Linux 内核中的 Bottom Half 机制分析与应用

Analysis and Application of Bottom Half Mechanism in Linux Kernel

李旭芳/LI Xu-fang

上海工程技术大学管理学院信息管理系,上海 200336

School of Management, Shanghai University of Engineering Science, Shanghai 200336, China

[摘要] 针对中断服务程序的执行,重点讨论了 Linux 内核中的 Bottom Half 机制,介绍了从 Linux2.0 到 Linux2.6 内核中所实现的 BH 函数接口、任务队列(task queue)、tasklet、软中断、工作队列(work queues)等 Bottom Half 机制,并分别从实现原理和具体使用方法两个方面进行了对比分析。

[关键词] 中断机制;Linux;Bottom Half 机制 [中图分类号] TP333

[文献标识码] A [文章编号] 1000-7857(2006)05-0066-05

Abstract: The Bottom Half mechanism in Linux kernel is discussed in the paper for the realization of interrupt programs. Furthermore, the Bottom Half, including BH function interface, task queue, tasklet, softirq, and work queues, which were evolved from linux2.0 to linux2.6, is analyzed in detail from the two aspects of realization theory and application.

Key Words: interrupt mechanism; Linux; Bottom Half mechanism **CLC Number:** TP333

Document Code: A **Article ID:** 1000-7857(2006)05-0066-05

1 引言

Linux 是类 UNIX 操作系统,是多用户、多任务的操作系统,具有良好的开放性和可靠性,在 Linux 平台下开发数控系统具有重要意义[1]。数控系统的多任务性和实时性[2],决定了系统中断成为整个系统必不可少的重要组成部分。那么,Linux 是如何执行中断服务的呢?

为了避免中断嵌套而使控制复杂,中断服务一般都是在将中断请求关闭的条件下执行的,但这样做存在的问题是,如果在中断服务执行的全过程,关闭中断则使 CPU 不能响应其它的中断请求;如果在全过程开启中断,则又可能造成中断嵌套(单 CPU 下中断嵌套执行,多 CPU 下并发执行同一中断)。Linux 将中断服务程序一分为二[3],各称作"Top Half"和"Bottom Half"。Top Half 通常对时间要求较为严格,必须在中断请求发生后立即或至少在一定的时间限制内完成。为了保证这种处理能圆满地完成,Top Half 通常是在 CPU 关闭中断的条件下执行的。Top Half 的范围包括:从在 IDT 中登记的中断入口函数一直到驱动程序注册在中断服务队列中的 ISR。Bottom Half 则是 Top Half 根据需要来调度执行的,这些操作允许延迟到稍后执行,它的时间要求并不严格,因此它通常是在 CPU 开启中断的条件下执行的。但是,Linux 早期实现的 Bottom Half 机制 BH 函数接口有 2 个缺点:① BH 代码的执行是严格"串行化"的,在任意一时刻,系统只能有 1 个 CPU 可以执行 Bottom Half 代码,以防止 2 个或多个 CPU 同时来执行 BH 函数而相互干扰;② BH 函数不允许嵌套。这 2 个缺点在单 CPU 系统中是无关紧要的,但在 SMP 系统中却是非常致命的,因为 BH 函数机制的严格串行化执行显然没有充分利用 SMP 系统的多 CPU 特点。为此,Linux2.4 内核在 BH 函数机制的基础上进行了扩展,这就是所谓的"软中断请求"(softirq)机制。Linux 2.6 内核在保留 softirq 和 tasklet 机制的同时,还提供了一个新的 Bottom Half 机制-work queues。

