：

```
#define DEFINE_SEQLOCK(x) \
        seqlock_t x = __SEQLOCK_UNLOCKED(x)

#define __SEQLOCK_UNLOCKED(lockname)          \
    {                                  \
        .seqcount = SEQCNT_ZERO(lockname),    \        /*顺序号设为 0*/
        .lock =    __SPIN_LOCK_UNLOCKED(lockname)   \        /*初始化自旋锁*/
    }
```

初始化已定义顺序锁实例的函数为 **seqlock_init(x)**，函数内将顺序号设为 0，并初始化保护自旋锁。

■顺序读操作

顺序读者读数据的操作方法如下：

```
seqlock_t seqlock;        /*顺序锁*/
unsigned int seq;         /*暂存顺序号*/

do {
    seq = read_seqbegin(&seqlock);       /*读取顺序号至 seq 变量*/
    ...                  /*对共享数据执行读操作*/
} while (read_seqretry(&seqlock, seq));   /*将当前顺序号与原读取的顺序号 seq 比较*/
```

read_seqbegin(&seqlock)函数读取顺序号，且直到读取到偶数值。因为写者在进/出临界区都会对顺序号加 1，读取偶数值以保证写者操作完成。

read_seqretry(&seqlock, seq)函数将当前的顺序号与 seq 中保存的顺序号比较，如果相等返回 0，不相等返回 1。

由以上函数可知，顺序读者在操作共享数据前先读取顺序号，然后对共享数据执行读操作，操作完成后比较当前顺序号是否与之前读取的顺序号相等，相等则读操作成功，不相等则需要重新执行读操作（写者操作了共享数据）。

■持锁读操作

持锁读者读操作方法如下：

```
seqlock_t seqlock;        /*定义顺序锁，还需初始化*/

read_seqlock_excl(seqlock_t *seqlock)
...                  /*读共享数据*/
read_sequnlock_excl(seqlock_t *seqlock)
```

read_seqlock_excl()和 read_sequnlock_excl()函数直接就是对 seqlock->lock 自旋锁的获取和释放，与顺序号无关。

持锁读者读操作与自旋锁相似，也提供了其它的一些变体的接口函数，例如：

●void read_seqlock_excl_irq(seqlock_t *sl)：申请读锁，并关本地中断。

●void read_sequnlock_excl_irq(seqlock_t *sl)：释放锁，并开本地中断。

●void read_seqlock_excl_bh(seqlock_t *sl)：申请读锁，并禁止软中断。

●void read_sequnlock_excl_bh(seqlock_t *sl)：释放锁，并使能软中断。

另外，读者可以根据情况选择顺序读还是持锁读，如果没有写者则采用顺序读方法，如果有写者则采用持锁读方法，如下所示。

```
seqlock_t seqlock;        /*顺序锁，还需初始化*/
unsigned int seq=0;         /*暂存顺序号*/

do {
   read_seqbegin_or_lock(&seqlock, &seq);      /*顺序号为偶数，顺序读；奇数，持锁读*/
   ...       /*对共享数据执行读操作*/
} while (need_seqretry(&seqlock, int seq));   /*顺序号是偶数，且有变化*/

done_seqretry(&seqlock, seq);   /*顺序号是奇数，释放自旋锁（持锁读）*/
```

read_seqbegin_or_lock()函数判断当前顺序号是否为偶数，如果为偶数（没有写者）则将顺序号读入 seq 变量，执行顺序读。如果为奇数（有写者），则获取 seqlock->lock 自旋锁，执行持锁读。

在操作完共享数据后判断顺序号是否为偶数且有变化，如果是则重新执行读操作，否则读操作成功。最后，如果原顺序号 seq 是奇数（持锁读）则在 done_seqretry()函数中释放 seqlock->lock 自旋锁。

■写操作

写者申请/释放顺序锁的操作如下，写者与写者之间是互斥的需要持有自旋锁：

```
seqlock_t seqlock;   /*顺序锁，还需初始化*/

write_seqlock(&seqlock);
...          /*写共享数据*/
write_sequnlock(&seqlock);
```

write_seqlock()函数在持有 seqlock->lock 自旋锁情况下对顺序号加 1，使其为奇数，表示有写者在操作共享数据。

write_sequnlock()函数对顺序号加 1，使其为偶数，表示写者退出临界区，然后释放 seqlock->lock 自旋锁。

同样写者持有和释放顺序锁的接口函数也存在变体，例如：

●void write_seqlock_irq(seqlock_t *sl)：申请写锁，并禁止本地中断。

●void write_sequnlock_irq(seqlock_t *sl)：释放写锁，并打开本地中断。

●void write_seqlock_bh(seqlock_t *sl)：申请写锁，并禁止软中断。
●void write_sequnlock_bh(seqlock_t *sl)：释放写锁，并使能软中断。

2 只含顺序号的顺序锁

只含顺序号的顺序锁就是由 seqcount 结构体实例表示的顺序锁：

```
typedef struct seqcount {
    unsigned sequence;    /*顺序号，无符号整数*/
    ...
} seqcount_t;
```

初始化 seqcount 实例的函数为 seqcount_init(s)，s 为指针，初始化函数将顺序号置 0。
SEQCNT_ZERO(lockname)宏用于定义顺序锁时初始化（赋初值），顺序号置 0，lockname 表示顺序锁名称。

seqcount_t 结构体中不含自旋锁，读操作中不需要持有自旋锁，写操作中调用者需要自定义自旋锁，并在持有锁的情况下执行写操作。

读者读共享数据的方法如下：

```
seqcount_t seq;      /*还需初始化*/
unsigned s;

do {
    s = read_seqcount_begin(&seq);      /*读顺序号（直至读到偶数）*/
    ...    /*读共享数据*/
} while (read_seqcount_retry(&seq, s)); /*比较当前顺序号与 s 值，不相等则重新读，相等读操作成功*/
```

写者写共享数据的方法如下：

```
spinlock_t lock;      /*自定义自旋锁，需初始化*/
seqcount_t seqcount;    /*只含顺序号的顺序锁，需初始化*/

spin_lock(&lock);    /*申请自旋锁*/
write_seqcount_begin(&seqcount);    /*顺序号加 1*/
...          /*写共享数据*/
write_seqcount_end(&seqcount);    /*顺序号加 1*/
spin_unlock(&lock);    /*释放自旋锁*/
```

6.9.6 信号量

这里说的信号量与第 5 章介绍的进程间通信的信号量是不同的概念。进程间通信的信号量由用户进程

使用，用于保护用户空间线程间共享的数据，而这里的信号量由内核使用，用于保护内核空间的共享数据。

前面介绍的自旋锁和顺序锁，进程在等待锁释放时处于忙等待状态，不进入睡眠，空耗 CPU 时间。信号量工作原理与自旋锁最大的不同在于获取锁的过程中，若不能立即得到锁，就会发生调度，转入睡眠。其它持有信号量的内核路径释放信号量时会唤醒睡眠等待的进程。

信号量适用于保护比较长的临界区。由于信号量会使进程进入睡眠，因此不能在中断上下文（包括硬中断和软中断）中使用。

使用信号量最经典的例子就是操作系统中生产者和消费者之间的关系。信号量是一个计数器，增加减/少计数值的进程相当于生产者/消费者，而信号量本身相当于销售的商店。生产者生产出产品将其放入商店，信号量计数值（商品数量）增加，并唤醒等待商品的消费者（如果有的话）。消费者从商店获取商品，信号量计数值减小，当计数值为 0 时，消费者进入睡眠，等待生产者生产出商品。

内核代码中使用信号量保护共享数据时，在进入临界区前需要获取信号量，如果信号量值大于 0 则将其值减 1，获得信号量，进入临界区。如果信号量值已经为 0，则进程在信号量上睡眠等待。获得信号量进程在离开临界区时，需要释放信号量，即将信号量值加 1，并唤醒第一个在信号量上睡眠等待的进程（如果有的话）。

信号量保护的共享数据可同时有多个路径访问，这取决于信号量初始化时赋的初值。当信号量只有两个值时（0 和 1），为二值信号量，其保护的临界区同时只能有一个路径进入。信号量值为 1 时可以获取信号量，进入临界区（信号量值减 1），为 0 时需等待。二值信号量又称为互斥信号量。

内核还实现了读写信号量，将对共享数据的操作区分读者和写者，允许多个读者同时访问共享数据（获得信号量），但只允许一个写者独占地访问共享数据（获得信号量）。信号量值为 0 表示信号量没有读者和写者，大于 0 表示有一个或多个读者，-1 表示有一个写者持有信号量。

信号量保护的临界区允许中断和内核抢占。普通信号量实现代码位于/kernel/locking/semaphore.c 文件内，读写信号量实现代码位于/kernel/locking/rwsem-spinlock.c 文件内（普通的实现方式）。

1 普通信号量

普通信号量由 semaphore 结构体表示，结构体定义在/include/linux/semaphore.h 头文件：

```
struct semaphore {
    raw_spinlock_t      lock;       /*原始自旋锁，保护结构体中其它成员*/
    unsigned int        count;      /*信号量值*/
    struct list_head    wait_list;  /*等待信号量的等待进程双链表*/
};
```

semaphore 结构体主要成员简介如下：

●**lock**：保护 semaphore 结构体中其它成员的原始自旋锁。

●**count**：表示允许进入临界区的内核执行路径数量，信号量值。

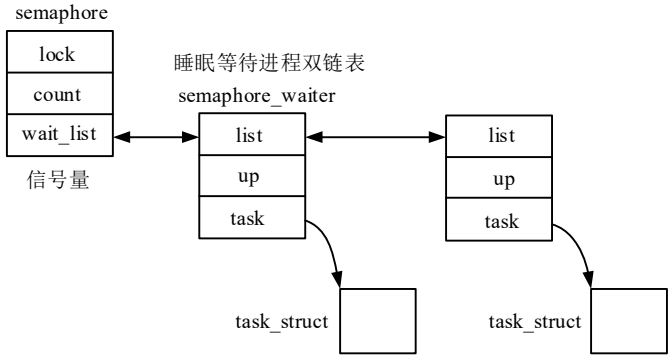
●**wait_list**：双链表成员，管理在该信号量上睡眠等待的进程（down 操作引起睡眠）。双链表链接的对象是 semaphore_waiter 结构体实例。

semaphore_waiter 结构体定义在/kernel/locking/semaphore.c 文件内：

```
struct semaphore_waiter {
    struct list_head list;          /*双链表成员，添加到信号量中 wait_list 双链表*/
    struct task_struct *task;       /*指向睡眠 task_struct 实例*/
    bool up;                        /*是否被唤醒*/
```

};

信号量数据结构组织关系如下图所示：



内核在/include/linux/semaphore.h 头文件定义了创建并初始化信号量的宏（互斥信号量）：

```
#define    DEFINE_SEMAPHORE(name) \     /*创建并初始化信号量，值为 1*/
    struct semaphore name = __SEMAPHORE_INITIALIZER(name, 1)

#define __SEMAPHORE_INITIALIZER(name, n)                    \
{                                                   \
    .lock       = __RAW_SPIN_LOCK_UNLOCKED((name).lock), \
    .count          = n,                                \
    .wait_list = LIST_HEAD_INIT((name).wait_list),          \
}
```

sema_init(struct semaphore *sem, int val)函数用于初始化一个已经定义的信号量，信号量值设为 **val**。

内核在/include/linux/semaphore.h 头文件声明了信号量操作的接口函数，下面主要介绍一下其中的 **down()**和 **up()**操作函数。

■down 操作

内核路径在进入信号量保护的临界区时，需要调用 down()函数获取信号量，函数定义如下：

```
void down(struct semaphore *sem)       /*/kernel/locking/semaphore.c*/
{
    unsigned long flags;

    raw_spin_lock_irqsave(&sem->lock, flags);  /*获取自旋锁，关闭并保存本地中断状态*/
    if (likely(sem->count > 0))        /*如果 count 大于 0，直接 count 值减 1 即可*/
        sem->count--;
    else                /*如果 count 等于 0，进程需要在信号量上睡眠*/
        __down(sem);     /*/kernel/locking/semaphore.c*/
    raw_spin_unlock_irqrestore(&sem->lock, flags);    /*释放自旋锁，恢复原本地中断状态*/
}
```

如果信号量 count 值大于 0，则直接 count 值减 1，函数即可返回，表示调用者可进入临界区。如果

count 值等于 0，则调用__down(sem)函数使当前进程在信号量上睡眠等待，等待 up 操作中将其唤醒。

__down(sem)函数定义如下：

```
static noinline void __sched __down(struct semaphore *sem)
{
    __down_common(sem, TASK_UNINTERRUPTIBLE, MAX_SCHEDULE_TIMEOUT);
                                /*进入不可中断睡眠状态，睡眠时间无限长，进程不能由信号唤醒*/
}
```

__down_common()函数定义如下：

```
static inline int __sched __down_common(struct semaphore *sem, long state,long timeout)
{
    struct task_struct *task = current;
    struct semaphore_waiter waiter;       /*semaphore_waiter 结构体实例*/

    list_add_tail(&waiter.list, &sem->wait_list);     /*添加到信号量睡眠进程双链表末尾*/
    waiter.task = task;
    waiter.up = false;     /*进程睡眠，没有唤醒*/

    for (;;) {
        if (signal_pending_state(state, task))       /*如果可被信号唤醒，且有挂起信号*/
            goto interrupted;
        if (unlikely(timeout <= 0))
            goto timed_out;
        __set_task_state(task, state);     /*设置进程状态，不可中断睡眠*/
        raw_spin_unlock_irq(&sem->lock);              /*释放原始自旋锁，开中断*/
        timeout = schedule_timeout(timeout);     /*进程调度，当前进程从就绪队列移除*/
        raw_spin_lock_irq(&sem->lock);              /*获取原始自旋锁，关中断*/
        if (waiter.up)     /*进程被 up 操作唤醒，函数返回 0*/
            return 0;
    }

    /*进程超时，此处超时时间为 MAX_SCHEDULE_TIMEOUT，无限延时*/
 timed_out:
    list_del(&waiter.list);   /*从等待双链表删除，返回错误码*/
    return -ETIME;

 interrupted:        /*如果进程是由挂起信号唤醒，返回错误码*/
    list_del(&waiter.list);
    return -EINTR;
}
```

__down_common()函数比较好理解，函数内定义 semaphore_waiter 实例，并插入到信号量等待进程双链表，进程状态设为不可中断睡眠状态，执行进程调度。进程被唤醒后，__down_common()函数返回 0，表示进程可以获得信号量，进入临界区。down()函数最后会释放自旋锁，恢复本地中断状态。

down 操作的其它几个变体函数如下：

●int **down_interruptible**(struct semaphore *sem)：获取信号量，信号量值为 0 时，进入可中断睡眠状态，获取信号量成功后返回 0，由信号唤醒返回错误码。

●int **down_killable**(struct semaphore *sem)：获取信号量，信号量值为 0 时，进入中度睡眠，可被致命信号唤醒（返回错误码），获取信号量成功返回 0。

●int **down_timeout**(struct semaphore *sem, long timeout)：获取信号量，信号量值为 0 时，进入不可中断睡眠状态，超时将被唤醒，返回错误码，获取信号量成功，返回 0。参数 timeout 为内核节拍计数值。

●int **down_trylock**(struct semaphore *sem)：尝试获取信号量，信号量值大于 0 时，获取信号量成功，返回 0；信号量值为 0 时，失败，不等待，返回 1。

■up 操作

内核路径在离开临界区时，调用 up()函数增加信号量值，函数定义如下（/kernel/locking/semaphore.c）：

```
void up(struct semaphore *sem)
{
    unsigned long flags;

    raw_spin_lock_irqsave(&sem->lock, flags);
    if (likely(list_empty(&sem->wait_list)))     /*如果睡眠等待进程双链表为空，增加 count 值即可*/
        sem->count++;
    else        /*睡眠等待进程双链表不为空，唤醒睡眠的进程（只唤醒第一个进程）*/
        __up(sem);        /*/kernel/locking/semaphore.c*/
    raw_spin_unlock_irqrestore(&sem->lock, flags);
}
```

