Tekkaman Ninja

tekkamanninja.blog.chinaunix.net

tekkamanninj

博客访问： 75923
博文数量： 263
博客积分： 15936
博客等级： 上将
技术积分： 13951
用 户 组： 普通用户
注册时间： 2007-03-27 11:22

加关注 短消息
论坛 加好友

个人简介

Fedora-ARM

文章分类

文章存档

Linux设备驱动程序学习（10）-时间、延迟及延缓操作

2007-11-29 11:27:42

分类： LINUX

Linux设备驱动程序学习（10）
-时间、延迟及延缓操作

度量时间差

时钟中断由系统定时硬件以周期性的间隔产生，这个间隔由内核根据 HZ 值来设定，HZ 是一个体系依赖的值，在 <linux/param.h>中定义或该文件包含的某个子平台相关文件中。作为通用的规则，即便如果知道 HZ 的值，在编程时应当不依赖这个特定值，而始终使用HZ。对于当前版本，我们应完全信任内核开发者，他们已经选择了最适合的HZ值，最好保持 HZ 的默认值。

对用户空间，内核HZ几乎完全隐藏，用户 HZ 始终扩展为 100。当用户空间程序包含 param.h，且每个报告给用户空间的计数器都做了相应转换。对用户来说确切的 HZ 值只能通过 /proc/interrupts 获得：/proc/interrupts 的计数值除以 /proc/uptime 中报告的系统运行时间。

对于ARM体系结构：在<linux/param.h>文件中的定义如下：

```
#ifdef __KERNEL__
# define HZ         CONFIG_HZ    /* Internal kernel timer frequency */
# define USER_HZ     100         /* 用户空间使用的HZ，User interfaces are in "ticks" */
# define CLOCKS_PER_SEC     (USER_HZ)     /* like times() */
#else
# define HZ         100
#endif
```

也就是说：HZ 由__KERNEL__和CONFIG_HZ决定。若未定义__KERNEL__，HZ为100；否则为CONFIG_HZ。而CONFIG_HZ是在内核的根目录的.config文件中定义，并没有在make menuconfig的配置选项中出现。Linux的\arch\arm\configs\s3c2410_defconfig文件中的定义为：

```
#
# Kernel Features
#
# CONFIG_PREEMPT is not set
# CONFIG_NO_IDLE_HZ is not set
CONFIG_HZ=200
# CONFIG_AEABI is not set
# CONFIG_ARCH_DISCONTIGMEM_ENABLE is not set
```

所以正常情况下s3c24x0的HZ为200。这一数值在后面的实验中可以证实。

每次发生一个时钟中断，内核内部计数器的值就加一。这个计数器在系统启动时初始化为 0， 因此它代表本次系统启动以来的时钟嘀哒数。这个计数器是一个 64-位 变量(即便在 32-位的体系上)并且称为“jiffies_64”。但是驱动通常访问 jiffies 变量（unsigned long）（根据体系结构的不同：可能是jiffies_64 ，可能是jiffies_64 的低32位）。使用 jiffies 是首选，因为它访问更快，且无需在所有的体系上实现原子地访问 64-位的 jiffies_64 值。

使用 jiffies 计数器

2013年（3）
2012年（61）
2011年（66）
2010年（27）
2009年（30）
2008年（23）
2007年（52）

我的朋友


小蜗牛快

cfm5538

jikaishi

shizhenc

pxy05215

李怀远

yan19900

wkm81018

xiousi

最近访客


apang199

appcount

zaicichu

lhxzui

小蜗牛快

小尾巴鱼

erain_30

hushup

wilfred_

订阅

推荐博文

- linux 3.x的 通用时钟架构 ...
- SCN的相关解析
- Flash驱动学习
- 浅谈nagios之state type和 no...
- DB2（Linux 64位）安装教程...
- insert语句造成latch:library...
- 2014.06.13 网络公开课《让我...
- MySQL Slave异常关机的处理（...
- 巧用shell脚本分析数据库用户...
- 查询linux,HP-UX的cpu信息...