2 BH 函数接口

从 Linux 2.0 开始便实现了 BH 函数接口,它设置了一个函数指针数组 bh_base[32],数组中的每个指针可以用来指向一个具体的 BH 函数,同时又设置了 2 个 32 位无符号整数 bh_active 和 bh_mask,每个无符号整数中的 32 位对应着数组 bh_base[ ]中的 32 个元素,用以确定是否开启/关闭 BH 函数。当执行某硬件中断服务程序时,如果需要执行一个 BH 函数,硬件中断服务的 Top Half 只需通过函数 mark_bh( )将 bh_active 中的相应位设成 1,然后返回,剩下的工作交给相应的 BH 来处理。当内核每次执行完 do_IRQ()中的中断服务程序之后,或者每次系统调用结束之时,函数 do_bottom_half ( ) 得到触发,它将执行相应的 bh_active 和 bh_mask 相应位都为 1 的 BH 函数。

BH 函数接口实现简单,但有严重的缺陷,最重要的 2 个就是:① 严格串行化不能利用 SMP;② 只有 32 个 BH 函数不够用。为此,在 2.0 内核的后续版本中提出了任务队列来改进。

3 任务队列(task queue)

3.1 任务队列实现

BH 的一个主要缺陷是只有 32 个 BH 函数,不够用。为此,早期的 2.0 内核后续版本又对 BH 进行了扩展,引入了 task queue。

收稿日期:2005-10-26

作者简介:李旭芳,女,上海市仙霞路 350 号上海工程技术大学管理学院信息管理系,助教,研究方向为计算机软件;
E-mail: lucylxf@163.com

它是一个双向队列链表,其结构如下:

```
struct tq_struct {
    struct list_head list;          /* 链表结构 */
        unsigned long sync;             /* 必须初始化为 0 */
        void (*routine)(void *);        /* 激活时调用的函数 */
        void *data;                     /* 传递给 routine 的参数 */
}。
```

队列的每个元素都有一个函数指针及其参数,它是由run_task_queue(task_queue *list)函数来调用的,其中 typedef struct list_head task_queue,这样 run_task_queue( )指定了运行哪个任务队列。可以手动调用 run_task_queue 来启动任务,或者将该函数以函数指针的形式挂接到 bh_base[32]数组来启动,这样便扩展了每个 BH 向量的函数个数,以前 bh_base[32]中的每个元素指向一个函数,现在通过 run_task_queue( )函数,使用 bh_base[32]中的每个元素可以调用一个函数链表。

3.2 使用任务队列

实现一个自己的任务队列可以采用如下的步骤进行。

1) 声明一个任务队列 my_queue,DECLARE_TASK_QUEUE (my_tqueue)。

2) 定义一个自己需要执行的任务结构,如 tq_stuct 结构的变量 my_task,并赋值好调用的函数指针及参数。

3) 将任务注册到 my_queue 中,queue_task (&my_task, &my_tqueue)。

4) 在适当的时机启动任务队列的执行。有两种方法:一种是将任务队列的运行纳入到 BH 函数机制中,一种是手动调用 run_task_queue(&my_tqueue)。

4 tasklet 机制

4.1 tasklet 机制原理

引入任务队列解决了 BH 的第一个问题——32 个 BH 函数不够用,但它仍然不能解决严格串行化带来的不能利用 SMP 的问题。为此,Linux 2.4 内核又引入了 tasklet 机制,使得可以让不同的 tasklet 并发运行在不同的 CPU 上,从而提高了 CPU 的利用率。tasklet 结构定义如下。

```
struct tasklet_struct{
    struct tasklet_struct *next;    /* 队列指针 */
    unsigned long state;    /* tasklet 的状态位,目前定义了两个位的含义 */
    atomic_t count;    /* 引用计数,通常用 1 表示 disabled */
    void (*func)(unsigned long);  /* 函数指针 */
    unsigned long data;           /* func 函数调用参数 */
}。
```