__up()函数用于唤醒等待进程双链表中第一个进程，函数定义如下：

```
static noinline void __sched __up(struct semaphore *sem)
{
    struct semaphore_waiter *waiter = list_first_entry(&sem->wait_list,struct semaphore_waiter, list);
    list_del(&waiter->list);     /*移除双链表中第一个 semaphore_waiter 实例*/
    waiter->up = true;        /*设置唤醒标记*/
    wake_up_process(waiter->task);     /*唤醒睡眠等待进程*/
}
```

up()操作检查睡眠等待进程双链表是否为空，如果为空则直接 count 值加 1 即可。如果不为空，则唤醒等待双链表中第一个进程。

2 读写信号量

读写信号量与前面介绍的读写自旋锁相似，允许多个读路径同时进入临界区，但是只允许一个写路径独占地进入临界区。

读写信号量有两种实现方式。如果内核配置选择了 RWSEM_GENERIC_SPINLOCK 选项（MIPS 默认选择），则读写信号量采用普通的实现方式（/include/linux/rwsem-spinlock.h），否则采用含 MCS 锁的实

现方式（/include/linux/rwsem.h）。这里我们只关注普通读写信号量的实现，这也是 MIPS 默认的实现方式。

读写信号量由 rw_semaphore 结构体表示（/include/linux/rwsem-spinlock.h）：

```
struct rw_semaphore {
    __s32                   count;      /*信号量值*/
    raw_spinlock_t          wait_lock;  /*保护结构体中其它成员的原始自旋锁*/
    struct list_head        wait_list;  /*在信号量等待的睡眠进程双链表*/
    ...
};
```

rw_semaphore 结构体主要成员简介如下：

●**count**：信号量值，取值含义如下：

取值	读者/写者数量
0	没有读者和写者
大于 0	有一个或多个读者
-1	有一个写者

●**wait_lock**：保护结构体中其它成员的原始自旋锁。

●**wait_list**：双链表成员，链接等待信号量的进程，链表成员为 rwsem_waiter 实例。申请读者信号量时，如果 count 值小于 0（表示有写者持有锁）或等待队列不为空，则读者进程进入睡眠等待，否则 count 值加 1，申请者获取读信号量。申请写者信号量时，如果 count 不为 0，则写者进程进入睡眠等待，如果 count 值为 0（没有写者和读者）则将其置为-1，获取写者信号量。

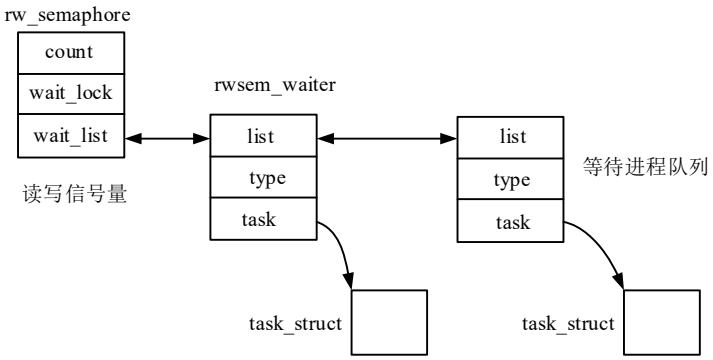
信号量等待进程双链表成员为 rwsem_waiter 结构体实例，定义在/kernel/locking/rwsem-spinlock.c 文件内：

```
struct rwsem_waiter {
    struct list_head list;          /*双链表成员*/
    struct task_struct *task;       /*指向等待进程 task_struct 实例*/
    enum rwsem_waiter_type type;    /*等待进程类型*/
};
```

等待进程类型由 rwsem_waiter_type 枚举类型表示：

```
enum rwsem_waiter_type {
    RWSEM_WAITING_FOR_WRITE,    /*写者进程*/
    RWSEM_WAITING_FOR_READ      /*读者进程*/
};
```

读写信号量数据结构组织关系与普通信号量相似，如下图所示：



内核在/include/linux/rwsem.h 头文件定义了声明和初始化读写信号量的宏：

```
#define DECLARE_RWSEM(name) \
    struct rw_semaphore name = __RWSEM_INITIALIZER(name)

#define __RWSEM_INITIALIZER(name)                \
    { .count = RWSEM_UNLOCKED_VALUE,             \    /*初值设为 0x0*/
      .wait_list = LIST_HEAD_INIT((name).wait_list), \
      .wait_lock = __RAW_SPIN_LOCK_UNLOCKED(name.wait_lock)   \
      __RWSEM_OPT_INIT(name)                     \
      __RWSEM_DEP_MAP_INIT(name) }
```

初始化已定义读写信号量的函数为 init_rwsem(sem)，主要是将信号量 count 值初始化为 0，并初始化双链表成员和自旋锁成员。

■读者信号量操作

读共享数据的路径获取信号量时，如果信号量的值大于等于 0 且没有等待进程（没有写者在等待），则信号量 count 值加 1 后获取读者信号量，函数返回，进入临界区；否则读者进入睡眠状态，添加到信号量睡眠等待队列双链表。

●获取信号量

获取读者信号量的操作函数为 **down_read**(struct rw_semaphore *sem)，定义在/kernel/locking/rwsem.c 文件内：

```
void __sched down_read(struct rw_semaphore *sem)
{
    might_sleep();
    rwsem_acquire_read(&sem->dep_map, 0, 0, _RET_IP_);

    LOCK_CONTENDED(sem, __down_read_trylock, __down_read);
}
```

down_read()函数内将调用**__down_read()**函数，函数定义在/kernel/locking/rwsem-spinlock.c 文件内：

```
void __sched __down_read(struct rw_semaphore *sem)
{
```

```
    struct rwsem_waiter waiter;     /*rwsem_waiter 实例*/
    struct task_struct *tsk;
    unsigned long flags;

    raw_spin_lock_irqsave(&sem->wait_lock, flags);      /*获取自旋锁，保存当前中断状态，关中断*/
    if (sem->count >= 0 && list_empty(&sem->wait_list)) {   /*count>=0 且没有等待进程*/
        sem->count++;           /*信号量值加 1 即可返回，获取读者信号量成功*/
        raw_spin_unlock_irqrestore(&sem->wait_lock, flags);    /*释放自旋锁，恢复中断状态*/
        goto out;
    }

    /*count<0（写者获得信号量）或有等待进程（有写者进程在等待）*/
    tsk = current;
    set_task_state(tsk, TASK_UNINTERRUPTIBLE);   /*当前进程设为不可中断睡眠状态*/

    waiter.task = tsk;
    waiter.type = RWSEM_WAITING_FOR_READ;     /*读者进程*/
    get_task_struct(tsk);

    list_add_tail(&waiter.list, &sem->wait_list);     /*当前进程添加到等待队列双链表末尾*/

    raw_spin_unlock_irqrestore(&sem->wait_lock, flags);  /*释放自旋锁，恢复中断状态*/

    for (;;) {      /*等待释放信号量*/
        if (!waiter.task)   /*当前进程被 up_write()函数唤醒时，waiter.task 赋为 NULL*/
            break;
        schedule();      /*进程调度*/
        set_task_state(tsk, TASK_UNINTERRUPTIBLE);       /*设为不可中断睡眠状态*/
    }
    __set_task_state(tsk, TASK_RUNNING);     /*当前进程被唤醒后，设置进程状态为可运行*/
 out:
    ;
}
```

当前路径若能获取读者信号量，直接将 count 值加 1 即可。否则，进入不可中断睡眠状态，添加到等待队列。

进程进入睡眠时 waiter.task 成员指向进程 task_struct 实例，在写者增加信号量的 up_write()操作中唤醒进程时会将 waiter.task 成员设为 NULL。读进程被唤醒后，设为可运行状态，down_read()函数返回，表示获取读信号量成功，可进入临界区。

●释放信号量

读者释放信号量的操作由 **up_read()**函数实现，函数内将 count 值减 1，如果减 1 后 count 值不为 0，说明还有其它的读者，up_read()函数返回。如果 count 值减 1 后为 0，则唤醒等待进程双链表中第一个进

程。

第一个等待进程必定为写者进程，因为如果当前是读者持有信号量（写者没有持有），再有读者申请读操作时，将申请成功，不会进入睡眠。只有写者申请信号量时才会申请不成功，进入睡眠等待，而其之后的读、写者申请信号量都将进入睡眠等待，排在第一个写者之后。

读者释放信号量的操作函数 **up_read**(struct rw_semaphore *sem)定义如下（/kernel/locking/rwsem.c）：

```
void up_read(struct rw_semaphore *sem)
{
    rwsem_release(&sem->dep_map, 1, _RET_IP_);
    __up_read(sem);     /*/kernel/locking/rwsem-spinlock.c*/
}
```

__up_read(sem)函数定义如下：

```
void    __up_read(struct rw_semaphore *sem)
{
    unsigned long flags;

    raw_spin_lock_irqsave(&sem->wait_lock, flags);
    if (--sem->count == 0 && !list_empty(&sem->wait_list))     /*count 减 1 后为 0，且有等待进程*/
        sem = __rwsem_wake_one_writer(sem);
                                /*唤醒第一个写者等待进程，/kernel/locking/rwsem-spinlock.c*/
    raw_spin_unlock_irqrestore(&sem->wait_lock, flags);
}
```

__rwsem_wake_one_writer(sem)函数用于唤醒第一个等待信号量的进程（第一个进程必定为写者进程），函数定义如下：

```
static inline struct rw_semaphore *__rwsem_wake_one_writer(struct rw_semaphore *sem)
{
    struct rwsem_waiter *waiter;

    waiter = list_entry(sem->wait_list.next, struct rwsem_waiter, list);     /*第一个等待进程*/
    wake_up_process(waiter->task);      /*唤醒进程*/
    return sem;
}
```

■写者信号量操作

写者申请读写信号量时，只有 count 值为 0 时才能直接申请成功，否则进入睡眠等待，申请成功后，写者将 count 值设为-1。写者释放信号量时，唤醒第一个睡眠等待的进程（如果是写进程）或双链表中第一个写者进程之前的所有读者进程。

●获取信号量

写者获取信号量的函数为 **down_write()**，定义在/kernel/locking/rwsem.c 文件内，代码如下：

```
void __sched down_write(struct rw_semaphore *sem)
{
```

```
    might_sleep();
    rwsem_acquire(&sem->dep_map, 0, 0, _RET_IP_);

    LOCK_CONTENDED(sem, __down_write_trylock, __down_write);   /*调用__down_write()函数*/
    rwsem_set_owner(sem);
}
```

__down_write()函数定义如下（/kernel/locking/rwsem-spinlock.c）：

```
void __sched __down_write(struct rw_semaphore *sem)
{
    __down_write_nested(sem, 0);     /*/kernel/locking/rwsem-spinlock.c*/
}
```

__down_write_nested()函数代码如下：

```
void __sched    __down_write_nested(struct rw_semaphore *sem, int subclass)
{
    struct rwsem_waiter waiter;   /*睡眠等待双链表成员*/
    struct task_struct *tsk;
    unsigned long flags;

    raw_spin_lock_irqsave(&sem->wait_lock, flags);

    tsk = current;     /*初始化 rwsem_waiter 实例，并插入信号量等待双链表末尾*/
    waiter.task = tsk;
    waiter.type = RWSEM_WAITING_FOR_WRITE;   /*写者进程*/
    list_add_tail(&waiter.list, &sem->wait_list);     /*插入双链表末尾*/

    for (;;) {
        if (sem->count == 0)     /*count 为 0 表示可获取写者信号量，跳出循环*/
            break;

        /*不能获取写者信号量，进程加入信号量睡眠等待队列双链表*/
        set_task_state(tsk, TASK_UNINTERRUPTIBLE);     /*设为不可中断睡眠状态*/
        raw_spin_unlock_irqrestore(&sem->wait_lock, flags);
        schedule();      /*进程调度*/
        raw_spin_lock_irqsave(&sem->wait_lock, flags);   /*唤醒后再尝试获取信号量*/
    }

     /*被唤醒并获取写者信号量成功*/
    sem->count = -1;          /*count 设为-1*/
    list_del(&waiter.list);   /*rwsem_waiter 实例从等待双链表中移除*/
    raw_spin_unlock_irqrestore(&sem->wait_lock, flags);
}
```

●**释放信号量**

写者释放信号量的操作函数为 **up_write()**，函数定义如下（/kernel/locking/rwsem.c）：

```
void up_write(struct rw_semaphore *sem)
{
    rwsem_release(&sem->dep_map, 1, _RET_IP_);

    rwsem_clear_owner(sem);
    __up_write(sem);     /*/kernel/locking/rwsem-spinlock.c*/
}
```

__up_write()函数定义如下：

```
void __up_write(struct rw_semaphore *sem)
{
    unsigned long flags;

    raw_spin_lock_irqsave(&sem->wait_lock, flags);

    sem->count = 0;       /*count 值置 0*/
    if (!list_empty(&sem->wait_list))       /*等待队列不为空，唤醒睡眠等待进程*/
        sem = __rwsem_do_wake(sem, 1);       /*/kernel/locking/rwsem-spinlock.c*/

    raw_spin_unlock_irqrestore(&sem->wait_lock, flags);
}
```

写者释放信号量的操作将信号量 count 值置 0，如果等待队列双链表不为空将唤醒等待的进程。如果第一个等待进程为写者，则只唤醒第一个进程，否则唤醒双链表中第一个写者进程之前的所有读者进程。

唤醒等待进程的函数__rwsem_do_wake()定义如下：

```
static inline struct rw_semaphore *__rwsem_do_wake(struct rw_semaphore *sem, int wakewrite)
/*wakewrite：传递参数值为 1，表示唤醒写者进程*/
{
    struct rwsem_waiter *waiter;
    struct task_struct *tsk;
    int woken;

    waiter = list_entry(sem->wait_list.next, struct rwsem_waiter, list);   /*第一个等待进程*/

    if (waiter->type == RWSEM_WAITING_FOR_WRITE) {     /*如果是写者进程*/
        if (wakewrite)                  /*直接唤醒第一个进程*/
            wake_up_process(waiter->task);
        goto out;
    }
```

```
        /*如果第一个等待进程为读者进程，则唤醒第一个写者进程之前的所有读者进程*/
        woken = 0;
        do {
            struct list_head *next = waiter->list.next;
            list_del(&waiter->list);
            tsk = waiter->task;
            smp_mb();
            waiter->task = NULL;     /*唤醒等待的读进程*/
            wake_up_process(tsk);    /*唤醒进程*/
            put_task_struct(tsk);      /*读者进程在进入睡眠时调用了 get_task_struct(tsk)*/
            woken++;
            if (next == &sem->wait_list)   /*扫描到双链表末尾*/
                break;
            waiter = list_entry(next, struct rwsem_waiter, list);   /*下一个睡眠等待进程*/
        } while (waiter->type != RWSEM_WAITING_FOR_WRITE);   /*遇到写者进程，跳出循环*/

        sem->count += woken;   /*唤醒读者进程数量*/
    out:
        return sem;
    }
```

__rwsem_do_wake()函数检测信号量等待队列双链表中第一个等待进程是写者还是读者，如果是写者则只唤醒第一个写者进程。如果第一个等待进程是读者，则唤醒队列中连续的读者进程，直到遇到写者进程，信号量 count 值设为唤醒的读者进程数量。

内核还定义了其它的读写信号量操作函数，例如（源代码请读者自行阅读）：

●int **down_read_trylock**(struct rw_semaphore *sem)：尝试获取读者信号量，不成功不等待，成功返回1。

●int **down_write_trylock**(struct rw_semaphore *sem)：尝试获取写者信号量，不成功不等待，成功返回1。

获取/释放读者信号量、获取/释放写者信号量的函数必须成对出现。读者可同时进入临界区，而写者只能单独进入临界区。读者进入临界区后，睡眠等待的第一个进程必定是写者进程，其后可能是写者或读者进程。读者退出临界区时，唤醒第一个写者进程。写者进入临界区后，其后的写者和读者都需要进入睡眠等待，写者退出临界区后，将唤醒第一个进程（写者进程）或第一个写者进程之前的所有读者进程。

6.9.7 互斥量

互斥量（又称互斥锁）只允许一个路径进入临界区，适合保护比较长的临界区。前面介绍的互斥信号量可以实现互斥量一样的功能，那么为什么还要单独实现互斥量呢？设计者认为互斥量语义相对于信号量要简单一些，在锁争用激烈的测试场景下，互斥量比信号量执行速度更快，可扩展性更好，另外互斥量数据结构的定义比信号量小，互斥量上的一些优化方案已经移植到了读写信号量中。