热词专题

- linux系统权限修复——学生误...
- Modbus协议使用
- linux
- busybox原理
- php环境搭建教程

这个计数器和用来读取它的工具函数包含在 <linux/jiffies.h>， 通常只需包含 <linux/sched.h>，它会自动放入 jiffies.h 。 jiffies 和 jiffies_64 必须被当作只读变量。当需要记录当前 jiffies 值（被声明为 volatile 避免编译器优化内存读）时，可以简单地访问这个 unsigned long 变量，如：

```
#include <linux/jiffies.h>
unsigned long j, stamp_1, stamp_half, stamp_n;

j = jiffies; /* read the current value */
stamp_1 = j + HZ; /* 1 second in the future */
stamp_half = j + HZ/2; /* half a second */
stamp_n = j + n * HZ / 1000; /* n milliseconds */
```

以下是一些简单的工具宏及其定义：

```
#define time_after(a,b)         \
    (typecheck(unsigned long, a) && \
     typecheck(unsigned long, b) && \
     ((long)(b) - (long)(a) < 0))
#define time_before(a,b)    time_after(b,a)
#define time_after_eq(a,b)     \
    (typecheck(unsigned long, a) && \
     typecheck(unsigned long, b) && \
     ((long)(a) - (long)(b) >= 0))
#define time_before_eq(a,b)    time_after_eq(b,a)
```

用户空间的时间表述法（struct timeval 和 struct timespec ）与内核表述法的转换函数：

```
#include <linux/time.h> /* #include <linux/jiffies.h> --> \kernel\time.c*/

struct timespec {
    time_t    tv_sec;        /* seconds */
    long    tv_nsec;    /* nanoseconds */
};
#endif

struct timeval {
    time_t        tv_sec;        /* seconds */
    suseconds_t    tv_usec;    /* microseconds */
};

unsigned long timespec_to_jiffies(struct timespec *value);
void jiffies_to_timespec(unsigned long jiffies, struct timespec *value);
unsigned long timeval_to_jiffies(struct timeval *value);
void jiffies_to_timeval(unsigned long jiffies, struct timeval *value);
```

访问jiffies_64 对于 32-位 处理器不是原子的，这意味着如果这个变量在你正在读取它们时被更新你可能读到错误的值。若需要访问jiffies_64，内核有一个特别的辅助函数，为你完成适当的锁定：

```
#include <linux/jiffies.h>
u64 get_jiffies_64(void);
```

处理器特定的寄存器

若需测量非常短时间间隔或需非常高的精度，可以借助平台依赖的资源。许多现代处理器包含一个随时钟周期不断递增的计数寄存器，他是进行高精度的时间管理任务唯一可靠的方法。最有名的计数器寄存器是 TSC （ timestamp counter），在 x86 的 Pentium 处理器开始引入并在之后所有的 CPU 中出现（包括 x86_64 平台）。它是一个 64-位 寄存器，计数 CPU 的时钟周期，可从内核和用户空间读取。在包含了 <asm/msr.h>（一个 x86-特定的头文件，它的名子代表"machine-specific registers"）的代码中可使用这些宏：

```
rdtsc(low32,high32);/*原子地读取 64-位TSC 值到 2 个 32-位 变量*/
rdtscl(low32);/*读取TSC的低32位到一个 32-位 变量*/
rdtscll(var64);/*读 64-位TSC 值到一个 long long 变量*/
```

```
/*下面的代码行测量了指令自身的执行时间:*/
unsigned long ini, end;
rdtscl(ini); rdtscl(end);
printk("time lapse: %li\n", end - ini);
```

一些其他的平台提供相似的功能，并且内核头文件提供一个体系无关的功能用来代替 rdtsc，称 get_cycles（定义在 <asm/timex.h>(由 <linux/timex.h> 包含)），原型如下:

```
#include <linux/timex.h>
 cycles_t get_cycles(void);
/*这个函数在每个平台都有定义，但在没有时钟周期计数器的平台上返回 0 */

/*由于s3c2410系列处理器上没有时钟周期计数器所以get_cycles定义如下：*/
typedef unsigned long cycles_t;

static inline cycles_t get_cycles (void)
{
    return 0;
}
```