把上面的结构与 tq_struct 比较,可以看出,tasklet 主要扩充了 2 点。① state 字段,定义了这个 tasklet 的当前状态,这是个 32 位的无符号长整数,当前只使用了 bit [1](TASKLET_STATE_RUN)和 bit[0](TASKLET_STATE_SCHED)2 个状态位。其中,bit[1]=1 表示这个 tasklet 当前正在某个 CPU 上被执行,它仅对 SMP 系统才有意义,其作用就是为了防止多个 CPU 同时执行一个 tasklet 的情形出现,bit [0] = 1 表示这个 tasklet 已经被调度并等待执行。② count 字段,对这个 tasklet 的引用计数值,只有当 count=0 时,tasklet 代码段才能执行,即此时 tasklet 才被启用,如果 count≠0,则这个 tasklet 是被禁止的。任何想要执行一个 tasklet 代码段的人都首先必须先检查其 count 成员是否为 0,这样便阻止了同一 tasklet 在同一个 CPU 上被嵌套调用。

多个 tasklet 可以通过 tasklet_struct 描述符中的 next 成员指针链接成一个单向对列。为此,Linux 专门定义了数据结构 tasklet_head 来描述一个 tasklet 队列的头部指针。如下所示:

```
struct tasklet_head{
    struct tasklet_struct *list;
}_attribute_ ((_aligned_(SMP_CACHE_BYTES)))。
```

tasklet 机制是软中断对 BH 的扩展,属于软中断机制的整体框架范围内,下面将介绍的 HI_SOFTIRQ 和 TASKLET_SOFTIRQ 软中断向量就是采用 tasklet 来实现的。Linux 为每一个 CPU 都定义了一个 tasklet 队列头,用来表示每个 CPU 负责执行的 tasklet 队列,如下

(kernel/softirq.c):

```
struct tasklet_head tasklet_vec[NR_CPUS] _cacheline_aligned;
struct tasklet_head tasklet_hi_vec [NR_CPUS] _cacheline_aligned。
```

其中,NR_CPUS 代表 CPU 的个数,tasklet_vec[ ]数组用于软中断向量 TASKLET_SOFTIRQ,而 tasklet_hi_vec[ ]数组则用于软中断向量 HI_SOFTIRQ。如果 CPUi(0≤i≤NR_CPUS-1)触发了软中断向量 TASKLET_SOFTIRQ,那么 tasklet_vec[i]队列中的每一个 tasklet 都将在 CPUi 服务于软中断向量 TASKLET_SOFTIRQ 时被 CPUi 所执行。同样,如果 CPUi(0≤i≤NR_CPUS-1)触发了软中断向量 HI_SOFTIRQ,那么 tasklet_hi_vec[i]队列中的每一个 tasklet 都将 CPUi 在对软中断向量 HI_SOFTIRQ 进行服务时被 CPUi 所执行。至于队列 tasklet_vec[i]和 tasklet_hi_vec[i]中的各个 tasklet 是如何被 CPUi 触发执行的,在第 4 小节将描述,其关键是软中断向量 TASKLET_SOFTIRQ 和 HI_SOFTIRQ 的软中断服务程序——tasklet_action( )函数和 tasklet_hi_action( )函数。

4.2 使用 tasklet 机制

使用 tasklet 机制可以采取如下步骤。

1) 定义一个在软中断机制中想要执行的函数,如 void my_tasklet_func(unsigned long)。

2) 定义一个 tasklet 结构 my_tasklet,并将您需要处理的函数与它相关联,如 DECLARE_TASKLET(my_tasklet,my_tasklet_func, data),其中的 data 为函数的调用参数。

3) 将 tasklet 加入到软中断处理队列中,采用如下方法:tasklet_schedule(&my_tasklet)。

5 软中断(softirq)

5.1 软中断原理

Linux2.4 内核开始实现了一个统一的软中断体系框架,它将老的 BH、tasklet 等都集成到了这个框架中。Linux 定义了结构 softirq_action,用来描述一个软中断描述符,如下所示:

```
struct softirq_action{
    void (*action)(struct softirq_action *);
    void *data;
}。
```