互斥量分为普通互斥量和实时互斥量。互斥量保护的临界区允许中断和内核抢占。

1 普通互斥量

普通互斥量相关的数据结构定义和接口函数声明在/include/linux/mutex.h 头文件，普通互斥量相关函数实现在/kernel/locking/mutex.c 文件内。

普通互斥量由 mutex 结构体表示，结构体定义在/include/linux/mutex.h 头文件内：

```
struct mutex {
    atomic_t        count;      /*互斥量值*/
    spinlock_t      wait_lock;  /*保护互斥量的自旋锁*/
    struct list_head    wait_list;  /*睡眠等待进程双链表*/
    ...
};
```

mutex 结构体主要成员简介如下：

●**count**：互斥量计数值，1 表示没有进程持有互斥量（也没有等待进程），0 表示有一个进程持有互斥量，但没有等待进程，-1 表示互斥量已被持有并且有进程在等待获取互斥量。

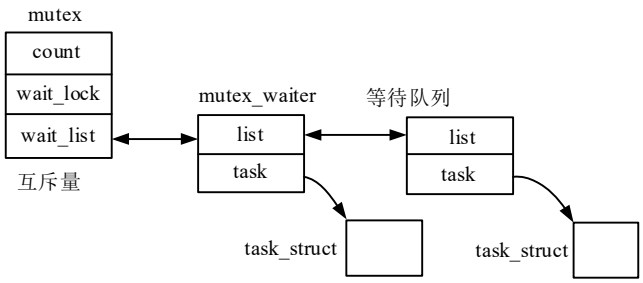
●**wait_lock**：保护互斥量结构体其它成员的自旋锁。

●**wait_list**：双链表成员，管理在互斥量上等待的进程，链表成员为 mutex_waiter 结构体实例。

mutex_waiter 结构体定义在/include/linux/mutex.h 头文件：

```
struct mutex_waiter {
    struct list_head    list;   /*双链表成员*/
    struct task_struct  *task;  /*指向等待进程 task_struct*/
    ...             /*OSQ 锁实现*/
};
```

互斥量数据结构组织关系如下图所示：



内核在/include/linux/mutex.h 头文件定义了创建并初始化普通互斥量的宏：

```
#define DEFINE_MUTEX(mutexname) \
    struct mutex mutexname = __MUTEX_INITIALIZER(mutexname)

#define __MUTEX_INITIALIZER(lockname) \
        { .count = ATOMIC_INIT(1) \         /*计数值初始化为 1*/
        , .wait_lock = __SPIN_LOCK_UNLOCKED(lockname.wait_lock) \   /*自旋锁初始化*/
        , .wait_list = LIST_HEAD_INIT(lockname.wait_list) \         /*双链表头初始化*/
        ...
        }
```

mutex_init(mutex)函数用于初始化一个已创建的互斥量，初始化内容与上面宏所做的初始化工作相同。

■获取互斥量

获取互斥量的函数为 mutex_lock(struct mutex *lock)，函数定义如下（/kernel/locking/mutex.c）：

```
void __sched mutex_lock(struct mutex *lock)
{
    might_sleep();      /*可能睡眠*/
    __mutex_fastpath_lock(&lock->count, __mutex_lock_slowpath);
                                        /*include/asm-generic/mutex-dec.h*/
    mutex_set_owner(lock);
}
```

__mutex_fastpath_lock()函数首先将 count 值置 0，并判断原值是否为 1（原子操作），如果原值为 1，函数返回，表示获取互斥量成功，这是快速路径。

如果置 0 后，count 原值不为 1（为 0 或-1），则将 count 值置-1，并调用__mutex_lock_slowpath()函数，进入慢速路径。也就是说如果 count 值为 1，则获取互斥量成功，并将其置 0。如果 count 值为 0 或-1，则将 count 值置-1，并调用__mutex_lock_slowpath()函数，当前进程进入睡眠等待。

如果当前进程不能持有互斥量，将调用__mutex_lock_slowpath()函数将当前进程添加到互斥量等待队列双链表，函数定义如下（/kernel/locking/mutex.c）：

```
__visible void __sched __mutex_lock_slowpath(atomic_t *lock_count)
{
    struct mutex *lock = container_of(lock_count, struct mutex, count);

    __mutex_lock_common(lock, TASK_UNINTERRUPTIBLE, 0,NULL, _RET_IP_, NULL, 0);
                                        /*进入不可中断睡眠状态*/
}
```

__mutex_lock_common()函数是一个通用的函数，用于使进程在互斥量上等待，函数定义如下：

```
static __always_inline int __sched __mutex_lock_common(struct mutex *lock, long state,
            unsigned int subclass,struct lockdep_map *nest_lock, unsigned long ip,
            struct ww_acquire_ctx *ww_ctx, const bool use_ww_ctx)
/*state：进程进入的状态，这里为 TASK_UNINTERRUPTIBLE*/
{
    struct task_struct *task = current;
    struct mutex_waiter waiter;     /*mutex_waiter 结构体实例*/
    unsigned long flags;
    int ret;

    preempt_disable();     /*禁止内核抢占*/
    if (mutex_optimistic_spin(lock, ww_ctx, use_ww_ctx)) {          /*如果使用了 OSQ 锁*/
        preempt_enable();
        return 0;
    }
```

```
spin_lock_mutex(&lock->wait_lock, flags);     /*获取自旋锁*/

if (!mutex_is_locked(lock) && (atomic_xchg(&lock->count, 0) == 1))
      goto skip_wait;         /*count 值为 1，则将 count 值置 0，函数返回，获得互斥量*/

debug_mutex_lock_common(lock, &waiter);
debug_mutex_add_waiter(lock, &waiter, task_thread_info(task));

/*count 值不为 1，执行以下代码*/
list_add_tail(&waiter.list, &lock->wait_list);    /*mutex_waiter 添加到等待队列双链表末尾*/
waiter.task = task;
lock_contended(&lock->dep_map, ip);
for (;;) {
      if (atomic_read(&lock->count) >= 0 &&(atomic_xchg(&lock->count, -1) == 1))
            break;       /*如果 count>=0，将 count 置于-1，且原值为 1 则获取互斥量成功*/

      /*有挂起信号，且可被信号唤醒，返回错误码*/
      if (unlikely(signal_pending_state(state, task))) {
            ret = -EINTR;
            goto err;
      }
      if (use_ww_ctx && ww_ctx->acquired > 0) {          /*use_ww_ctx 为 0*/
            ret = __ww_mutex_lock_check_stamp(lock, ww_ctx);
            if (ret)
                  goto err;
      }

      __set_task_state(task, state);     /*设置进程状态（这里为 TASK_UNINTERRUPTIBLE）*/

      spin_unlock_mutex(&lock->wait_lock, flags);     /*释放自旋锁*/
      schedule_preempt_disabled();     /*进程调度，/kernel/sched/core.c*/
      spin_lock_mutex(&lock->wait_lock, flags);     /*唤醒后获取自旋锁*/
}

/*进程被释放互斥量的进程唤醒*/
__set_task_state(task, TASK_RUNNING);    /*设置进程为可运行状态*/

mutex_remove_waiter(lock, &waiter, current_thread_info());
                                          /*mutex_waiter 实例从等待双链表移除*/
if (likely(list_empty(&lock->wait_list))) /*等待双链表为空，将 count 值置 0，表示没有等待进程*/
      atomic_set(&lock->count, 0);
debug_mutex_free_waiter(&waiter);
```

```
skip_wait:
        lock_acquired(&lock->dep_map, ip);
        mutex_set_owner(lock);

        if (use_ww_ctx) {        /*这里为 0*/
            struct ww_mutex *ww = container_of(lock, struct ww_mutex, base);
            ww_mutex_set_context_slowpath(ww, ww_ctx);
        }
        spin_unlock_mutex(&lock->wait_lock, flags);   /*释放自旋锁*/
        preempt_enable();       /*使能内核抢占*/
        return 0;
        ...
    }
```

__mutex_lock_common()函数不难理解，不能获取互斥量时，当前进程添加到互斥量睡眠等待队列双链表末尾，进入睡眠，将互斥量计数值设为-1。进程被唤醒时再次尝试获取信号量（count 值为 1），获取成功，则将进程从等待队列双链表中移除，判断双链表是否为空，是则将 count 置 0，否则保持-1。如果唤醒后，count 值不为 1，则进程继续睡眠等待。

内核还定义了普通互斥量其它的操作接口函数，例如：

●int **mutex_lock_interruptible**(struct mutex *lock)：获取互斥量，不成功，进程进入可中断睡眠状态。

●int **mutex_lock_killable**(struct mutex *lock)：获取互斥量，不成功进程进入中度睡眠状态，可被致命信号唤醒。

●int **mutex_trylock**(struct mutex *lock)：尝试获取互斥量，成功返回 1，不成功返回 0，不睡眠。

■释放互斥量

释放互斥量的函数 mutex_unlock(struct mutex *lock)定义如下（/kernel/locking/mutex.c）：

```
void __sched mutex_unlock(struct mutex *lock)
{
    ...
    __mutex_fastpath_unlock(&lock->count, __mutex_unlock_slowpath);
                                        /*include/asm-generic/mutex-dec.h*/
}
```

__mutex_fastpath_unlock()函数对 count 值加 1（原子操作），加 1 后仍小于等于 0，表示有等待的进程，则调用__mutex_unlock_slowpath()函数唤醒等待进程。加 1 后若 count 值为 1，表示没有等待的进程，直接返回。

__mutex_unlock_slowpath()函数定义如下（/kernel/locking/mutex.c）：

```
__visible void __mutex_unlock_slowpath(atomic_t *lock_count)
{
    struct mutex *lock = container_of(lock_count, struct mutex, count);

    __mutex_unlock_common_slowpath(lock, 1);
```

```
}
```

__mutex_unlock_common_slowpath()函数定义如下，用于唤醒一个在互斥量上等待的进程：

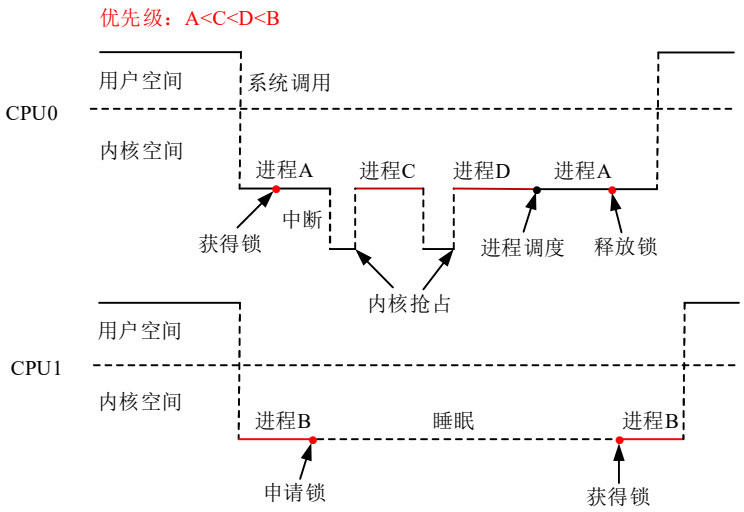
```
static inline void __mutex_unlock_common_slowpath(struct mutex *lock, int nested)
{
        unsigned long flags;
        if (__mutex_slowpath_needs_to_unlock())     /*返回 1*/
              atomic_set(&lock->count, 1);       /*count 值置 1*/

        spin_lock_mutex(&lock->wait_lock, flags);
        debug_mutex_unlock(lock);
        if (!list_empty(&lock->wait_list)) {   /*如果等待进程双链表不为空，唤醒队列中第一个进程*/
            struct mutex_waiter *waiter =list_entry(lock->wait_list.next,struct mutex_waiter, list);
            debug_mutex_wake_waiter(lock, waiter);
            wake_up_process(waiter->task);     /*唤醒进程*/
        }
        spin_unlock_mutex(&lock->wait_lock, flags);
}
```

__mutex_unlock_common_slowpath()函数将 count 值置 1，并唤醒第一个睡眠等待的进程即可。在互斥量上等待的进程唤醒后，会重新设置 count 值，详见前面介绍的__mutex_lock_common()函数。

2 实时互斥量

进程获取信号量、普通互斥量，当不能获取成功时，将添加到信号量、普通互斥量的等待队列中。进程是以申请的时间先后顺序在等待队列中排列，唤醒时也是按时间先后顺序唤醒的，这看似公平，但是没有考虑进程的优先级。例如，假设一个普通进程在一个实时进程之前申请信号量、普通互斥量，它将在实时进程之前被唤醒运行，普通进程抢在了实时进程之前运行，这称为优先级反转，将影响系统的实时性。

更糟糕的是如果持有以上锁的进程在临界区被其它进程抢占了，将延长锁释放的时间，高优先级的进程将等待更长的时间。如下图所示：



假设上图中进程优先级顺序为：A<C<D<B。A 优先级最低，B 的优先级最高。A 进入了临界区，随

后 B 申请锁，申请不成功，进入睡眠等待，A 在临界区相继被进程 C、D 抢占，最后 A 继续运行，离开临界区释放锁，这时 B 才被唤醒，得以运行，进入临界区。在这种情况下，优先级比 B 更低的 C、D 都抢在了 B 之前运行，高优先级的进程 B 经过了长时间等待，这将影响系统的实时性。

实时互斥量，用红黑树来管理睡眠等待的进程，等待进程以优先级从高到低在红黑树中从左至右排列，释放锁唤醒进程时，唤醒优先级最高的进程。添加等待进程到红黑树时，会将当前持有实时互斥量进程的优先级（动态优先级，临时的）调整为红黑树中等待进程优先级的最高值，这称为优先级继承。以防止持有实时互斥量的进程在临界区被低优先级的进程抢占，保证其尽快离开临界区，释放实时互斥量。

若要内核支持实时斥量需选择 RT_MUTEXES 配置选项，内核配置选择 FUTEX 选项时（/init/Kconfig）将自动选择 RT_MUTEXES 配置选项（没有单独选择 RT_MUTEXES 的选项）。

实时互斥量数据结构定义及接口函数声明在/include/linux/rtmutex.h 头文件，在/kernel/locking/rtmutex.c 文件内实现实时互斥量操作的接口函数。

■数据结构

实时互斥量由 rt_mutex 结构体表示，结构体定义如下（/include/linux/rtmutex.h）：

```
struct rt_mutex {
    raw_spinlock_t      wait_lock;     /*保护实时互斥量其它成员的自旋锁*/
    struct rb_root          waiters;       /*红黑树根节点，管理等待实时互斥量的进程*/
    struct rb_node        *waiters_leftmost;       /*红黑树中最左边节点（优先级最高的进程）*/
    struct task_struct    *owner;          /*指向当前持有本实时互斥量的进程 task_struct 实例*/
    ...
};
```

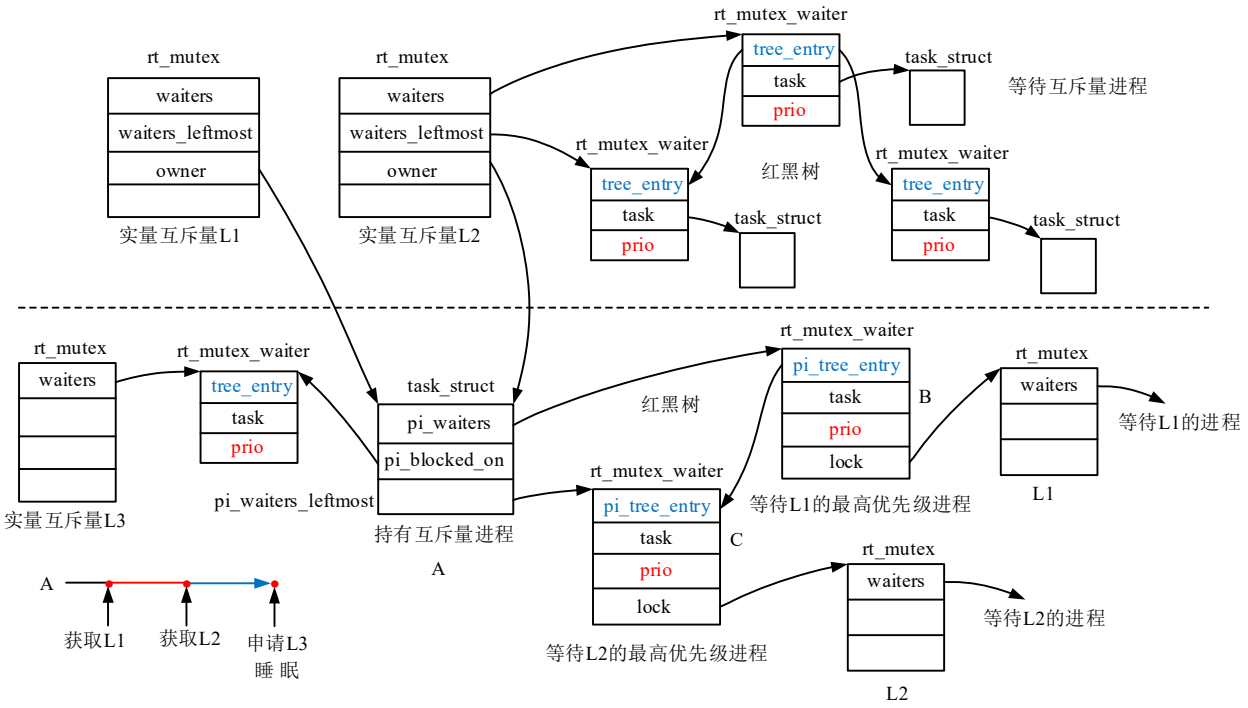
实时互斥量某一时刻只能被一个进程持有，owner 成员指向持有实时互斥量的进程。实时互斥量通过红黑树来管理等待的进程，进程在红黑树中按优先级从高到低在树中从左至右排列（限期调度类中进程按到期时间从小到大排列）。