获取当前时间

驱动一般无需知道时钟时间（用年月日、小时、分钟、秒来表达的时间），只对用户程序才需要,如 cron 和 syslogd。 内核提供了一个将时钟时间转变为秒数值的函数：

```
unsigned long
mktime(const unsigned int year0, const unsigned int mon0,
       const unsigned int day, const unsigned int hour,
       const unsigned int min, const unsigned int sec)
{
    unsigned int mon = mon0, year = year0;

    /* 1..12 -> 11,12,1..10 */
    if (0 >= (int) (mon -= 2)) {
        mon += 12;    /* Puts Feb last since it has leap day */
        year -= 1;
    }

    return ((((unsigned long)
          (year/4 - year/100 + year/400 + 367*mon/12 + day) +
          year*365 - 719499
     )*24 + hour /* now have hours */
     )*60 + min /* now have minutes */
    )*60 + sec; /* finally seconds */
}
/*这个函数将时间转换成从1970年1月1日0小时0分0秒到你输入的时间所经过的秒数，溢出时间为2106-02-07 06:28:16。本人认为这个函数的使用应这样：若你要计算2000-02-07 06:28:16 到2000-02-09 06:28:16 所经过的秒数：unsigned long time1 = mktime(2000,2,7,6,28,16)-mktime(2000,2,9,6,28,16)； 若还要转成jiffies，就再加上:unsigned long time2 = time1*HZ.注意溢出的情况！*/
```

为了处理绝对时间，<linux/time.h> 导出了 do_gettimeofday 函数，它填充一个指向 struct timeval 的指针变量。绝对时间也可来自 xtime 变量，一个 struct timespec 值，为了原子地访问它，内核提供了函数 current_kernel_time。它们的精确度由硬件决定，原型是:

```
#include <linux/time.h>
void do_gettimeofday(struct timeval *tv);
struct timespec current_kernel_time(void);
/*得到的数据都表示当前时间距UNIX时间基准1970－01－01 00：00：00的相对时间*/
```

以上两个函数在ARM平台都是通过 xtime 变量得到数据的。

全局变量xtime：它是一个timeval结构类型的变量，用来表示当前时间距UNIX时间基准1970－01－01 00：00：00的相对秒数值。

结构timeval是Linux内核表示时间的一种格式（Linux内核对时间的表示有多种格式，每种格式都有不同的时间精度），其时间精度是微秒。该结构是内核表示时间时最常用的一种格式，它定义在头文件include/linux/time.h中，如下所示：

```
struct timeval {
time_t tv_sec; /* seconds */
suseconds_t tv_usec; /* microseconds */
};
```

其中，成员tv_sec表示当前时间距UNIX时间基准的秒数值，而成员tv_usec则表示一秒之内的微秒值，且1000000>tv_usec>＝0。

Linux内核通过timeval结构类型的全局变量xtime来维持当前时间，该变量定义在kernel/timer.c文件中，如下所示：

```
/* The current time */
volatile struct timeval xtime __attribute__ ((aligned (16)));
```

但是，全局变量xtime所维持的当前时间通常是供用户来检索和设置的，而其他内核模块通常很少使用它（其他内核模块用得最多的是jiffies），因此对xtime的更新并不是一项紧迫的任务，所以这一工作通常被延迟到时钟中断的底半部（bottom half）中来进行。由于bottom half的执行时间带有不确定性，因此为了记住内核上一次更新xtime是什么时候，Linux内核定义了一个类似于jiffies的全局变量wall_jiffies，来保存内核上一次更新xtime时的jiffies值。时钟中断的底半部分每一次更新xtime的时侯都会将wall_jiffies更新为当时的jiffies值。全局变量wall_jiffies定义在kernel/timer.c文件中：

```
/* jiffies at the most recent update of wall time */
unsigned long wall_jiffies;
```

原文网址：http://blog.csdn.net/freedom1013/archive/2007/03/13/1528310.aspx

延迟执行

设备驱动常常需要延后一段时间执行一个特定片段的代码，常常允许硬件完成某个任务.

长延迟

有时，驱动需要延后执行相对长时间，长于一个时钟嘀哒。

忙等待(尽量别用)

若想延迟执行若干个时钟嘀哒，精度要求不高。最容易的(尽管不推荐) 实现是一个监视 jiffy 计数器的循环。这种忙等待实现的代码如下:

```
while (time_before(jiffies, j1))
    cpu_relax();
```

对 cpu_relex 的调用将以体系相关的方式执行，在许多系统中它根本不做任何事，这个方法应当明确地避免。对于ARM体系来说：

```
#define cpu_relax()             barrier()
```

也就是说在ARM上运行忙等待相当于：

```
while (time_before(jiffies, j1)) ;
```

这种忙等待严重地降低了系统性能。如果未配置内核为抢占式，这个循环在延时期间完全锁住了处理器，计算机直到时间 j1 到时会完全死掉。如果运行一个可抢占的内核时会改善一点，但是忙等待在可抢占系统中仍然是浪费资源的。更糟的是，当进入循环时如果中断碰巧被禁止，jiffies 将不会被更新，并且 while 条件永远保持真，运行一个抢占的内核也不会有帮助，唯一的解决方法是重启。