其中,函数指针 action 指向软中断请求的服务函数。基于上述软中断描述符,Linux 在 kernel/softirq.c 文件中定义了一个全局的 softirq_vec[32]数组:static struct softirq_action softirq_vec[32]。这样系统一共定义了 32 个软中断请求描述符。软中断向量 i (0≤i≤31)所对应的软中断请求描述符就是 softirq_vec[i]。这个数组是个系统全局数组,即它被所有的 CPU 所共享。每个 CPU 虽然都有它自己的触发和控制机制,并且只执行它自己所触发的软中断请求,但是各个 CPU 所执行的软中断服务例程却是相同的,即都是执行 softirq_vec[]数组中定义的软中断服务函数。

为了实现"谁触发,谁执行"的思想,需要为每个 CPU 都定义它自己的触发和控制变量。为此,Linux 定义了数据结构 irq_cpustat_t 来描述一个 CPU 的软中断统计信息,其数据结构定义如下:

```
typedef struct {
```

```
        unsigned int __softirq_active;
        unsigned int __softirq_mask;
        unsigned int __local_irq_count;
        unsigned int __local_bh_count;
        unsigned int __syscall_count;
        unsigned int __nmi_count; /* arch dependent */
} __cacheline_aligned irq_cpustat_t。
```

结构中 _softirq_active 和 _softirq_mask 就是用于触发和控制软中断的成员变量。_softirq_active 变量是一个 32 位的无符号整数,表示 softirq_vec[32]中软中断向量 0~31 的状态,如果 bit[i](0≤i≤31)为 1,则表示软中断向量 i 在某个 CPU 上已经被触发而处于活跃状态;为 0 表示处于非活跃状态。__softirq_mask 变量也是一个 32 位的无符号整数,表示软中断向量的屏蔽掩码,如果 bit[i](0≤i≤31)为 1,则表示开启(enable)软中断向量 i,为 0 表示该软中断向量被禁止(disabled)。根据系统中当前的 CPU 个数(由宏 NR_CPUS 表示),Linux 为每个 CPU 都定义了它自己的中断统计信息结构,如下所示:irq_cpustat_tirq_stat[NR_CPUS]。这样,每个 CPU 都只操作它自己的中断统计信息结构。假设有一个编号为 i 的 CPU,那么它只能操作中断统计信息结构 irq_stat[i](0≤id≤NR_CPUS-1),从而使各 CPU 之间互不影响。

软中断的触发:函数 _CPU_raise_softirq(int cpu, int nr)用于在编号为 CPU 的处理器上触发软中断向量 nr。它通过将相应 irq_stat[cpu]的 _softirq_active 成员变量中的相应位设置为 1 来实现软中断触发。

软中断的执行:do_softirq()函数负责软中断的执行,它首先判断当前 CPU i 是否已经处在软中断服务中,如果没有,则根据 irq_stat [ ] 的 __softirq_active 和 __softirq_mask 的值来调用 softirq_vec[]的软中断服务。softirq_vec[ ]最多有 32 个元素,也就是说最多有 32 个软中断服务例程,这样岂不是和 BH 一样?这时就要引用到上面所描述的 tasklet 机制了。在 2.4 内核中,do_softirq( )有4 个执行时机,分别是从系统调用中返回、从异常中返回、从调度程序中以及处理完硬件中断之后返回。do_softirq 将遍历所有的 softirq_vec[ ],依次调用其中的 action( )函数。

5.2 软中断框架下的 tasklet

在软中断向量 0~31 中,Linux2.4 内核仅仅使用了软中断向量 0~3,其余被留待系统以后扩展:

```
enum{
    HI_SOFTIRQ=0,
    NET_TX_SOFTIRQ,
    NET_RX_SOFTIRQ,
    TASKLET_SOFTIRQ
}。
```

重点来看 HI_SOFTIRQ 和 TASKLET_SOFTIRQ,它们均为采用 tasklet 机制实现的软中断向量,Linux 不鼓励用户扩展使用其它未使用的软中断向量,它认为这 2 个已经能够满足绝大多数用户的需要了。在软中断框架的初始化函数 softirq_init( )中,调用 open_softirq ( ) 函数启用软中断向量 TASKLET_SOFTIRQ 和 HI_SOFTIRQ,并将它们的软中断服务函数指针分别指向 tasklet_action( )函数和 tasklet_hi_action( )函数。