红黑树中节点为 rt_mutex_waiter 结构体实例，结构体定义在/kernel/locking/rtmutex_common.h 头文件：

```
struct rt_mutex_waiter {
    struct rb_node          tree_entry;      /*红黑树节点*/
    struct rb_node          pi_tree_entry;
                        /*添加到持有实时互斥量进程 task_struct 实例中的优先级继承（pi）红黑树*/
    struct task_struct    *task;        /*关联进程结构*/
    struct rt_mutex       *lock;       /*指向实时互斥量*/
    ...
    int prio;       /*优先级*/
};
```

睡眠等待进程通过关联 rt_mutex_waiter 实例的 tree_entry 成员，按优先级从高到低在实时互斥量管理的红黑树中从左至右排列。另外，rt_mutex_waiter 实例的 pi_tree_entry 成员，可能添加到持有实时互斥量进程管理的红黑树中，见下文。

以上数据结构组织关系如下图所示：



进程 task_struct 结构体中与实时互斥量相关的成员如下：

```
task_struct{
    ...
    raw_spinlock_t pi_lock;              /*保护 pi_waiters 红黑树的自旋锁*/
    struct wake_q_node wake_q;           /*唤醒时用于添加到唤醒进程队列*/
#ifdef CONFIG_RT_MUTEXES                 /*需支持实时互斥量*/
    struct rb_root pi_waiters;           /*红黑树根节点，管理 rt_mutex_waiter 实例*/
    struct rb_node *pi_waiters_leftmost;     /*pi_waiters 红黑树中最左边节点*/
    struct rt_mutex_waiter *pi_blocked_on;   /*指向进程关联的 rt_mutex_waiter 实例*/
#endif
    ...
}
```

task_struct 结构体中 pi_waiters 结构体是一个红黑树根节点，管理 rt_mutex_waiter 实例。下面将解释此红黑树中 rt_mutex_waiter 实例的来源。

一个进程可以同时持有多个实时互斥量，如上图中所示。假设进程 A 持有实时互斥量 L1 和 L2，在等待 L3，进程 A 的 pi_waiters 红黑树管理的是等待 L1 的最高优先级进程 B 和等待 L2 的最高优先级进程 C，也就是 L1 管理红黑树中最左边节点和 L2 管理红黑树中最左边节点。

为了防止优先级反转，进程 A 优先级（在临界区中临时的动态优先级）不能低于 B 和 C 之中的最高优先级，因为 A 同时持有两个实时互斥量，需同时满足两个不发生优先级反转的条件。

另外，实时互斥量还要考虑死锁问题（信号量、互斥量也有，编程时要特别注意）。例如，在上图中 A 持有 L1 和 L2，在等待 L3，若 L3 被在等待 L1 或 L2 的进程持有，将发生死锁。如下所示假设 D 持有 L3：

D->L1->A->L3->D

上式表示：D 在等待 L1，L1 被 A 持有，A 在等待 L3，L3 被 D 持有，此时将发生死锁。

读者可参考内核实时互斥量说明文档/Documentation/locking/rt-mutex-design.txt 等。

内核中创建并初始化实时互斥量的宏定义如下：

```
#define DEFINE_RT_MUTEX(mutexname) \
    struct rt_mutex mutexname = __RT_MUTEX_INITIALIZER(mutexname)

#define __RT_MUTEX_INITIALIZER(mutexname) \
    { .wait_lock = __RAW_SPIN_LOCK_UNLOCKED(mutexname.wait_lock) \     /*初始化自旋锁*/
    , .waiters = RB_ROOT \       /*初始化红黑树根节点*/
    , .owner = NULL \            /*指向 NULL*/
    ...}
```

rt_mutex_init(mutex)函数用于初始化已经创建的实时互斥量，初始化工作与上面的宏相同。

■获取实时互斥量

获取实时互斥量的 **rt_mutex_lock**(struct rt_mutex *lock)函数定义如下（/kernel/locking/rtmutex.c）：

```
void __sched rt_mutex_lock(struct rt_mutex *lock)
{
    might_sleep();        /*可能睡眠*/
    rt_mutex_fastlock(lock, TASK_UNINTERRUPTIBLE, rt_mutex_slowlock);
                                        /*rt_mutex_slowlock()函数表示慢速路径函数*/
}
```

rt_mutex_fastlock()函数定义如下（/kernel/locking/rtmutex.c）：

```
static inline int rt_mutex_fastlock(struct rt_mutex *lock, int state,
            int (*slowfn)(struct rt_mutex *lock, int state,struct hrtimer_sleeper *timeout,
                                                    enum rtmutex_chainwalk chwalk))
{
    if (likely(rt_mutex_cmpxchg(lock, NULL, current))) {      /*快速路径是否成功，成功返回 1*/
                                /*lock->owner 为 NULL，则赋为 current，返回 1，否则返回 0*/
        rt_mutex_deadlock_account_lock(lock, current);
                                        /*没有选择 DEBUG_RT_MUTEXES 为空操作*/
        return 0;
    } else
        return slowfn(lock, state, NULL, RT_MUTEX_MIN_CHAINWALK);
                                        /*慢速路径，调用 rt_mutex_slowlock()函数*/
}
```

这里假设没有选择 DEBUG_RT_MUTEXES 配置选项（/lib/Kconfig.debug），rt_mutex_cmpxchg()函数是获取实时互斥量的快速路径，函数内判断 lock->owner 是否为 NULL（没有进程持有实时互斥量），是则赋值 current（当前进程结构实例），函数返回 1，否则返回 0。rt_mutex_cmpxchg()函数返回 1 表示获取实时互斥量成功，rt_mutex_lock()函数返回。

简单地说快速路径就是判断 lock->owner 是否为 NULL，若是则表示没有进程持有实时互斥量，当前进程将持有实时互斥量，rt_mutex_lock()函数返回。

若 lock->owner 不为 NULL，表示实时互斥量已经被其它进程持有，则调用 rt_mutex_slowlock()函数进入慢速路径。

rt_mutex_slowlock()函数定义如下（/kernel/locking/rtmutex.c）：

```
static int __sched rt_mutex_slowlock(struct rt_mutex *lock, int state,
                                     struct hrtimer_sleeper *timeout,enum rtmutex_chainwalk chwalk)
/*state：TASK_UNINTERRUPTIBLE，timeout：NULL（超时定时器），
*chwalk：此处为 RT_MUTEX_MIN_CHAINWALK，
*rtmutex_chainwalk 枚举类型定义在/kernel/locking/rtmutex_common.h 头文件。
*/
{
    struct rt_mutex_waiter waiter;          /*定义 rt_mutex_waiter 实例*/
    int ret = 0;

    debug_rt_mutex_init_waiter(&waiter);
    RB_CLEAR_NODE(&waiter.pi_tree_entry);
    RB_CLEAR_NODE(&waiter.tree_entry);

    raw_spin_lock(&lock->wait_lock);        /*获取自旋锁*/

    /*再次尝试能否获取实时斥量*/
    if (try_to_take_rt_mutex(lock, current, NULL)) {        /*lock->owner=current*/
        raw_spin_unlock(&lock->wait_lock);
        return 0;
    }

    set_current_state(state);          /*设置当前进程状态*/

    /*启动超时定时器，如果有的话*/
    if (unlikely(timeout))
        hrtimer_start_expires(&timeout->timer, HRTIMER_MODE_ABS);

    ret = task_blocks_on_rt_mutex(lock, &waiter, current, chwalk);        /*准备睡眠，检测死锁等*/

    if (likely(!ret))
        ret = __rt_mutex_slowlock(lock, state, timeout, &waiter);
                /*当前进程进入睡眠等待，唤醒后尝试获取实时互斥量，lock->owner=current*/

    /*获得实时互斥量成功，执行 if 内语句*/
    if (unlikely(ret)) {
        __set_current_state(TASK_RUNNING);        /*设为可运行状态*/
```

```
            if (rt_mutex_has_waiters(lock))        /*除本进程外，还有等待实时互斥量的进程*/
                remove_waiter(lock, &waiter);          /*移除 waiter 实例等*/
            rt_mutex_handle_deadlock(ret, chwalk, &waiter);      /*处理死锁，输出信息，进程调度等*/
        }

        fixup_rt_mutex_waiters(lock);      /*若没有等待实时互斥量的进程了，lock->owner 清零*/
        raw_spin_unlock(&lock->wait_lock);

        /*移除超时定时器*/
        if (unlikely(timeout))
            hrtimer_cancel(&timeout->timer);

        debug_rt_mutex_free_waiter(&waiter);
        return ret;
    }
```

rt_mutex_slowlock()函数的主要工作是构建 rt_mutex_waiter 实例，添加到前面介绍的数据结构中，当前进程进入睡眠等待，被唤醒后（获得实时互斥量），将 rt_mutex_waiter 实例从以上数据结构中移出。

rt_mutex_slowlock()函数中调用的 task_blocks_on_rt_mutex()和 remove_waiter()函数执行的工作是相对的。

task_blocks_on_rt_mutex()函数将 rt_mutex_waiter 实例添加到申请实时互斥量管理的红黑树中，若是最高优先级等待进程，还需要调整当前持有实时互斥量进程的优先级，并将 rt_mutex_waiter 实例添加到其下 pi_waiters 红黑树中。另外，还可能需要执行死锁检测。

进程被唤醒获得实时互斥量后，调用 remove_waiter()函数将 rt_mutex_waiter 实例从以上数据结构中移出，执行与 task_blocks_on_rt_mutex()函数相反的工作。有兴趣的读者可阅读相关源代码。

获取实时互斥量还有其它变体函数，例如：

●int **rt_mutex_lock_interruptible**(struct rt_mutex *lock)：获取实时互斥量，不成功进程进入可中断睡眠状态。

●int **rt_mutex_timed_lock**(struct rt_mutex *lock,struct hrtimer_sleeper *timeout)：获取实时互斥量，带超时限制的。

●int **rt_mutex_trylock**(struct rt_mutex *lock)：尝试获取实时互斥量，成功返回 1，不成功返回 0，不等待。

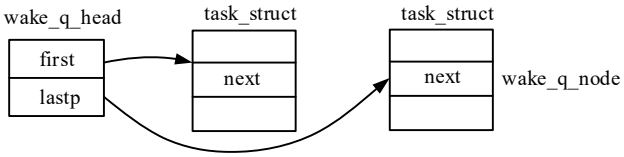
■释放实时互斥量

在介绍释放实时互斥量前，先看一下 wake_q_head 和 wake_q_node 结构体定义，如下所示：

```
struct wake_q_head {          /*唤醒队列头，/include/linux/sched.h*/
    struct wake_q_node *first;
    struct wake_q_node **lastp;
};
```

```
struct wake_q_node {            /*唤醒队列成员，/include/linux/sched.h*/
    struct wake_q_node *next;       /*指向下一个节点*/
};
```

wake_q_head 表示一个队列头，管理要唤醒进程队列（单链表），队列成员为 wake_q_node。进程结构 task_struct 结构体中嵌入了 wake_q_node 结构体成员 wake_q，如下图所示。



WAKE_Q(name)宏用于定义并初始化一个名称为 name 的 wake_q_head 实例。释放实时互斥量时，要唤醒的进程通过 wake_q 成员添加到 wake_q_head 队列（只有一个唤醒进程）。

释放实时互斥量的函数为 rt_mutex_unlock()，定义如下（/kernel/locking/rtmutex.c）：

```
void __sched rt_mutex_unlock(struct rt_mutex *lock)
{
    rt_mutex_fastunlock(lock, rt_mutex_slowunlock);     /*慢速路径函数为 rt_mutex_slowunlock()*/
}

static inline void rt_mutex_fastunlock(struct rt_mutex *lock,
                                       bool (*slowfn)(struct rt_mutex *lock,struct wake_q_head *wqh))
{
    WAKE_Q(wake_q);             /*初始化唤醒进程队列头*/

    if (likely(rt_mutex_cmpxchg(lock, current, NULL))) {       /*lock->owner 为 current，则赋为 NULL*/
        rt_mutex_deadlock_account_unlock(current); /*没有选择 DEBUG_RT_MUTEXES 为空操作*/

    } else {            /*调用 rt_mutex_slowunlock()函数选择等待进程，并唤醒*/
        bool deboost = slowfn(lock, &wake_q);       /*选择唤醒的等待进程，有等待进程返回 true*/

        wake_up_q(&wake_q);         /*唤醒进程，唤醒进程执行 lock->owner=current*/

        if (deboost)
            rt_mutex_adjust_prio(current);          /*恢复当前进程优先级*/
    }
}
```

rt_mutex_fastunlock()函数的快速路径中，检测 lock->owner 是否为 current，是则赋值 NULL，函数返回，释放实时互斥量结束。

这里要说明的一点是 lock->owner 的最低 2 位是标记位，若 lock->owner 等于 current，暗示最低 2 位为 0，表示没有等待实时互斥量的进程了，因此 lock->owner 赋 NULL 后可以直接返回。如果 lock->owner 的最低 2 位不是 0，则不等于 current，表示还有等待实时互斥量的进程，还需要执行慢速路径。读者可阅读内核关于实时互斥量的说明文档。

慢速路径 rt_mutex_slowunlock()函数，用于选择唤醒等待进程，最后由 wake_up_q()函数唤醒等待进程。

rt_mutex_slowunlock()函数定义如下（/kernel/locking/rtmutex.c）：

```
static bool __sched rt_mutex_slowunlock(struct rt_mutex *lock,struct wake_q_head *wake_q)
{
    raw_spin_lock(&lock->wait_lock);

    debug_rt_mutex_unlock(lock);          /*没有选择 DEBUG_RT_MUTEXES 为空操作*/
    rt_mutex_deadlock_account_unlock(current);   /*没有选择 DEBUG_RT_MUTEXES 为空操作*/

    while (!rt_mutex_has_waiters(lock)) {          /*没有等待进程了，则执行循环体，否则不执行*/
        if (unlock_rt_mutex_safe(lock) == true)
            return false;
        raw_spin_lock(&lock->wait_lock);
    }

    mark_wakeup_next_waiter(wake_q, lock);
                    /*选择实时互斥量红黑树中最左边节点关联的进程，添加到 wake_q 队列*/
    raw_spin_unlock(&lock->wait_lock);
    return true;        /*还有等待进程，返回 true*/
}
```

6.9.8 RCU 机制

RCU（Read-Copy-Update）的意思是读-复制-更新，是内核最复杂的并发控制机制。RCU 主要用于保护用指针引用的共享数据，即内核动态分配的数据（静态分配的不行），可以防止在并发读写访问时出现无效的指针。

前面介绍的锁机制，要么只能有一个读者或写者进入临界区，要么只允许多个读者同时进入临界区，而写者需单独进入临界区。RCU 机制允许多个读者同时进入临界区，还允许多个读者和单个写者同时进入临界区，但是多个写者之间还是互斥的。RCU 机制对读者的开销小（可忽略不计），而将同步开销留给写者。