让出处理器

忙等待加重了系统负载，必须找出一个更好的技术：不需要CPU时释放CPU 。 这可通过调用schedule函数实现（在 <linux/sched.h> 中声明）：

```
while (time_before(jiffies, j1)) {
    schedule();
}
```

在计算机空闲时运行空闲任务（进程号 0，由于历史原因也称为swapper）可减轻处理器工作负载、降低温度、增加寿命。

超时

实现延迟的最好方法应该是让内核为我们完成相应的工作。

（1）若驱动使用一个等待队列来等待某些其他事件,并想确保它在一个特定时间段内运行，可使用：

```
#include <linux/wait.h>
long wait_event_timeout(wait_queue_head_t q, condition, long timeout);
long wait_event_interruptible_timeout(wait_queue_head_t q, condition, long timeout);
/*这些函数在给定队列上睡眠，但是它们在超时(以 jiffies 表示)到后返回。如果超时，函数返回 0；如果这个进程被其他事件唤醒，则返回以 jiffies 表示的剩余的延迟实现；返回值从不会是负值*/
```

（2）为了实现进程在超时到期时被唤醒而又不等待特定事件（避免声明和使用一个多余的等待队列头），内核提供了 schedule_timeout 函数：

```
#include <linux/sched.h>
signed long schedule_timeout(signed long timeout);

/*timeout 是要延时的 jiffies 数。除非这个函数在给定的 timeout 流失前返回，否则返回值是 0 。schedule_timeout 要求调用者首先设置当前的进程状态。为获得一个不可中断的延迟，可使用 TASK_UNINTERRUPTIBLE 代替。如果你忘记改变当前进程的状态，调用 schedule_time 如同调用 shcedule，建立一个不用的定时器。一个典型调用如下:*/
set_current_state(TASK_INTERRUPTIBLE);
schedule_timeout (delay);
```

短延迟

当一个设备驱动需要处理硬件的延迟（latency潜伏期），涉及到的延时通常最多几个毫秒，在这个情况下，不应依靠时钟嘀哒，而是内核函数 ndelay，udelay和 mdelay ，他们分别延后执行指定的纳秒数，微秒数或者毫秒数，定义在 <asm/delay.h>,原型如下：

```
#include <linux/delay.h>
void ndelay(unsigned long nsecs);
void udelay(unsigned long usecs);
void mdelay(unsigned long msecs);
```

重要的是记住这 3 个延时函数是忙等待；其他任务在时间流失时不能运行。每个体系都实现 udelay，但是其他的函数可能未定义；如果它们没有定义，<linux/delay.h> 提供一个缺省的基于 udelay 的版本。在所有的情况中，获得的延时至少是要求的值，但可能更多。udelay 的实现使用一个软件循环，它基于在启动时计算的处理器速度和使用整数变量 loos_per_jiffy确定循环次数。

为避免在循环计算中整数溢出，传递给udelay 和 ndelay的值有一个上限，如果你的模块无法加载和显示一个未解决的符号：__bad_udelay，这意味着你调用 udleay时使用太大的参数。
作为一个通用的规则：若试图延时几千纳秒，应使用 udelay 而不是 ndelay；类似地，毫秒规模的延时应当使用 mdelay 完成而不是一个更细粒度的函数。

有另一个方法获得毫秒(和更长)延时而不用涉及到忙等待的方法是使用以下函数（在<linux/delay.h> 中声明）：

```
void msleep(unsigned int millisecs);
unsigned long msleep_interruptible(unsigned int millisecs);
void ssleep(unsigned int seconds)
```

若能够容忍比请求的更长的延时，应使用 schedule_timeout，msleep 或 ssleep。

内核定时器

当需要调度一个以后发生的动作，而在到达该时间点时不阻塞当前进程，则可使用内核定时器。内核定时器用来调度一个函数在将来一个特定的时间（基于时钟嘀哒）执行，从而可完成各类任务。
内核定时器是一个数据结构，它告诉内核在一个用户定义的时间点使用用户定义的参数执行一个用户定义

的函数，函数位于 <linux/timer.h> 和 kernel/timer.c 。被调度运行的函数几乎确定不会在注册它们的进程在运行时运行，而是异步运行。实际上，内核定时器通常被作为一个"软件中断"的结果而实现。当在进程上下文之外(即在中断上下文)中运行程序时，必须遵守下列规则：