Linux 为软中断向量 HI_SOFTIRQ 和 TASKLET_SOFTIRQ 实现了专门的触发和执行函数:其中,tasklet_schedule (struct tasklet_struct *t)函数和 tasklet_hi_schedule(struct tasklet_struct *t)函数分别用来在当前 CPU i 上触发软中断向量 TASKLET_SOFTIRQ 和 HI_SOFTIRQ,并把指定的 tasklet 加入当前 CPU i 所对应的 tasklet_vec[i]或 tasklet_hi_vec[i]队列中去等待执行;而 tasklet_action( ) 函数和 tasklet_hi_action( )函数则分别是软中断向量 TASKLET_SOFTIRQ 和 HI_SOFTIRQ 的软中断服务函数,它们是软中断与 tasklet 之间的桥梁。这样,当 do_softirq( )调用 softirq_vec[TASKLET_SOFTIRQ]和 softirq_vec[HI_SOFTIRQ]的软中断服务时,执行的就是 tasklet_action 和 tasklet_hi_action 函数,由这 2 个函数负责将当前 CPU i 的 tasklet_vec [i]或 tasklet_hi_vec[i]队列上的各个 tasklet 放到当前 CPU i 上来执行。

5.3 软中断框架下的 BH

Linux2.4 内核实现了软中断框架后,为了兼容,它并没有抛弃老的 BH 机制,而是通过 tasklet 这个桥梁,以一种新的实现方式将 BH 集成到软中断框架下来[4]。在新的软中断框架下,老的 bh_base[32]函数指针数组仍然被保留,并且增加了一个元素类型为 tasklet_struct 的 bh_task_vec[32]数组与 bh_base 一一对应。由于 BH 函数同一时刻只能在一个 CPU 上执行,为此,2.4 内核新定义了一个全局的自旋锁 global_bh_lock 来保护 BH 函数。

在软中断框架的初始化函数 softirq_init ( ) 中,它将 bh_task_vec[32]的每一个 tasklet_struct 中的 func 函数指针指向同一个函数 bh_action ( ),将 tasklet_struct 的 data 字段设置为该 tasklet_struct 在 bh_task_vec 数组中的索引,并以参数的形式将其传给 bh_action 函数,这样 bh_action( )将根据参数调用 bh_base [32]中的指定 BH 函数,即 bh_action(nr)将调用 bh_base[nr]的函数指针,该函数是连接 tasklet 机制和 Bottom Half 的桥梁。

在软中断框架中,实际是通过 HI_SOFTIRQ 这个软中断向量来执行 BH 函数的。首先,我们来看触发,在新的软中断框架下,mark_bh(int nr)被实现为调用 tasklet_hi_schedule(bh_tasklet_vec+nr),在 tasklet_hi_schedule 函数中,bh_tasklet_vec[nr]将被挂接在当前 CPUi 的 tasklet_hi_vec[i]链上(其中 i 为当前 CPU 编号,也就是说哪个 CPU 提出了 bottom half 的请求,则在哪个 CPU 上执行该请求),然后发出 HI_SOFTIRQ 软中断信号,从而在 HI_SOFTIRQ 的中断响应中启动运行。然后再来看执行,do_softirq()->softirq_vec[HI_SOFTIRQ](即 tasklet_hi_action( )函数)->循环调用当前 CPU i 的 tasklet_hi_vec[i]链中的每一个 func( ),这样被挂接到该链上的 bh_tasklet_vec[nr]被触发执行->bh_action (nr)->调用 bh_base[nr]的函数指针。

6 工作队列(work queues)