RCU 机制如何实现以上功能呢？读者在读共享数据前，创建一个指针，指向共享数据，可直接进行读操作，并允许多个读者同时执行读操作。写者在写之前创建一个共享数据的副本，复制原数据，在副本上进行修改，然后将指向共享数据的指针指向新创建的副本，最后在所有读者离开临界区后（读操作完成）释放原共享数据（用新实例替换旧实例）。

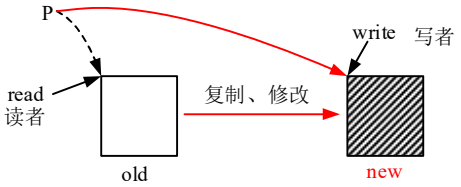
RCU 机制适用于保护读多写少的共享数据。RCU 机制相关代码位于/kernel/rcu/目录下。

1 概述

先简要介绍一下 RCU 机制的原理，如下图所示。假设内核指针 p 指向的数据结构实例是由 RCU 机制保护的共享数据。读者访问共享数据时，先创建指针 read，read 指向 p 指向的共享数据，读者通过 read 对共享数据进行读操作，允许多个读者同时读共享数据。

写者修改共享数据前先分配一个共享数据新实例，由 write 指针指向新实例，然后将 p 指向实例的数

据复制到新实例，并对新实例进行修改（写操作），修改完成后将 p 指向新实例（新实例替换旧实例），最后在所有读者都离开临界区时释放旧实例。写者与写者之间要通过锁机制来保证串行化，不可并发。



RCU 机制的重点和难点是如何判断所有读者都离开临界区。RCU 机制中有两个非常重要的概念，用于表示 CPU 核的状态（表示 CPU 核是否在读临界区内），分别是静止状态 QS（Quiescent State）和宽限期 GP（GracePeriod）。

静止状态：如果一个 CPU 核处于 RCU 读者临界区中，就说 CPU 核的状态是活跃的。CPU 核不在 RCU 读者临界区，就说 CPU 的状态是静止的。

宽限期：宽限期是对系统内所有 CPU 核而言的，不是针对哪一个 CPU 核的，这是 RCU 机制设置的一个期限（时间段）。宽限期有开始和结束之分，从宽限期开始那一刻开始，当所有 CPU 核都离开 RCU 读者临界区时，认为一个宽限期结束了，此时 RCU 机制会销毁在本宽限期及之前注册的需要销毁的旧共享数据实例，因为读者都离开临界区了，已经没有读者在访问这些旧实例了。

根据在读者临界区中的状态 RCU 机制分为以下 4 种类型（写者临界区由自旋锁保护）：

（1）**不可抢占 RCU**（RCU-sched）：在读者临界区禁止内核抢占。

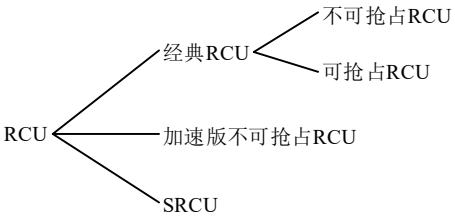
（2）**加速版不可抢占 RCU**（RCU-bh）：在读者临界区禁止软中断，同时也禁止了内核抢占。

（3）**可抢占 RCU**（RCU-preempt）：也称为实时 RCU，允许在读者临界区中被抢占。

（4）**可睡眠 RCU**（SRCU）：在写者临界区允许睡眠。

经典 RCU 根据内核配置选项，选择不可抢中 RCU 或可抢占 RCU 实现。

RCU 机制分类如下图所示：



RCU 相关配置选项位于/init/Kconfig 文件内。经典 RCU 实现方式与 PREEMPT_RCU 配置选项相关：

●**PREEMPT**：内核若选择了 PREEMPT 选项，则自动选择 PREEMPT_RCU 配置选项，经典 RCU 实现为可抢占 RCU。

●**!PREEMPT**：没有选择 PREEMPT 选项，不选择 PREEMPT_RCU 配置选项，经典 RCU 实现为不可抢占 RCU。

加速版不可抢占 RCU 始终实现。可睡眠 RCU 需选择 SRCU 选项（选择 TASKS_RCU 时自动选择 SRCU），由体系结构（平台）相关配置文件选择此选项。

经典 RCU 机制接口函数如下（声明在/include/linux/rcupdate.h 和/include/linux/rcutree.h 头文件）：

读者接口函数：

●**rcu_read_lock()**：读者进入读临界区，不可抢占 RCU 禁止内核抢占，可抢占 RCU 主要包含一个编译器屏障。

●**read=rcu_dereference(p)**：指向共享数据的指针 p 复制到读指针 read，读者通过 read 读共享数据。

●**rcu_read_unlock()**：读者离开读临界区，不可抢占 RCU 使能内核抢占，可抢占 RCU 主要包含一个编译器屏障。

写者需要先创建新的共享数据实例并执行复制和修改操作（如果有多个写者需要持有自旋锁）：

●***write=*p**：复制共享数据，不是标准的接口函数，随后写者通过 write 指针对副本进行修改。

●**rcu_assign_pointer(p, write)**：将 write 指针复制给 p（需保证是原子操作，由体系结构代码实现），使 p 指向新的共享数据，又称为发布。

●**call_rcu(struct rcu_head *head,void (*func)(struct rcu_head *head))**：向 RCU 机制注册销毁旧共享数据的回调函数，在宽限期结束后，由 RCU 机制自动调用。head 指向共享数据中嵌入的 rcu_head 实例成员，func()为销毁共享数据的函数。

●**rcu_barrier()**：等待所有使用 call_rcu()注册的回调函数执行完毕，函数内可能进入睡眠。

●**synchronize_rcu()**：同步等待宽限期结束（函数可能睡眠），然后由写者释放旧实例。

call_rcu()函数是异步释放旧实例的机制，写者向 RCU 机制注册回调函数，由 RCU 机制在宽限期结束后调用注册的函数释放旧实例。

synchronize_rcu()函数（/kernel/rcu/tree_plugin.h）是由写者同步释放旧实例的机制，写者等待宽限期结束，然后由写者释放旧实例。synchronize_rcu()函数返回时说明宽限期已结束，写者随后可调用 kfree()等函数释放共享数据。

synchronize_rcu_expedited()函数（/kernel/rcu/tree_plugin.h）与 synchronize_rcu()函数类似，也是同步等待宽限期结束，不过它会向其它 CPU 核发送 CPU 核间中断请求，使其它 CPU 核尽快离开临界区，强制快速结束宽限期（加速宽限期）。

synchronize_rcu()和 synchronize_rcu_expedited()函数对于不可抢占 RCU 和可抢占 RCU 有不同的实现。

若要确保使用不可抢占 RCU 应当使用如下接口函数：

●**rcu_read_lock_sched()**：读者进入临界区。

●**read=rcu_dereference_sched(p)**：读者获取共享数据指针。

●**rcu_read_unlock_sched()**：读者离开临界区。

●**call_rcu_sched**(struct rcu_head *head,void (*func)(struct rcu_head *rcu))：注册释放旧实例的回调函数。

●**rcu_barrier_sched**()：等待所有旧实例释放完。

●**synchronize_sched**()：同步等待宽限期结束。

●**synchronize_sched_expedited**()：加速宽限期。

加速不可抢占 RCU 接口函数如下（不受配置选项限制）：

●**rcu_read_lock_bh()**：读者进入临界区，禁止软中断（同时禁止内核抢占）。

●**read=rcu_dereference_bh(p)**：读者获取共享数据指针。

●**rcu_read_unlock_bh()**：读者离开临界区，使能软中断（同时使能内核抢占）。

●**call_rcu_bh**(struct rcu_head *head,void (*func)(struct rcu_head *head))：写者注册回调函数。

●**rcu_barrier_bh**()：等待所有注册的回调函数执行完。

●**synchronize_rcu_bh**()：同步等待宽限期结束。

●**synchronize_rcu_bh_expedited()**：加速宽限期。

可睡眠 RCU 与上面介绍的 RCU 使用的数据结构不同，接口函数类似，在本小节最后再做介绍。

RCU 机制最常见的使用场合是保护大多数时候读的链表（如双链表、散列链表等），这些操作通过 RCU 机制保证操作的原子性，允许并发地对链表进行访问。

RCU 链表操作函数定义在/include/linux/rculist.h、rculist_bl.h、rculist_nulls.h 等头文件内，例如：

●**list_add_rcu**(struct list_head *new, struct list_head *head)：将 new 节点添加到 head 双链表头部。

●**list_add_tail_rcu**(struct list_head *new,struct list_head *head)：将 new 节点添加到 head 双链表末尾。

●**list_del_rcu**(struct list_head *entry)：删除节点 entry。

RCU 链表操作函数并没有改变链表节点数据结构，只是在对链表进行操作时使用了 RCU 机制，相当于持锁操作，对链表结构并没有影响。

下小节后面将简要介绍 RCU 机制使用的数据结构，以及 RCU 机制实现的原理。

2 数据结构

RCU 机制允许读者和写者并发地访问共享数据（写者与写者仍互斥）。若读者与写者同时进入临界区，读者读到的是旧的数据，写者会将新实例赋予引用共享数据的指针，此时读者可能还在临界区，还在读旧实例中的数据。因此，RCU 机制需要等待读者离开临界区，结束读操作后才能释放旧实例。

RCU 机制中使用的数据结构与其它内核锁机制使用的数据结构不同，其它锁机制通常是一把锁保护一个共享数据，而 RCU 机制的数据结构可用于保护所有由 RCU 机制保护的共享数据。RCU 机制不会去区分保护数据结构的类型和数量，CPU 核的静止状态表示 CPU 核离开了所有的 RCU 读临界区，没有访问任何由 RCU 机制保护的共享数据。

宽限期是一个期限，一个时间段，从宽限期开始所有 CPU 核都至少经历了一次静止状态，标志着宽限期结束，以及新宽限期的开始。宽限期结束时，可释放在此之前注册的需要释放的旧实例。

RCU 需要通过一个位图来记录所有 CPU 核的静止状态。在宽限期开始时，置位位图中所有 CPU 核对应的标记位，CPU 核在经历一个静止状态后，清零相应的位。当所有的标记位都清零后，表示所有 CPU 核都退出了临界区（最少经历了一个静止状态），宽限期结束。由于位图是共公数据，需要用自旋锁进行保护，如果系统内 CPU 核数量较多，对自旋锁的竞争将非常激烈，会导致系统性能很差。

为了解决此问题，内核设计了 Tree RCU（树状 RCU），利用层次的树状数据结构来记录 CPU 核的静止状态。Tree RCU 中将 CPU 核按层级进行分组，每组一个位图，当处理器经历一个静止状态时，只需要清零本级组内位图的标记位，当组内所有 CPU 核位图都清零时，再清零上一层级中分组对应的标记位，以减少对位图的竞争。

Tiny RCU（微型 RCU）是 Tree RCU 的一个简化版本，只在不支持内核抢占的单核处理器系统中使用，即内核配置没有选择 SMP 和 PREEMPT 配置选项，其它情况使用 Tree RCU。因此下面将主要介绍 Tree RCU 数据结构的实现。

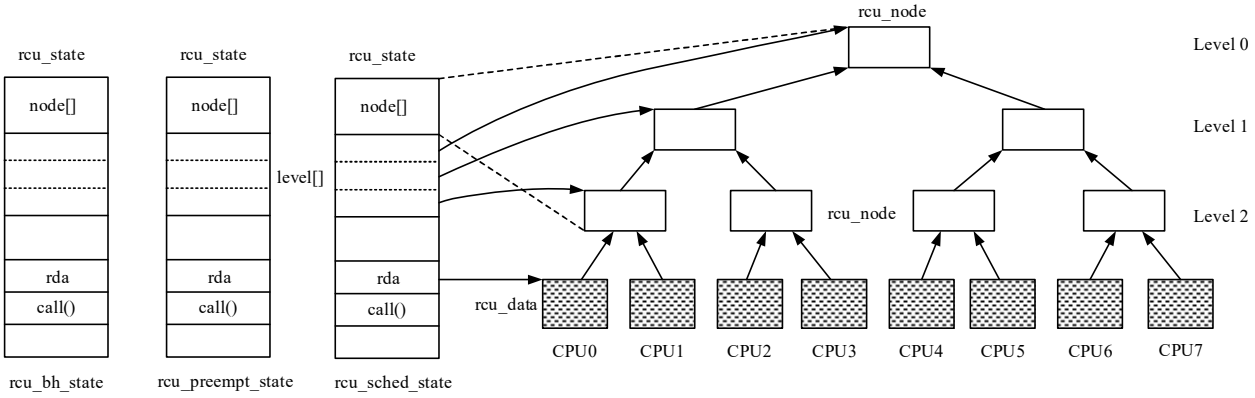
■**RCU 分层树**

Tree RCU 数据结构如下图所示，每种类型的 RCU（不含 SRCU）对应一个 rcu_state 结构体实例，适用于所有由 RCU 保护的共享数据。

rcu_state 结构体中包含各层节点 rcu_node 结构体实例，rcu_node 结构体一个 CPU 核的分组，分组也是分层级的。如下图中所示，在 2 层每 2 个 CPU 核一组，在 1 层每 4 个 CPU 核为一组。

rcu_node 结构体节点中包含组内 CPU 核的静止状态位图，当节点中位图清零时，将清零上一层级中 rcu_node 结构体中对应的标记位。最底层 rcu_node 节点的子节点是 rcu_data 结构体，表示 CPU 核的状态，每个 CPU 核对应一个 rcu_data 实例。

rcu_state 结构体中包含了所有 rcu_node 节点和 rcu_data 实例指针，是最顶层的数据结构。



rcu_state 实例中 rcu_node 节点的最大层级数为 4（MAX_RCU_LVLS，Level0~Level3）。层级结构的实际层级数量及每一层的节点数量受配置参数 NR_CPUS、RCU_FANOUT_LEAF 和 RCU_FANOUT 的控制，内核保证 rcu_node 节点数量最少。

配置选项 RCU_FANOUT_LEAF 表示最底层（每个）rcu_node 节点子节点 rcu_data 实例数量的最大值，即最底层分组组内 CPU 核最大数量。

配置选项 RCU_FANOUTT 表示除最底层外，其它层级中每个节点（rcu_node 实例）子节点（rcu_node 实例）数量的最大值。以上两个配置选项，对于 32 位系统默认值是 32，64 位系统默认值是 64。