（1）不允许访问用户空间；
（2）current 指针在原子态没有意义；
（3）不能进行睡眠或者调度. 例如：调用 kmalloc(..., GFP_KERNEL) 是非法的，信号量也不能使用因为它们可能睡眠。

通过调用函数 in_interrupt()能够告知是否它在中断上下文中运行，无需参数并如果处理器当前在中断上下文运行就返回非零。
通过调用函数 in_atomic()能够告知调度是否被禁止，若调度被禁止返回非零；调度被禁止包含硬件和软件中断上下文以及任何持有自旋锁的时候。

在后一种情况，current 可能是有效的，但是访问用户空间是被禁止的，因为它能导致调度发生. 当使用 in_interrupt()时，都应考虑是否真正该使用的是 in_atomic 。他们都在 <asm/hardirq.h> 中声明。

内核定时器的另一个重要特性是任务可以注册它本身在后面时间重新运行，因为每个 timer_list 结构都会在运行前从激活的定时器链表中去连接,因此能够立即链入其他的链表。一个重新注册它自己的定时器一直运行在同一个 CPU.

即便在一个单处理器系统，定时器是一个潜在的态源，这是异步运行直接结果。因此任何被定时器函数访问的数据结构应当通过原子类型或自旋锁被保护，避免并发访问。

定时器 API

内核提供给驱动许多函数来声明、注册以及删除内核定时器：

```
#include <linux/timer.h>
struct timer_list {
    struct list_head entry;
    unsigned long expires;/*期望定时器运行的绝对 jiffies 值，不是一个 jiffies_64 值，因为定时器不被期望在将来很久到时*/
    void (*function)(unsigned long); /*期望调用的函数*/
    unsigned long data;/*传递给函数的参数，若需要在参数中传递多个数据项,可以将它们捆绑成单个数据结构并且将它的指针强制转换为 unsiged long 的指针传入。这种做法在所有支持的体系上都是安全的并且在内存管理中相当普遍*/
    struct tvec_t_base_s *base;
#ifdef CONFIG_TIMER_STATS
    void *start_site;
    char start_comm[16];
    int start_pid;
#endif
};
/*这个结构必须在使用前初始化，以保证所有的成员被正确建立（包括那些对调用者不透明的初始化）：*/
void init_timer(struct timer_list *timer);
struct timer_list TIMER_INITIALIZER(_function, _expires, _data);
/*在初始化后和调用 add_timer 前，可以改变 3 个公共成员：expires、function和data*/
void add_timer(struct timer_list * timer);
int del_timer(struct timer_list * timer);/*在到时前禁止一个已注册的定时器*/
int del_timer_sync(struct timer_list *timer); /*如同 del_timer ，但还保证当它返回时，定时器函数不在任何 CPU 上运行，以避免在 SMP 系统上竞态，并且在 单处理器内核中和 del_timer 相同。这个函数应当在大部分情况下优先考虑。 如果它被从非原子上下文调用，这个函数可能睡眠，但是在其他情况下会忙等待。当持有锁时要小心调用 del_timer_sync ，如果这个定时器函数试图获得同一个锁，系统会死锁。如果定时器函数重新注册自己，调用者必须首先确保这个重新注册不会发生；这通常通过设置一个" 关闭 "标志来实现，这个标志被定时器函数检查*/
int mod_timer(struct timer_list *timer, unsigned long expires); /*更新一个定时器的超时时间，常用于超时定时器。也可在正常使用 add_timer时在不活动的定时器上调用mod_timer*/
int timer_pending(const struct timer_list * timer); /*通过调用timer_list结构中一个不可见的成员，返回定时器是否在被调度运行*/
```

内核定时器的实现《LDD3》介绍的比较笼统，以后看《ULK3》的时候再细细研究。

一个内核定时器还远未完善，因为它受到 jitter 、硬件中断，还有其他定时器和其他异步任务的影响。虽然一个简单数字 I/O关联的定时器对简单任务是足够的，但不合适在工业环境中的生产系统，对于这样的任务，你将最可能需要实时内核扩展（RT-Linux）.