前面描述的 Bottom Half 机制有一个共同的特点,它们都是在中断上下文环境中运行。为了适应更灵活的需求,从 Linux 2.6 内核开始提出了一种新的 Bottom Half 机制——工作队列(work queues),它和以往的 Bottom Half 机制之间最大的差异就在于它能够以内核线程的形式运行在可调度的进程上下文环境中。当需要在驱动的延迟操作中分配大的内存、获得信号,或执行阻塞 I/O 操作时,使用工作队列将是非常有用的。如果不需要使用内核线程来执行延迟操作,则可以考虑使用 tasklet 来代替。

6.1 工作队列机制实现

工作队列使用内核线程来执行驱动中需延迟执行的工作(Bottom Half),这些内核线程被称为工作线程。在 Linux2.6 内核中,系统除了提供默认线程来帮助组驱动方便的执行延迟操作外,还允许驱动自己产生自定义的工作线程来执行某些特殊的延迟工作。

默认工作线程被称作 events/n,其中 n 为 CPU 个数,也就是说,每个 CPU 都有一个默认工作线程,如在单 CPU 系统中有一个默认工作线程 events/0。一般情况下,大多数驱动都使用默认工作线程来执行自己的 Bottom-Half 工作。某些情况下,驱动产生自己的自定义工作线程可以满足更高的性能要求,并可以减轻默认工作线程的负担。

workqueue_struct 代表工作线程,其数据结构[5]定义如下:

```
/*
 * The externally visible workqueue abstraction is an array of
```

```
 * per-CPU workqueues:
 */
struct workqueue_struct {
        struct CPU_workqueue_struct CPU_wq[NR_CPUS];
        const char *name;
        struct list_head list;
};
```

每种类型的工作线程都有一个这样的结构与其关联,它有一个重要的成员 CPU_wq,即元素类型为 CPUworkqueue_struct 的数组,表示系统中的每个 CPU 都有自己的工作线程,假设需要在有 2 个 CPU 的计算机上创建类型为 myworker 的工作线程,则系统除了有 2 个类型为 events 的默认工作线程外,还有 2 个类型为 myworker 的工作线程,每一个 events 或 myworker 都有一个 CPU_workqueue_struct 结构与之关联。events 和 myworker 类型定义如下:

```
struct workqueue_struct *keventd_wq;
keventd_wq = create_workqueue("events");
struct workqueue_struct *myworker;
myworker = create_workqueue("myworker");
```

CPU_workqueue_struct 数据结构定义如下:

```
struct CPU_workqueue_struct {
    spinlock_t lock;              /* lock protecting this structure */
    long remove_sequence;  /* least-recently added (next to run) */
    long insert_sequence;         /* next to add */
    struct list_head worklist;    /* list of work */
    wait_queue_head_t more_work;
    wait_queue_head_t work_done;
    struct workqueue_struct *wq; /* associated workqueue_struct */
    task_t *thread;               /* associated thread */
    int run_depth;                /* run_workqueue() recursion depth */
};
```

CPU_workqueue_struct 结构用来表示该 CPU 所对应的工作线程及所需处理的工作队列,其中 thread 成员为所关联的工作线程,wq 为所属的 workqueue_struct,worklist 为该 CPU 上的所需处理的工作队列。

在实现上,所有的工作线程都是普通内核线程,它使用 worker_thread( )作为线程函数,该线程函数在进行一段初始化操作后便进入无限循环,并睡眠等待。当有需处理的延迟工作被加入到工作队列中时,该线程函数被唤醒并循环处理对应工作队列中的每一个工作单元;当工作队列中没有工作单元需要被处理时,它又重新进入睡眠状态,等待下一次的唤醒。