内核由以上配置选项及NR_CPUS确定具体层级数量及各层子节点数量，详见/kernel/rcu/tree.h头文件。

rcu_state 结构体定义在/kernel/rcu/tree.h 头文件：

```
struct rcu_state {
    struct rcu_node node[NUM_RCU_NODES];     /*层级结构中所有 rcu_node 实例*/
    struct rcu_node *level[RCU_NUM_LVLS];    /*指向每一层中起始 rcu_node 实例*/
    u32 levelcnt[MAX_RCU_LVLS + 1];          /*每一层级中最大节点数量（并不是实际的数量）*/
    u8 levelspread[RCU_NUM_LVLS];            /*每一层级中节点数量*/
    u8 flavor_mask;                          /* bit in flavor mask. */
    struct rcu_data __percpu *rda;           /*指向 rcu_data 实例，percpu 变量*/
    void (*call)(struct rcu_head *head, void (*func)(struct rcu_head *head)); /* call_rcu() flavor. */
    u8    fqs_state ____cacheline_internodealigned_in_smp;
    u8    boost;                             /* Subject to priority boost. */
    unsigned long gpnum;                     /*当前宽限期编号*/
    unsigned long completed;                 /*上一个结束宽限期的编号*/
    struct task_struct *gp_kthread;          /*指向宽限期线程*/
```

```
    wait_queue_head_t gp_wq;        /*等待宽限期等待进程队列*/
    short gp_flags;                 /*用来向宽限期线程传递命令*/
    short gp_state;                 /*宽限期线程睡眠唤醒的状态*/
    ...
    const char *name;               /*名称*/
    char abbr;                      /* Abbreviated name. */
    struct list_head flavors;       /*将实例添加到 rcu_struct_flavors 全局双链表*/
};
```

rcu_state 结构体中主要成员简介如下：

●**node[NUM_RCU_NODES]**：层级结构中所有 rcu_node 节点实例，节点数量 NUM_RCU_NODES 是能满足需要的最少数量。

●**level[RCU_NUM_LVLS]**：rcu_node 指针数组，指向每层中第一个 rcu_node 实例。

●**rda**：指向每个 CPU 核对应的 rcu_data 实例，percpu 变量。

●**call()**：注册回调函数的函数。

●**gpnum**：当前宽限期编号。

●**completed**：上一个结束宽限期的编号。

●**gp_kthread**：指向宽限期线程 task_struct 实例。宽限期线程用于初始化宽限期、结束宽限期等工作。

●**gp_wq**：宽限期线程等待进程队列。

●**gp_flags**：向宽限期线程传递命令。

●**gp_state**：宽限期线程睡眠状态。

●**flavors**：双链表成员，将实例链接到 rcu_struct_flavors 全局双链表。

Tree RCU 层级结构中除最底层外，其它层级中的节点是 rcu_node 结构体实例。rcu_node 结构体表示一个 CPU 核分组的信息，结构体定义如下（/kernel/rcu/tree.h）：

```
struct rcu_node {
    raw_spinlock_t lock;        /*保护 rcu_node 实例的原始自旋锁*/
    unsigned long gpnum;        /*本节点当前宽限期编号*/
    unsigned long completed; /*本节点上一个结束的宽限期编号*/
    unsigned long qsmask;       /*静止状态位图，标记 CPU 核经历了正常宽限期的静止状态*/
    unsigned long expmask; /*静止状态位图，记录经历了加速宽限期的静止状态*/
    unsigned long qsmaskinit;   /*宽限期开始时 qsmask 的初始值*/
    unsigned long qsmaskinitnext;   /*下一个宽限期开始时 qsmask 的初始值*/
    unsigned long grpmask;      /*本节点在父节点 qsmask 位图中的位掩码*/
    int   grplo;            /*本分组中最小 CPU 核编号*/
    int   grphi;            /*本分组中最大 CPU 核编号*/
    u8    grpnum;           /*本节点在上一层分组中的编号*/
    u8    level;            /*本节点所在层级编号，顶层为 0*/
    bool wait_blkd_tasks;     /* Necessary to wait for blocked tasks to */

    struct rcu_node *parent;    /*指向父节点*/
    struct list_head blkd_tasks;    /*阻塞进程双链表，表示在读临界区被其它进程抢占的进程*/
    struct list_head *gp_tasks;   /*指向 blkd_tasks 链表中成员，见下文*/
```

```
    struct list_head *exp_tasks;    /*指向 blkd_tasks 链表中成员，见下文*/
    ...
} ____cacheline_internodealigned_in_smp;
```

rcu_node 结构体中主要成员语义在上面已经注释了，不再重复了。

Tree RCU 层级结构中最底层节点的子节点是 rcu_data 结构体实例，每个 CPU 核对应一个 rcu_data 实例，描述 CPU 核的 RCU 状态。rcu_data 结构体定义如下（/kernel/rcu/tree.h）：

```
struct rcu_data {
    unsigned long completed;    /*上一次结束宽限期编号，初始值为-300（类型为无符号整数）*/
    unsigned long gpnum;        /*当前宽限期编号，初始值为-300*/
    unsigned long rcu_qs_ctr_snap;/* Snapshot of rcu_qs_ctr to check */
    bool        passed_quiesce;     /*记录 CPU 核是否经历了静止状态*/
    bool        qs_pending;     /*表示 CPU 核正在等待静止状态*/
    bool        beenonline;     /* CPU online at least once. */
    bool        gpwrap;         /* Possible gpnum/completed wrap. */
    struct rcu_node *mynode;    /*指向父节点，rcu_node 实例*/
    unsigned long grpmask;      /*在父节点 qsmask 位图中掩码*/
    ...
    struct rcu_head *nxtlist;       /*管理回调函数的链表*/
    struct rcu_head **nxttail[RCU_NEXT_SIZE];
    unsigned long nxtcompleted[RCU_NEXT_SIZE];
    ...
    int cpu;        /*对应 CPU 核编号*/
    struct rcu_state *rsp;      /*指向 rcu_state 实例*/

};
```

rcu_data 结构体主要成员简介如下：

- **completed**：上一次结束宽限期编号。
- **gpnum**：当前宽限期编号。
- **grpmask**：本节点在父节点位图中掩码。
- **passed_quiesce**：记录 CPU 核是否经历了静止状态，内核检测到 CPU 核处于静止状态时置位。
- **mynode**：指向父节点 rcu_node 实例。
- **nxtlist**：指向 rcu_head 实例。
- **nxttail[RCU_NEXT_SIZE]**：指针数组，数组项是 rcu_head 实例指针。
- **nxtcompleted[RCU_NEXT_SIZE]**：无符号整型数组。
- **cpu**：关联 CPU 核编号。
- **rsp**：指向 rcu_state 实例。

■初始化

内核在/kernel/rcu/tree.c 文件内定义了创建 RCU 数据结构实例的宏：

```
#define RCU_STATE_INITIALIZER(sname, sabbr, cr) \       /*cr 为回调函数指针*/
```

```
DEFINE_RCU_TPS(sname) \                 /*没有选择 TRACING 为空*/
static DEFINE_PER_CPU_SHARED_ALIGNED(struct rcu_data, sname##_data); \ /*定义 rcu_data 实例*/
struct rcu_state sname##_state = { \        /*创建并初始化 rcu_state 实例*/
    .level = { &sname##_state.node[0] }, \
    .rda = &sname##_data, \     /*指向 rcu_data 实例*/
    .call = cr, \                           /*注册回调函数的函数*/
    .fqs_state = RCU_GP_IDLE, \
    .gpnum = 0UL - 300UL, \
    .completed = 0UL - 300UL, \
    .orphan_lock = __RAW_SPIN_LOCK_UNLOCKED(&sname##_state.orphan_lock), \
    .orphan_nxttail = &sname##_state.orphan_nxtlist, \
    .orphan_donetail = &sname##_state.orphan_donelist, \
    .barrier_mutex = __MUTEX_INITIALIZER(sname##_state.barrier_mutex), \
    .name = RCU_STATE_NAME(sname), \
    .abbr = sabbr, \
}
```

RCU_STATE_INITIALIZER()宏内同时定义了 rcu_data 实例（percpu 变量）和 rcu_state 实例并初始化。

在/kernel/rcu/tree.c 文件内创建了不可抢占 RCU、加速版不可抢占 RCU 的 rcu_state 实例：

```
RCU_STATE_INITIALIZER(rcu_sched, 's', call_rcu_sched);   /*实例名称 rcu_sched_state*/
RCU_STATE_INITIALIZER(rcu_bh, 'b', call_rcu_bh);   /*实例名称 rcu_bh_state*/

static struct rcu_state *const rcu_state_p;       /*指向经典 RCU 使用的实例*/
static struct rcu_data __percpu *const rcu_data_p;
LIST_HEAD(rcu_struct_flavors);                  /*全局双链表，管理 rcu_state 实例*/
```

在/kernel/rcu/tree_plugin.h 头文件定义了可抢占 RCU 的 rcu_state 实例：

```
RCU_STATE_INITIALIZER(rcu_preempt, 'p', call_rcu);        /*实例名称 rcu_preempt_state*/
static struct rcu_state *const rcu_state_p = &rcu_preempt_state;     /*指向可抢占 RCU 实例*/
static struct rcu_data __percpu *const rcu_data_p = &rcu_preempt_data;
```

只有选择了 PREEMPT_RCU 配置项，才会定义 rcu_preempt_state 实例，以及 rcu_state_p 和 rcu_data_p 全局指针变量，否则不存在这些实例和变量。

如果说内核支持内核抢占 **rcu_state_p** 指向可抢占 RCU 的 rcu_preempt_state 实例，否则指向不可抢占 RCU 的 rcu_sched_state 实例。

内核在/kernel/rcu/tree.c 文件内定义了 **rcu_init()**函数，用于进一步初始化各数据结构实例，函数在内核启动函数中被调用，函数代码如下：

```
void __init rcu_init(void)
{
    int cpu;
```

```
    rcu_early_boot_tests();     /*没有选择 PROVE_RCU 选项为空操作，/kernel/rcu/update.c*/

    rcu_bootup_announce();  /*输出信息，/kernel/rcu/tree_plugin.h*/
    rcu_init_geometry();        /*计算层级结构信息，/kernel/rcu/tree.c*/
    rcu_init_one(&rcu_bh_state, &rcu_bh_data);   /*初始化 rcu_bh_state 实例*/
    rcu_init_one(&rcu_sched_state, &rcu_sched_data);   /*初始化 rcu_sched_state 实例*/
    if (dump_tree)
        rcu_dump_rcu_node_tree(&rcu_sched_state);
    __rcu_init_preempt();
                /*初始化 rcu_preempt_state 实例，调用 rcu_init_one()函数，/kernel/rcu/tree_plugin.h*/
    open_softirq(RCU_SOFTIRQ, rcu_process_callbacks);     /*开启 RCU 软中断*/

    cpu_notifier(rcu_cpu_notify, 0);     /*向 CPU 通知链注册通知，/include/linux/cpu.h*/
    pm_notifier(rcu_pm_notify, 0);       /*向 PM 通知链注册通知*/
    for_each_online_cpu(cpu)     /*对每个在线的 CPU 核调用 rcu_cpu_notify()函数*/
        rcu_cpu_notify(NULL, CPU_UP_PREPARE, (void *)(long)cpu);     /*/kernel/rcu/tree.c*/
}
```

rcu_init()函数的主要工作是调用 rcu_init_one()函数初始化各 rcu_state 实例（含 rcu_node 和 rcu_state 实例），开启 RCU_SOFTIRQ 软中断，软中断执行函数为 rcu_process_callbacks()（后面再详细介绍此函数）。rcu_init()函数最后对每个在线的 CPU 核调用 rcu_cpu_notify()函数，初始化 CPU 核在各 rcu_state 实例中对应的 rcu_data 实例。

rcu_init_one()函数初始化 rcu_state 实例，并调用 rcu_boot_init_percpu_data()函数初始化 rcu_data 实例中部分成员，最后将 rcu_state 实例添加到全局双链表 rcu_struct_flavors。

rcu_cpu_notify()函数用于初始化 CPU 核在各 rcu_state 实例中对应的 rcu_data 实例，rcu_data 实例中部分成员已在 rcu_init_one()函数中调用的 rcu_boot_init_percpu_data()函数中初始化了。

rcu_cpu_notify()函数代码简列如下：

```
int rcu_cpu_notify(struct notifier_block *self,unsigned long action, void *hcpu)
/*action:CPU_UP_PREPARE*/
{
    long cpu = (long)hcpu;
    ...
    switch (action) {
    case CPU_UP_PREPARE:
    case CPU_UP_PREPARE_FROZEN:
        rcu_prepare_cpu(cpu);     /*/kernel/rcu/tree.c*/
        rcu_prepare_kthreads(cpu);
        rcu_spawn_all_nocb_kthreads(cpu);
        break;
        ...
    default:
        break;
```

```
        }
        return NOTIFY_OK;
    }
```

rcu_prepare_cpu(cpu)函数遍历内核中 rcu_state 实例，对每个实例调用 rcu_init_percpu_data()函数初始化 CPU 核在 rcu_state 实例中对应的 rcu_data 实例，函数定义如下（/kernel/rcu/tree.c）：

```
    static void rcu_init_percpu_data(int cpu, struct rcu_state *rsp)
    /*cpu：CPU 核编号，rsp：rcu_state 实例指针*/
    {
        unsigned long flags;
        unsigned long mask;
        struct rcu_data *rdp = per_cpu_ptr(rsp->rda, cpu);     /*rcu_data 实例指针*/
        struct rcu_node *rnp = rcu_get_root(rsp);       /*最顶层 rcu_node 节点指针*/
        ...
        if (!rdp->nxtlist)
            init_callback_list(rdp);    /* Re-enable callbacks on this CPU. */
        ...
        rnp->qsmaskinitnext |= mask;
        rdp->gpnum = rnp->completed;     /*初始值-300*/
        rdp->completed = rnp->completed;
        rdp->passed_quiesce = false;
        rdp->rcu_qs_ctr_snap = per_cpu(rcu_qs_ctr, cpu);
        rdp->qs_pending = false;
        ...
    }
```

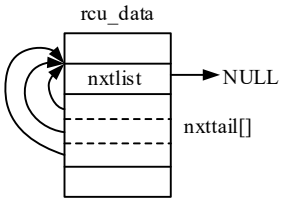
这里我们主要看一下以上函数中调用的 **init_callback_list(rdp)**函数，它用于实初始化 rcu_data ->nxtlist 链表，链表管理着写者注册的释放旧对象的回调函数。

init_callback_list(rdp)函数代码如下：

```
    static void init_callback_list(struct rcu_data *rdp)
    {
        if (init_nocb_callback_list(rdp))     /*没有选择 RCU_NOCB_CPU 配置项，返回 false*/
            return;
        init_default_callback_list(rdp);     /*/kernel/rcu/tree.c*/
    }

    static void init_default_callback_list(struct rcu_data *rdp)
    {
        int i;

        rdp->nxtlist = NULL;
        for (i = 0; i < RCU_NEXT_SIZE; i++)
```

```
        rdp->nxttail[i] = &rdp->nxtlist;       /*指针数组项都指向 rdp->nxtlist 实例*/
    }
```

rcu_data 实例中 nxtlist 成员是指向 rcu_head 实例的指针（初始值为 NULL），nxttail[]是指针数组成员，数组项是指向 rcu_head 实例的指针，初始化结果如下图所示：



■注册回调函数

写者在发布新实例后，需要调用 call_rcu(struct rcu_head *head,void (*func)(struct rcu_head *head))函数注册释放旧实例的函数，在 RCU 软中断中将执行回调函数，释放旧实例。

使用 RCU 机制保护的共享数据，数据结构中通常嵌入 rcu_head 结构体成员，例如表示文件系统超级块的 super_block 结构体：

```
struct super_block {
    ...
    struct rcu_head    rcu;
    ...
};
```

rcu_head 结构体定义在/include/linux/types.h 头文件：

```
#define rcu_head callback_head

struct callback_head {
     struct callback_head *next;     /*指向下一个 callback_head 实例，单链表*/
     void (*func)(struct callback_head *head);     /*回调函数，head 指向本 callback_head 实例*/
};
```

call_rcu(struct rcu_head *head,void (*func)(struct rcu_head *head))函数中，第一个参数 head 指向共享数据实例中内嵌的 rcu_head 结构体成员，与回调函数 func()中 head 参数相同。func()函数是释放旧实例的函数。

如果选择了 PREEMPT_RCU 配置选项，call_rcu()函数将回调函数注册到 rcu_preempt_state 实例，否则注册到 rcu_sched_state 实例，call_rcu_bh()函数将回调函数注册到 rcu_bh_state 实例，详见头文件/include/linux/rcupdate.h。

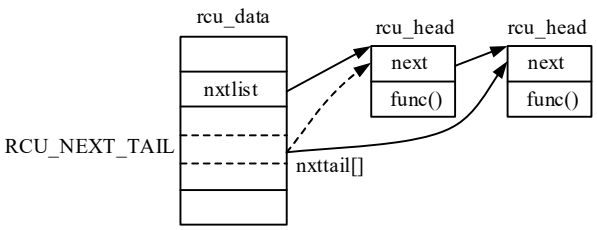
下面以注册到 rcu_preempt_state 实例为例，说明注册回调函数的实现（/kernel/rcu/tree_plugin.h）。

```
void call_rcu(struct rcu_head *head, void (*func)(struct rcu_head *rcu))
{
     __call_rcu(head, func, rcu_state_p, -1, 0);
                              /*rcu_state_p 指向 rcu_preempt_state 实例，/kernel/rcu/tree.c*/
}
```

__call_rcu()函数定义如下（/kernel/rcu/tree.c）：

```
static void __call_rcu(struct rcu_head *head, void (*func)(struct rcu_head *rcu),     \
                                                    struct rcu_state *rsp, int cpu, bool lazy)
/*rsp：指向 rcu_preempt_state 实例，cpu：-1，lazy：0*/
{
    unsigned long flags;
    struct rcu_data *rdp;
    WARN_ON_ONCE((unsigned long)head & 0x1); /* Misaligned rcu_head! */
    ...
    head->func = func;     /*func()函数赋予 callback_head  实例*/
    head->next = NULL;

    local_irq_save(flags);        /*保存中断状态，并中断*/
    rdp = this_cpu_ptr(rsp->rda);     /*本 CPU 核对应的 rcu_data 实例*/

    /*将 rcu_head 实例添加到本 CPU 核 rcu_data 实例中的单链表*/
    if (unlikely(rdp->nxttail[RCU_NEXT_TAIL] == NULL) || cpu != -1) {    /*条件不成立*/
        ...
    }
    ...
    if (lazy)
        rdp->qlen_lazy++;
    else
        rcu_idle_count_callbacks_posted();
    smp_mb();
    *rdp->nxttail[RCU_NEXT_TAIL] = head;     /*指针数组最末尾项指向注册的 rcu_head 实例*/
    rdp->nxttail[RCU_NEXT_TAIL] = &head->next;   /*指向下一实例的指针*/
    ...
    __call_rcu_core(rsp, rdp, head, flags);        /*触发 RCU 软中断等，/kernel/rcu/tree.c*/
    local_irq_restore(flags);          /*恢复中断状态*/
}
```