Tasklets

另一个有关于定时的内核设施是 tasklet。它类似内核定时器：在中断时间运行且运行同一个 CPU 上，并接收一个 unsigned long 参数。不同的是：无法要求在一个指定的时间执行函数，只能简单地要求它在以后的一个由内核选择的时间执行。它对于中断处理特别有用：硬件中断必须尽快处理，但大部分的数据管理可以延后到以后安全的时间执行。 实际上，一个 tasket，就象一个内核定时器，在一个"软中断"的上下文中执行(以原子模式)。软件中断是在使能硬件中断时执行异步任务的一个内核机制。

tasklet 以一个数据结构形式存在，使用前必须被初始化。初始化能够通过调用一个特定函数或者通过使用某些宏定义声明结构:

```
#include <linux/interrupt.h>
struct tasklet_struct
{
    struct tasklet_struct *next;
    unsigned long state;
    atomic_t count;
    void (*func)(unsigned long);
    unsigned long data;
};
void tasklet_init(struct tasklet_struct *t,
 void (*func)(unsigned long), unsigned long data);

#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 }

void tasklet_disable(struct tasklet_struct *t);
/*函数暂时禁止给定的 tasklet被 tasklet_schedule 调度，直到这个 tasklet 被再次被
enable；若这个 tasklet 当前在运行，这个函数忙等待直到这个tasklet退出*/
void tasklet_disable_nosync(struct tasklet_struct *t);
/*和tasklet_disable类似，但是tasklet可能仍然运行在另一个 CPU */
void tasklet_enable(struct tasklet_struct *t);
/*使能一个之前被disable的 tasklet;若这个 tasklet 已经被调度，它会很快运行。
tasklet_enable 和tasklet_disable必须匹配调用，因为内核跟踪每个 tasklet 的"禁止次数"*/
void tasklet_schedule(struct tasklet_struct *t);
/*调度 tasklet 执行，如果tasklet在运行中被调度，它在完成后会再次运行；这保证了在其他事
件被处理当中发生的事件受到应有的注意. 这个做法也允许一个 tasklet 重新调度它自己*/
void tasklet_hi_schedule(struct tasklet_struct *t);
/*和tasklet_schedule类似，只是在更高优先级执行。当软中断处理运行时，它处理高优先级
tasklet 在其他软中断之前，只有具有低响应周期要求的驱动才应使用这个函数，可避免其他软件
中断处理引入的附加周期*/
void tasklet_kill(struct tasklet_struct *t);
/*确保了 tasklet 不会被再次调度来运行，通常当一个设备正被关闭或者模块卸载时被调用。如
果 tasklet 正在运行，这个函数等待直到它执行完毕。若 tasklet 重新调度它自己，则必须阻止
在调用 tasklet_kill 前它重新调度它自己，如同使用 del_timer_sync*/
```

tasklet 的特点:

（1）一个 tasklet 能够被禁止并且之后被重新使能；它不会执行，直到它被使能与被禁止相同的的次数；

（2）如同定时器，一个 tasklet 可以注册它自己；

（3）一个 tasklet 能被调度来执行以正常的优先级或者高优先级；

（4） 如果系统不在重负载下,taslet 可能立刻运行，但是从不会晚于下一个时钟嘀哒；

（5）一个 tasklet 可能和其他 tasklet 并发，但是它自己是严格地串行的 ，且tasklet 从不同时运行在不同处理器上，通常在调度它的同一个 CPU 上运行。

工作队列

工作队列类似 taskets，允许内核代码请求在将来某个时间调用一个函数，不同在于：
（1）tasklet 在软件中断上下文中运行，所以 tasklet 代码必须是原子的。而工作队列函数在一个特殊内核进程上下文运行，有更多的灵活性，且能够休眠。
（2）tasklet 只能在最初被提交的处理器上运行，这只是工作队列默认工作方式。
（3）内核代码可以请求工作队列函数被延后一个给定的时间间隔。
（4）tasklet 执行的很快，短时期，并且在原子态，而工作队列函数可能是长周期且不需要是原子的，两个机制有它适合的情形。

工作队列有 struct workqueue_struct 类型，在 <linux/workqueue.h> 中定义。一个工作队列必须明确的在使用前创建，宏为：

```
struct workqueue_struct *create_workqueue(const char *name);
struct workqueue_struct *create_singlethread_workqueue(const char *name);
```

每个工作队列有一个或多个专用的进程("内核线程")，这些进程运行提交给这个队列的函数。 若使用 create_workqueue，就得到一个工作队列它在系统的每个处理器上有一个专用的线程。在很多情况下，过多线程对系统性能有影响，如果单个线程就足够则使用 create_singlethread_workqueue 来创建工作队列。