每个 CPU 的同一类型工作单元被连接成一个工作队列,work_struct 用来表示每一个需要被延迟处理的工作单元,其数据结构定义如下:

```
struct work_struct {
   unsigned long pending;     /* is this work pending? */
   struct list_head entry;    /* link list of all work */
   void (*func)(void *);      /* handler function */
   void *data;                /* argument to handler */
   void *wq_data;             /* used internally */
   struct timer_list timer;  /* timer used by delayed work queues */
};
```

worker_thread( )线程函数的主要任务就是遍历工作队列上所有需要被处理的工作单元,如果队列非空,则调用 run_workqueue( )处理具体的延迟工作,其主体形式如下:

```
    for ( ; ; )
    {
        /*cwq 为 CPU_workqueue_struct 类型 */
        if (! list_empty(&cwq->worklist))
                run_workqueue(cwq);
    }
```

run_workqueue( )主体形式如下:

```
while (! list_empty(&cwq->worklist))
{
struct work_struct *work;
void (*f)(void *);
void *data;
work = list_entry(cwq->worklist.next, struct work_struct, entry);
 f=work->func;
 data = work->data;
 list_del_init(cwq->worklist.next);
 f(data);
 }
```

该函数的主要任务就是遍历工作队列,通过 list_entry 取得每一个工作单元的 work_struct 结构,并以 work->data 为参数调用其具体处理函数 work->func( )。

使用工作队列来完成延迟工作的大概过程为:驱动为需要延迟处理的工作建立一 work_struct 结构,该结构即为工作单元,它还包含一函数指针用来处理具体的延迟工作;该工作单元被添加到当前 CPU 的默认工作线程或自定义工作线程的工作队列中等待处理,在某一时刻,工作线程被唤醒,它将循环处理工作队列中的每一个工作单元。

6.2 使用工作队列

使用系统中的默认工作队列是非常容易的。首先,要为需要延迟处理的工作单元建立一个 work_struct 结构,内核提供了下面 2 个宏来方便地建立该结构:

```
DECLARE_WORK(name, void (*func)(void *), void *data);
INIT_WORK (struct work_struct *work, void (*func)(void *), void *data)。
```

其中,DECLARE_WORK 创建类型为 work_struct 的 name 变量,而 INIT_WORK 创建一个指向 work_struct 结构的 work 指针,func 为具体的工作单元处理函数,data 为 func 的参数。

work_struct 创建好后,需要将该结构放入到默认工作线程的工作队列中去,内核提供以下 2 个宏操作:

```
schedule_work(work);
schedule_delayed_work(work, delay)。
```

这 2 个宏操作的主要区别就在于一个是立即被调度,一个是延迟 delay 个时钟周期后被调度。

考虑到系统提供的默认工作线程可能满足不了一些驱动的特殊要求,系统同时也提供了接口以方便驱动产生满足自己需求的自定义类型的工作线程。创建工作线程使用 struct workqueue_struct *create_workqueue (const char *name)。其中,name 为该类型工作工作线程的名字,例如创建类型为 myworker 的工作线程的代码如下:

```
struct workqueue_struct *myworker;
myworker = create_workqueue("myworker");
```

将 work_struct 加入到自定义工作线程的工作队列中可以采用以下接口:

```
int queue_work(struct workqueue_struct *wq, struct work_struct *work);
int queue_delayed_work (struct workqueue_struct *wq,struct work_struct *work,unsigned long delay)。
```

3G 手机与 NGN 通信设备间的多媒体互通设计

Design for Communication Between 3G Mobile Phone and NGN Terminal

吴醒峰/WU Xing-feng, 刘元安/LIU Yuan-an

北京邮电大学电信工程学院,北京 100876

School of Telecommunication Engineering, Beijing University of Posts and Telecommunications, Beijing 100876, China

[摘要] 随着移动通信和 NGN 网络的发展,实现 NGN 终端和 3G 手机之间的多媒体互通成为一个必须解决的问题。3G 网络中的无线多媒体通信协议是 3G-324M,而 NGN 分组网络中的多媒体通信标准则是 H.323/SIP,支持两种设备互通必须解决不同层面的问题。为此,提出了基于软交换技术实现 3G 无线多媒体与 NGN 网络互连的方案,并深入探讨了方案的实现机制。

[关键词] NGN; 3G; 多媒体 **[中图分类号]** TN919

[文献标识码] A **[文章编号]** 1000-7857(2006)05-0070-03

Abstract: With the development of wireless communication and NGN, the communication between 3G mobile phone and NGN terminal becomes a necessary problem to resolve. The multimedia communication protocol for 3G is 3G-324M; however, the protocol for NGN is H.323/SIP. The scheme based on soft switch to implement the multimedia communication between 3G and NGN is presented and its implement mechanism is discussed in depth.