__call_rcu()函数将 rcu_head 实例添加到当前 CPU 核对应 rcu_data 实例中的回调函数单链表末尾，如下图所示：



__call_rcu()函数最后调用__call_rcu_core()函数触发 RCU 软中断等，函数定义如下（/kernel/rcu/tree.c）：

```
static void __call_rcu_core(struct rcu_state *rsp, struct rcu_data *rdp,    \
```

```
                                        struct rcu_head *head, unsigned long flags)
{
    bool needwake;
    /*CPU 核运行 idle 进程且不在中断中，rcu_is_watching()返回 true*/
    /*CPU 核不在运行 idle 进程，或在中断中，执行 if 内语句*/
    if (!rcu_is_watching())
        invoke_rcu_core();      /*当前 CPU 核在线，触发 RCU_SOFTIRQ 软中断*/

    /*禁止中断或 CPU 核下线，返回*/
    if (irqs_disabled_flags(flags) || cpu_is_offline(smp_processor_id()))
        return;

    /*如果有过多的回调函数或等待时间过长*/
    if (unlikely(rdp->qlen > rdp->qlen_last_fqs_check + qhimark)) {

        /*是否忽略已完成的宽限期*/
        note_gp_changes(rsp, rdp);

        /*开启新的宽限期*/
        if (!rcu_gp_in_progress(rsp)) {
            struct rcu_node *rnp_root = rcu_get_root(rsp);
            raw_spin_lock(&rnp_root->lock);
            smp_mb__after_unlock_lock();
            needwake = rcu_start_gp(rsp);        /*开启新宽限期*/
            raw_spin_unlock(&rnp_root->lock);
            if (needwake)
                rcu_gp_kthread_wake(rsp);       /*唤醒宽限期线程*/
        } else {
            rdp->blimit = LONG_MAX;
            if (rsp->n_force_qs == rdp->n_force_qs_snap &&
                                    *rdp->nxttail[RCU_DONE_TAIL] != head)
                force_quiescent_state(rsp);
            rdp->n_force_qs_snap = rsp->n_force_qs;
            rdp->qlen_last_fqs_check = rdp->qlen;
        }
    }
}
```

call_rcu()注册函数的主要工作是将 head 指向实例关联到 rdp->nxttail[]指针数组最末尾项，触发 RCU 软中断，在软中断处理函数中将调用注册的回调函数销毁旧实例，见下文。

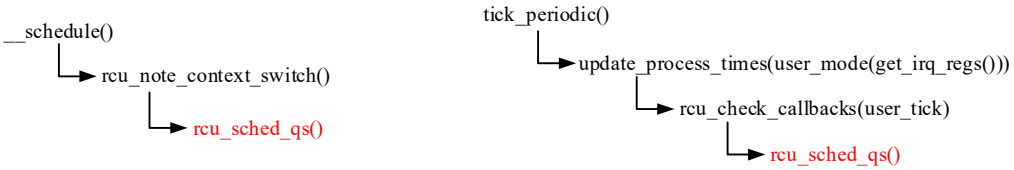
3 记录静止状态

RCU 机制在开启一个新宽限期时，会置位 rcu_state 实例中各节点、CPU 核的位图，CPU 核经历一个静止状态时，会逐级清零标记位，当所有 CPU 核都最少经历一次静止状态时，结束宽限期，开启一个新的宽限期。

下面介绍 RCU 机制中如何记录 CPU 核的静止状态，各类型的 RCU 记录静止状态的方式不同。

■不可抢占 RCU

对于不可抢占 RCU，由于在读临界区中不允许内核抢占，即不会发生进程调度。rcu_read_lock_sched()和 rcu_read_unlock_sched()函数内主要工作就是禁止和使能内核抢占。因此，如果发生了进程调度，则可认为 CPU 核已经离开了读临界区。另外，如果 CPU 核正在用户空间运行或在运行 idle 进程，也可以确定 CPU 离开了读临界区。内核在进程调度和周期节拍工作函数中将会记录不可抢占 RCU 中当前 CPU 核的静止状态，函数调用关系如下图所示：



在周期节拍工作中 rcu_check_callbacks()函数内判断当前 CPU 核如果处于用户态或正在运行 idle 进程且时钟中断处于第一层中断，则将调用 rcu_sched_qs()函数，记录 CPU 核的静止状态。

rcu_sched_qs()函数定义如下（/kernel/rcu/tree.c）：

```
void rcu_sched_qs(void)
{
    if (!__this_cpu_read(rcu_sched_data.passed_quiesce)) {      /*passed_quiesce 为 0*/
      trace_rcu_grace_period(TPS("rcu_sched"),__this_cpu_read(rcu_sched_data.gpnum),TPS("cpuqs"));
      __this_cpu_write(rcu_sched_data.passed_quiesce, 1);      /*置 1*/
    }
}
```

rcu_sched_qs()函数内主要是对 rcu_sched_data.**passed_quiesce** 成员写 1，表示 CPU 核已经历过静止状态，内核并不关心经历了多少次。

■加速版不可抢占 RCU

加速版不可抢占 RCU 在进入和离开读临界区时的主要工作是禁止和使能软中断（同时禁止和使能内核抢占）。内核通过以下事件来确定 CPU 核的静止状态：

（1）执行完软中断。

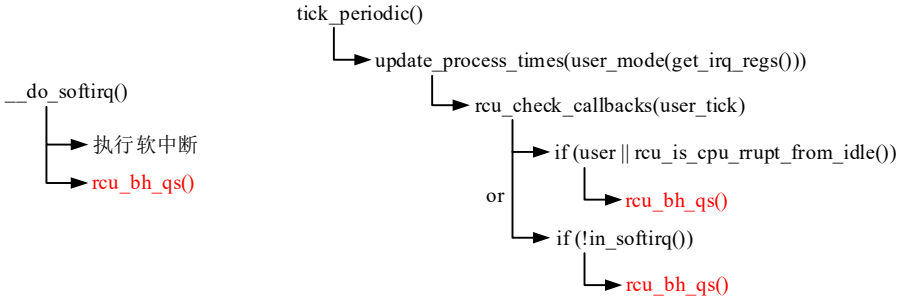
（2）CPU 核在用户空间运行或运行空闲进程（同不可抢占 RCU）。

（3）CPU 核当前不在执行软中断。

（4）内核抢占计数中，软中断计数为 0。因为进入 RCU 读临界区会禁止软中断，软中断计数值加 1，退出时减 1，若为 0，表示不在读临界区。

内核在软中断执行完毕后，将记录 CPU 核静止状态。在周期节拍工作中 rcu_check_callbacks()函数内判断当前 CPU 核如果处于用户态或正在运行 idle 进程且时钟中断处于第一层中断，或者软中断计数值为

0，则将调用 rcu_bh_qs()函数，记录 CPU 核的静止状态。函数调用关系如下图所示：



rcu_bh_qs()函数定义如下（/kernel/rcu/tree.c）：

```
void rcu_bh_qs(void)
{
    if (!__this_cpu_read(rcu_bh_data.passed_quiesce)) {
        trace_rcu_grace_period(TPS("rcu_bh"),__this_cpu_read(rcu_bh_data.gpnum),TPS("cpuqs"));
        __this_cpu_write(rcu_bh_data.passed_quiesce, 1);
    }
}
```

与不可抢占 RCU 类似，函数内主要是将 rcu_bh_data.**passed_quiesce** 成员值置 1。

■可抢占 RCU

可抢占 RCU 允许进程在读临界区被抢占，这给 CPU 核静止状态的记录带来了复杂性。复杂性主要表现在两个方面：一是进程在当前 CPU 核进入临界区后，执行进程调度后，进程可能被迁移到其它 CPU 核运行，而在其它 CPU 核退出临界区，二是读临界区可能嵌套。

可抢占 RCU 观察静止状态的实现方案如下：

（1）不是以处理器为单位观察静止状态，而是以最底层分组为单位观察静止状态。

（2）进程需要记录读临界区的嵌套层数，进入读临界区时嵌套层数加 1，退出读临界区时嵌套层数减 1。当嵌套层数变为 0 时，说明进程已经退出最外层的读临界区。

（3）当进程在读临界区里面被其它进程抢占时，把进程添加到分组的阻塞进程链表中，并且记录分组。

（4）当进程退出最外层读临界区时，把进程从分组的阻塞进程链表中删除。如果分组的阻塞进程链表为空，那么观察到一个静止状态。这里的分组是第 3 步记录的分组，即进入读临界区时的分组，而不是当前的分组，也就是说不管进程有没有迁移，在哪个分组进入临界区，就在那个分组退出临界区。

如果启用了可抢占 RCU，进程 task_struct 结构中增加了以下成员：

```
task_struct{
        ...
    #ifdef CONFIG_PREEMPT_RCU
      int rcu_read_lock_nesting;        /*读临界区嵌套计数值*/
      union rcu_special rcu_read_unlock_special;      /*表示进程状态*/
      struct list_head rcu_node_entry;        /*将进程添加到 rcu_node 实例阻塞进程双链表，见下文*/
```

```
        struct rcu_node *rcu_blocked_node;      /*指向当前 CPU 核所在 rcu_node 实例*/
    #endif    /* #ifdef CONFIG_PREEMPT_RCU */
        ...
}
```

rcu_special 联合体定义在/include/linux/sched.h 头文件：

```
union rcu_special {
    struct {
        bool blocked;
        bool need_qs;
    } b;
    short s;   /*位集合*/
};
```

●blocked：表示进程在读临界区被抢占，阻塞了当前宽限期。

●need_qs：表示进程需要报告正常宽限期的静止状态。

在 RCU 机制最底层的 rcu_node 节点实例中记录了在读临界区阻塞的进程，相关成员定义如下：

```
struct rcu_node {
    ...
    struct list_head blkd_tasks;    /*阻塞进程双链表，表示在读临界区被其它进程抢占的进程*/
    struct list_head *gp_tasks;   /*指向 blkd_tasks 链表中成员*/
    struct list_head *exp_tasks;    /*指向 blkd_tasks 链表中成员*/
    ...
} ____cacheline_internodealigned_in_smp;
```

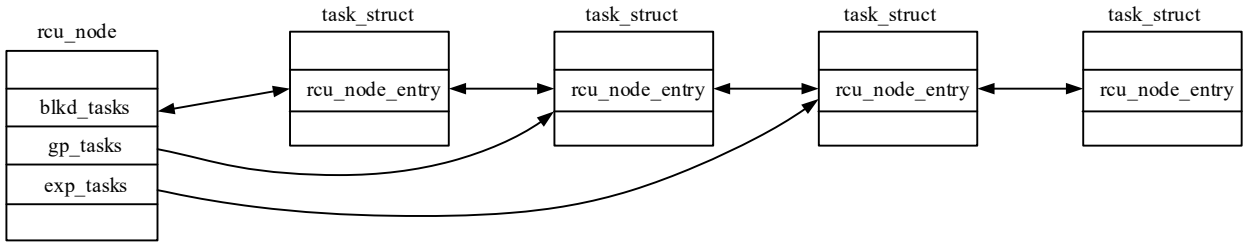
rcu_node 结构体中相关成员语义如下：

●**blkd_tasks**：阻塞进程双链表，表示在读临界区被其它进程抢占的进程。

●**gp_tasks**：指向阻塞当前正常宽限期的第一个进程，从这个节点开始到链表末尾部的所有进程都阻塞当前正常宽限期。

●**exp_tasks**：指向阻塞当前加速宽限期的第一个进程，从这个节点开始到链表末尾部的所有进程都阻塞当前加速宽限期。

可抢占 RCU 阻塞进程双链表结构如下图所示：



从链表首部到 gp_tasks 的前一个链表节点是没有阻塞当前宽限期的进程。

从 gp_tasks 到尾部是阻塞当前正常宽限期的进程。

从 exp_tasks 到尾部是阻塞当前加速宽限期的进程。

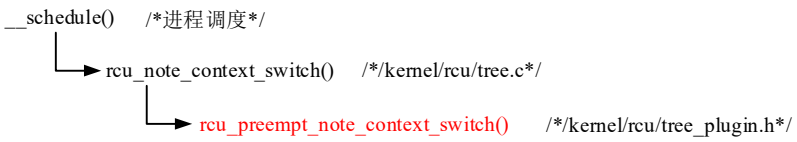
●**进入读临界区**

在读者进入临界区的 rcu_read_lock()函数中将调用__rcu_read_lock()函数将 task_struct 实例中的嵌套计数值加 1，函数定义如下：

```
#ifdef CONFIG_PREEMPT_RCU
void __rcu_read_lock(void)       /*/kernel/rcu/update.c*/
{
    current->rcu_read_lock_nesting++;     /*嵌套计数值加 1*/
    barrier();
}
#endif
```

●**在读临界区被抢占**

当读者在临界区被抢占时，在进程调度函数中会将当前进程添加到 rcu_node 实例阻塞双链表，函数调用关系如图所示：



rcu_preempt_note_context_switch()函数定义在/kernel/rcu/tree_plugin.h 头文件内，代码简列如下：

```
static void rcu_preempt_note_context_switch(void)
{
    struct task_struct *t = current;       /*当前进程，被抢占的进程*/
    unsigned long flags;
    struct rcu_data *rdp;
    struct rcu_node *rnp;

    if (t->rcu_read_lock_nesting > 0 && !t->rcu_read_unlock_special.b.blocked) {
                                          /*当前进程被第一次抢占，将进程添加到阻塞双链表*/
        rdp = this_cpu_ptr(rcu_state_p->rda);        /*当前 CPU 核 rcu_data 实例*/
        rnp = rdp->mynode;       /*当前 CPU 核父节点（rcu_node 实例），进入临界区时所在分组*/
        raw_spin_lock_irqsave(&rnp->lock, flags);
        smp_mb__after_unlock_lock();
        t->rcu_read_unlock_special.b.blocked = true;     /*标记当前进程被抢占*/
        t->rcu_blocked_node = rnp;            /*分组 rcu_node 实例*/
        ...
        if ((rnp->qsmask & rdp->grpmask) && rnp->gp_tasks != NULL) {
            list_add(&t->rcu_node_entry, rnp->gp_tasks->prev); /*添加到 gp_tasks 指向节点前面*/
            rnp->gp_tasks = &t->rcu_node_entry;
            ...
        } else {
            list_add(&t->rcu_node_entry, &rnp->blkd_tasks);     /*添加到阻塞双链表头部*/
            if (rnp->qsmask & rdp->grpmask)
```

```
                rnp->gp_tasks = &t->rcu_node_entry;   /*指向当前进程*/
            }
            ...
            raw_spin_unlock_irqrestore(&rnp->lock, flags);
        } else if (t->rcu_read_lock_nesting < 0 && t->rcu_read_unlock_special.s) {
                                                    /*正在退出最外层临界区*/
            rcu_read_unlock_special(t);      /*将进程从阻塞双链表删除，记录分组静止状态等*/
        }
        rcu_preempt_qs();    /*记录静止状态，/kernel/rcu/tree_plugin.h*/
    }
```

进程在读临界区中第一次被抢占时，将被插入到 rcu_node 实例阻塞进程双链表中，当进程正在从最后一层读临界区中退出，将从阻塞双链表中移除。这里的 rcu_node 实例是进程进入读临界区时所在 CPU 核对应的 rcu_node 实例，而不是迁移之后所在分组的 rcu_node 实例（如果发生了迁移）。

如果阻塞当前进程 A 的进程 B 也在读临界区中被其它进程 C 抢占了，则 B 在阻塞双链表中将插入到进程 A 的前面，依此类推。

rcu_preempt_qs()函数用于记录 CPU 核的静止状态，但是此处似乎意义不大。因为可抢占 RCU 不是以 CPU 核来记录静态状态，而是以分组 rcu_node 来记录静止状态。如果 rcu_node 实例中阻塞链表为空，表示分组处于静止状态。