Key Words: NGN; 3G; Multimedia **CLC Number:** TN919

Document Code: A **Article ID:** 1000-7857(2006)05-0070-03

1 引言

随着数据业务、多媒体业务在网络中主导地位的逐步确立,NGN 将朝着全 IP 的方向发展,IP 将成为语音、数据、信令的统一载体。软交换技术是构造 NGN 的核心技术,其最主要特征是采用控制、承载、业务三者分离的层次结构,使网络框架更清晰,为通信网提供一个不受传输模式(如 IP、ATM 等)限制的业务环境;除此之外,其最大优势在于业务能够真正独立于网络,使得业务和应用的提供具有较大的灵活性。近来各种无线多媒体通信业务蓬勃发展,诸如移动视频电话、移动多媒体、移动用户游戏等由无线多媒体通信触发的增值业务正在成为各运营商和设备商研发的重点。3G 网络是未来无线通信发展的方向,它的一个重要目标就是以 VOIP 方式提供语音业务以及多样化的多媒体业务,并实现 3G 核心承载网络与 IP 网的融合统一。3G 网络与 IP 网的融合统一不仅可以使运营商提高对网络资源的利用率,而且移动终端也可以通过统一的 IP 网方便地访问和享受因特网上的各种信息和服务。根据 UMTS 和 CDMA2000 的通信条件及信道特性,在原有的视频通信标准 H.324 基础上制订了 3GPP 和 3GPP2 无线视频通信标准 3G-324M,而在 NGN 网络中,多媒体通信主要是通过基于 H.323 和 SIP 协议标准的设备实现。这样就出现了一个问题:在 NGN 网络体系下,如何实现 3G-324M 终端和分组

收稿日期: 2006-02-08

作者简介: 吴醒峰,男,北京市海淀区西土城路 10 号北京邮电大学 171 信箱,博士研究生,主要从事无线局域网 QoS 设计和无线网络的通信协议研究;E-mail:wuxingfeng@sohu.com

刘元安(通讯作者),男,北京邮电大学电信工程学院,教授,主要从事 10 Mbit/s 以上移动通信系统及关键技术、GSM 和 CDMA 智能天线技术、数字系统电磁兼容技术和新型电磁兼容测量技术等研究;E-mail:yuliu@bupt.edu.cn

7 结束语

Linux 2.0 内核中 BH 函数机制的引入将中断服务程序一分为二,使得处理起来更加灵活。但它存在 2 个问题:32 个 BH 函数不够用;串行化问题。Linux 2.0 内核的后续版本中引入的任务队列机制很好地解决了第一个问题,Linux 2.4 内核中引入的 tasklet 机制更好地解决了这 2 个问题,并将老的 BH 函数统一纳入到软中断的体系框架下。

参考文献(References)

[1] 张江陵, 刘劲松, 冯丹. 磁盘阵列环境下 LINUX 中断机制的改进与测量[J]. 小型微型计算机系统, 2005,26(2):302-306.

[2] 李月蛾, 乔晓燕. 实时数据采集系统的中断机制设计与研究[J]. 计算机应用与软件, 2004, 21(4):44-45.

[3] DANIEL PB, CESATI M. Understanding the Linux kernel [M]. Sebastopol, CA: O´REILLY Press , 2001.

[4] 李善平, 刘文峰, 李程远. Linux 内核 2.4 版源代码分析大全 [M]. 北京: 机械工业出版社, 2002.

[5] 毛德操, 胡希明. Linux 内核源代码情景分析(上、下册)[M]. 杭州: 浙江大学出版社, 2001.

(责任编辑 王 芷)