●退出读临界区

读者退出临界区的 rcu_read_unlock()函数中将调用__rcu_read_unlock()函数，函数定义如下：

```
void __rcu_read_unlock(void)
{
    struct task_struct *t = current;       /*当前进程*/

    if (t->rcu_read_lock_nesting != 1) {     /*嵌套计数不为 1，大于 1*/
        --t->rcu_read_lock_nesting;       /*嵌套计数减 1*/
    } else {            /*如果嵌套计数为 1，表示进程正在退出最外层临界区*/
        barrier();
        t->rcu_read_lock_nesting = INT_MIN;     /*负数，标记正在退出最外层读临界区*/
        barrier();
        if (unlikely(READ_ONCE(t->rcu_read_unlock_special.s)))   /*进程在读临界区被抢占过*/
            rcu_read_unlock_special(t);       /*将进程从阻塞双链表删除，报告分组静止状态等*/
        barrier();
        t->rcu_read_lock_nesting = 0;       /*嵌套计数值清 0*/
    }
    ...
}
```

退出临界区时判断嵌套计数是否为 1，如果不为 1（大于 1）则直接将计数值减 1 即可。如果为 1 表示进程在退出最后一层嵌套，则先将嵌套计数设为负数，将进程从阻塞双链表移除，最后嵌套计数设为 0。这里要把嵌套计数值设为负数的原因是如果在退出最外层嵌套的过程中被抢占，则在进程调度的函数中执

行退出最外层的工作（将进程从阻塞双链表删除，见上文）。

4 RCU 软中断

在 RCU 机制初始化函数中将开启 RCU_SOFTIRQ 软中断，软中断处理函数为 rcu_process_callbacks()。在周期节拍工作中，如果检测到 RCU 机制有挂起的工作将触发 RCU_SOFTIRQ 软中断。

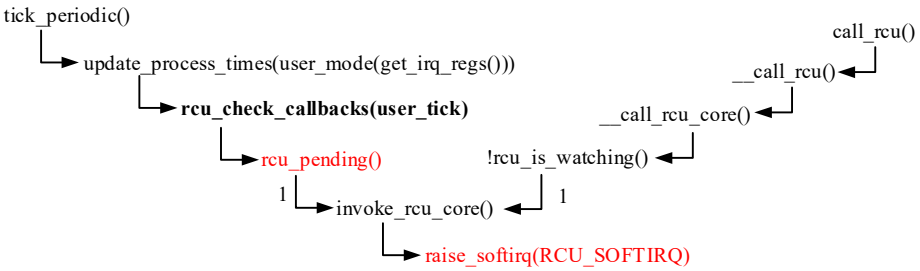
在写者注册回调函数的 call_rcu()等函数中也可能会触发 RCU_SOFTIRQ 软中断。

RCU 软中断主要工作是在 RCU 层级结构中逐级向上报告 CPU 核静止状态，判断是否需要重新开启一个宽限期，如是需要则唤醒宽限期线程初始化新宽限期（结束旧宽限期）。另外，软中断处理函数内还将调用注册的回调函数，释放旧共享数据实例。

■触发软中断

在 RCU 初始化函数 rcu_init()中将开启 RCU 软中断，执行函数为 rcu_process_callbacks()。

在周期节拍工作中将检测当前 CPU 核是否有挂起的 RCU 工作，如果有将触发 RCU_SOFTIRQ 软中断。在注册回调函数的 call_rcu()等函数中，若 CPU 核不在运行 idle 进程，或在处理中断，也将触发 RCU_SOFTIRQ 软中断，函数调用关系如下图所示：



call_rcu()函数前面介绍过了，下面主要介绍周期节拍工作中调用的 rcu_pending()函数，函数定义如下：

```
static int rcu_pending(void)        /*/kernel/rcu/tree.c*/
{
    struct rcu_state *rsp;

    for_each_rcu_flavor(rsp)        /*遍历内核中所有 rcu_state 实例*/
        if (__rcu_pending(rsp, this_cpu_ptr(rsp->rda)))    /*当前 CPU 核对应的 rcu_data 实例*/
            return 1;
    return 0;
}
```

__rcu_pending()函数遍历各类型 rcu_state 实例，调用__rcu_pending()函数检测当前 CPU 核在实例中对应的 rcu_data 实例，看是否有挂起工作，有则返回 1，没有返回 0。只要有一个 rcu_data 实例中有挂起的工作，rcu_pending()函数就返回 1，表示当前 CPU 核有挂起的 RCU 工作。

__rcu_pending()函数定义如下（/kernel/rcu/tree.c）：

```
static int __rcu_pending(struct rcu_state *rsp, struct rcu_data *rdp)
{
    struct rcu_node *rnp = rdp->mynode;
```

```
rdp->n_rcu_pending++;

check_cpu_stall(rsp, rdp);

/*是否是 NO_HZ_FULL CPU，忽略  RCU? */
if (rcu_nohz_full_cpu(rsp))
        return 0;

/*RCU 是否在等待本 CPU 的静止状态*/
if (rcu_scheduler_fully_active &&rdp->qs_pending && !rdp->passed_quiesce &&
        rdp->rcu_qs_ctr_snap == __this_cpu_read(rcu_qs_ctr)) {
        rdp->n_rp_qs_pending++;
} else if (rdp->qs_pending &&
                        (rdp->passed_quiesce ||rdp->rcu_qs_ctr_snap != __this_cpu_read(rcu_qs_ctr))) {
        rdp->n_rp_report_qs++;
        return 1;         /*当前 CPU 核经历了静止状态，返回 1*/
}

/*CPU 是否有注册的回调函数，还没有经历静止状态*/
if (cpu_has_callbacks_ready_to_invoke(rdp)) {
        rdp->n_rp_cb_ready++;
        return 1;             /*返回 1，有注册的回调函数*/
}

/*是否需要启动新的宽限期*/
if (cpu_needs_another_gp(rsp, rdp)) {
        rdp->n_rp_cpu_needs_gp++;
        return 1;
}

/*是否有另一个宽限期完成了*/
if (READ_ONCE(rnp->completed) != rdp->completed) {    /* outside lock */
        rdp->n_rp_gp_completed++;
        return 1;
}

/*是否开始了新的宽限期? */
if (READ_ONCE(rnp->gpnum) != rdp->gpnum ||
        unlikely(READ_ONCE(rdp->gpwrap))) { /* outside lock */
        rdp->n_rp_gp_started++;
        return 1;
}
```

```
    /* Does this CPU need a deferred NOCB wakeup? */
    if (rcu_nocb_need_deferred_wakeup(rdp)) {
        rdp->n_rp_nocb_defer_wakeup++;
        return 1;
    }

    /*没有挂起的工作*/
    rdp->n_rp_need_nothing++;
    return 0;          /*返回 0*/
}
```

__rcu_pending()函数需要做许多的检测工作，如果有挂起的 RCU 工作将返回 1，否则返回 0。

如果 rcu_pending()函数返回 1，在周期节拍工作中（rcu_check_callbacks()函数内）将触发 RCU 软中断。

■软中断处理函数

RCU_SOFTIRQ 软中断处理函数为 **rcu_process_callbacks()**。软中断处理函数中主要是处理 RCU 挂起的工作，如报告静止状态、判断是否要开启新宽限期（是则唤醒宽限期线程）、执行注册的回调函数等。

软中断处理函数 rcu_process_callbacks()定义如下（/kernel/rcu/tree.c）：

```
static void rcu_process_callbacks(struct softirq_action *unused)
{
    struct rcu_state *rsp;

    if (cpu_is_offline(smp_processor_id()))
        return;
    trace_rcu_utilization(TPS("Start RCU core"));
    for_each_rcu_flavor(rsp)     /*遍历各类型 rcu_state 实例*/
        __rcu_process_callbacks(rsp);
    trace_rcu_utilization(TPS("End RCU core"));
}
```

软中断处理函数遍历 RCU 机制中每个 rcu_state 实例调用__rcu_process_callbacks(struct rcu_state *rsp)函数，函数定义如下：

```
static void __rcu_process_callbacks(struct rcu_state *rsp)
{
    unsigned long flags;
    bool needwake;
    struct rcu_data *rdp = raw_cpu_ptr(rsp->rda);

    rcu_check_quiescent_state(rsp, rdp);
                /*在层级结构中逐级上报 CPU 核静止状态（标记位图清零且阻塞双链为空）*/
```

```
        local_irq_save(flags);
        if (cpu_needs_another_gp(rsp, rdp)) {       /*是否需要启动新的宽限期*/
            raw_spin_lock(&rcu_get_root(rsp)->lock);
            needwake = rcu_start_gp(rsp);       /*判断是否需要唤醒宽限期线程*/
            raw_spin_unlock_irqrestore(&rcu_get_root(rsp)->lock, flags);
            if (needwake)
                rcu_gp_kthread_wake(rsp);     /*唤醒宽限期线程 wake_up(&rsp->gp_wq);，见下文*/
        } else {
            local_irq_restore(flags);
        }

        /*处理注册的回调函数*/
        if (cpu_has_callbacks_ready_to_invoke(rdp))     /*回调函数链表不为空*/
            invoke_rcu_callbacks(rsp, rdp);   /*执行回调函数，销毁旧共享数据实例，head->func(head)*/

        do_nocb_deferred_wakeup(rdp);
    }
```

__rcu_process_callbacks()函数调用 rcu_check_quiescent_state()函数报告静止状态，如果需要开启新宽限期，将唤醒宽限期线程。

调用 invoke_rcu_callbacks()函数用于执行回调函数，若没有选择 RCU_BOOST 配置选项，则此函数内调用 rcu_do_batch()函数执行注册的回调函数释放旧实例。若选择了 RCU_BOOST 配置选项，则由内核线程执行回调函数释放旧实例，详见/kernel/rcu/tree_plugin.h 头文件。

5 宽限期线程

内核在启动阶段为每个 rcu_state 实例都创建了一个宽限期线程，专门用于启动新宽限期和结束当前宽限期。内核创建宽限期线程的函数定义如下（/kernel/rcu/tree.c）：

```
    static int __init rcu_spawn_gp_kthread(void)
    {
        ...
        for_each_rcu_flavor(rsp) {       /*遍历 rcu_state 实例*/
            t = kthread_create(rcu_gp_kthread, rsp, "%s", rsp->name);     /*创建内核线程*/
            BUG_ON(IS_ERR(t));
            rnp = rcu_get_root(rsp);
            raw_spin_lock_irqsave(&rnp->lock, flags);
            rsp->gp_kthread = t;          /*指向宽限期线程*/
            if (kthread_prio) {
                /*线程优先级，受 RCU_KTHREAD_PRIO、RCU_BOOST 选项或模块参数等控制*/
                sp.sched_priority = kthread_prio;
                sched_setscheduler_nocheck(t, SCHED_FIFO, &sp);
            }
            wake_up_process(t);          /*唤醒线程*/
```

```
            raw_spin_unlock_irqrestore(&rnp->lock, flags);
        }
        rcu_spawn_nocb_kthreads();
        rcu_spawn_boost_kthreads();          /*创建执行回调函数的内核线程（RCU_BOOST）*/
        return 0;
    }
    early_initcall(rcu_spawn_gp_kthread);       /*内核启动后期初始化子系统时调用此函数*/
```

宽限期线程执行函数 rcu_gp_kthread()代码简列如下（/kernel/rcu/tree.c）：

```
static int __noreturn rcu_gp_kthread(void *arg)
{
    int fqs_state;
    int gf;
    unsigned long j;
    int ret;
    struct rcu_state *rsp = arg;
    struct rcu_node *rnp = rcu_get_root(rsp);

    rcu_bind_gp_kthread();
    for (;;) {

        /*处理宽限期开始*/
        for (;;) {
            ...
            rsp->gp_state = RCU_GP_WAIT_GPS;
            wait_event_interruptible(rsp->gp_wq,
                        READ_ONCE(rsp->gp_flags) &RCU_GP_FLAG_INIT);
            if (rcu_gp_init(rsp))            /*初始化新宽限期*/
                break;
            ...
        }

        /*处理强制宽限期*/
        ...
        /*处理宽限期结束*/
        rcu_gp_cleanup(rsp);
    }
}
```

6 SRCU

可抢占 RCU 读临界区允许抢占，但不允许睡眠。可睡眠 RCU（SRCU）允许在读临界区里面睡眠。

在读临界区睡眠可能导致宽限期很长。为了避免影响整个系统，使用 SRCU 的子系统需要定义一个 SRCU 域，每个 SRCU 域有自己的读端临界区和宽限期。每个使用 SRCU 的子系统需要定义和初始化相关的数据结构。

前面介绍的 RCU 其 rcu_state 实例是对所有受控的共享数据有效的，而 SRCU 的数据结构只对某个子系统的共享数据有效。

SRCU 数据结构为 srcu_struct，定义在/include/linux/srcu.h 头文件，SRCU 实现代码位于/kernel/rcu/srcu.c 文件内。

SRCU 机制接口函数如下：

●DEFINE_SRCU(name)：定义并初始化 srcu_struct 实例。

●init_srcu_struct(struct srcu_struct *sp)：初始化已定义 srcu_struct 实例。

●**idx**=srcu_read_lock(struct srcu_struct *sp)：进入读临界区，sp 指向 srcu_struct 实例，返回索引值。

●read=srcu_dereference(p, sp)：获取 SRCU 保护的指针。

●srcu_read_unlock(struct srcu_struct *sp, int **idx**)：离开读临界区。

●call_srcu(struct srcu_struct *sp, struct rcu_head *head,void (*func)(struct rcu_head *head))：写者注册回调函数。

●srcu_barrier(struct srcu_struct *sp)：等待所有回调函数执行完毕。

●synchronize_srcu(struct srcu_struct *sp)：写者同步等待宽限期结束。

●synchronize_srcu_expedited(struct srcu_struct *sp)：写者同步等待加速宽限期结束。

SRCU 相关数据结构及函数代码请读者自行阅读。

由于作者水平有限，对 RCU 机制只进行了简要的介绍。

6.10 小结

本章介绍了内核一些“杂项”的工作，包括异常处理（含 TLB 异常、系统调用、中断等）、内核中断管理、软中断、工作队列、时间管理、并发与同步控制等，这些工作对系统的正常运行至关重要。

MIPS 体系结构主要的异常有：TLB 异常、系统调用、中断等。TLB 异常主要工作是利用内存页表项填充 TLB，以建立进程虚拟内存与物理内存的映射关系。系统调用是内核与用户进程唯一的正常的接口，进程通过系统调用使用内核提供的功能。中断用于处理外部设备产生的中断。在异常处理程序返回前夕将执行内核工作，如进程调度、处理挂起信号等。

内核建立了通用的中断管理结构，驱动程序需向内核注册中断及其处理函数。中断处理中的工作通常分为上半部和下半部，上半部执行紧急的工作，直接在中断处理程序中执行，下半部执行不那么紧急的工作，可在软中断或工作队列中执行。

软中断和工作队列都可以用于执行中断下半部工作。软中断可能在中断处理程序中调用，会延迟中断的返回，影响系统实时性，且软中断中不能睡眠，禁止内核抢占。工作队列是通用的执行内核工作的机制，在进程上下文中执行，因此不受任何影响，可抢占可睡眠。

时间管理部分的主要功能是向内核提供时间计数值（计时器）和向 CPU 核提供时钟设备。计时器和时钟设备都基于时钟硬件实现。计时器是公共的资源，所有进程都可从中获取时间值。每个 CPU 核都有自己的时钟设备，时钟设备利用了时钟硬件可定时产生中断的特性，在中断处理函数中执行特定的内核工作。

内核节拍就是由时钟设备产生的，在节拍中将执行周期性的内核工作，如周期性调度器、处理到期定时器等等。时钟设备根据其工作模式分为低分辨率周期时钟、低分辨率动态时钟和高分辨率时钟等。动态

时钟是指 CPU 核在无事可做时（运行 idle 进程时）暂停周期时钟，以节省电力。

内核锁机制是保护内核共享数据一致性（同步）的措施，是内核能够正常运行的重要保证。内核锁机制包括原子变量、自旋锁、顺序锁、信号量、互斥量、RCU 机制等，内核代码中根据不同的场景采用不同的锁机制保护共享数据。