提交一个任务给一个工作队列，在这里《LDD3》介绍的内核2.6.10和我用的新内核2.6.22.2已经有不同了，老接口已经不能用了，编译会出错。这里我只讲2.6.22.2的新接口，至于老的接口我想今后内核不会再有了。从这一点我们可以看出内核发展。

```
/*需要填充work_struct或delayed_work结构，可以在编译时完成，宏如下：*/
struct work_struct {
    atomic_long_t data;
#define WORK_STRUCT_PENDING 0        /* T if work item pending execution */
#define WORK_STRUCT_FLAG_MASK (3UL)
#define WORK_STRUCT_WQ_DATA_MASK (~WORK_STRUCT_FLAG_MASK)
    struct list_head entry;
    work_func_t func;
};

struct delayed_work {
    struct work_struct work;
    struct timer_list timer;
};

DECLARE_WORK(n, f)
/*n 是声明的work_struct结构名称，f是要从工作队列被调用的函数*/
DECLARE_DELAYED_WORK(n, f)
/*n是声明的delayed_work结构名称，f是要从工作队列被调用的函数*/

/*若在运行时需要建立 work_struct 或 delayed_work结构，使用下面 2 个宏定义:*/
INIT_WORK(struct work_struct *work, void (*function)(void *));
PREPARE_WORK(struct work_struct *work, void (*function)(void *));
INIT_DELAYED_WORK(struct delayed_work *work, void (*function)(void *));
PREPARE_DELAYED_WORK(struct delayed_work *work, void (*function)(void *));
/* INIT_* 做更加全面的初始化结构的工作，在第一次建立结构时使用. PREPARE_* 做几乎同样的工作，但是它不初始化用来连接 work_struct或delayed_work 结构到工作队列的指针。如果这个结构已经被提交给一个工作队列，且只需要修改该结构,则使用 PREPARE_* 而不是 INIT_* */

/*有 2 个函数来提交工作给一个工作队列:*/
int queue_work(struct workqueue_struct *queue, struct work_struct *work);
int queue_delayed_work(struct workqueue_struct *queue, struct delayed_work *work,
unsigned long delay);
/*每个都添加work到给定的workqueue。如果使用 queue_delay_work，则实际的工作至少要经过指定的 jiffies 才会被执行。 这些函数若返回 1 则工作被成功加入到队列；若为0，则意味着这个 work 已经在队列中等待，不能再次加入*/
```

在将来的某个时间，这个工作函数将被传入给定的 data 值来调用。这个函数将在工作线程的上下文运

行，因此它可以睡眠（你应当知道这个睡眠可能影响提交给同一个工作队列的其他任务）工作函数不能访问用户空间，因为它在一个内核线程中运行，完全没有对应的用户空间来访问。
取消一个挂起的工作队列入口项可以调用：

```
int cancel_delayed_work(struct delayed_work *work);
void cancel_work_sync(struct work_struct *work)
```

如果这个入口在它开始执行前被取消，则返回非零。内核保证给定入口的执行不会在调用 cancel_delay_work 后被初始化. 如果 cancel_delay_work 返回 0，但是，这个入口可能已经运行在一个不同的处理器，并且可能仍然在调用 cancel_delayed_work 后在运行. 要绝对确保工作函数没有在 cancel_delayed_work 返回 0 后在任何地方运行，你必须跟随这个调用来调用：

```
void flush_workqueue(struct workqueue_struct *queue);
```

在 flush_workqueue 返回后，没有在这个调用前提交的函数在系统中任何地方运行。
而cancel_work_sync会取消相应的work，但是如果这个work已经在运行那么cancel_work_sync会阻塞，直到work完成并取消相应的work。

当用完一个工作队列，可以去掉它，使用：

```
void destroy_workqueue(struct workqueue_struct *queue);
```

共享队列

在许多情况下，设备驱动不需要它自己的工作队列。如果你只偶尔提交任务给队列，简单地使用内核提供的共享的默认的队列可能更有效。若使用共享队列，就必须明白将和其他人共享它，这意味着不应当长时间独占队列(不能长时间睡眠)，并且可能要更长时间才能获得处理器。

使用的顺序：
（1） 建立 work_struct 或 delayed_work

```
static struct work_struct jiq_work;
static struct delayed_work jiq_work_delay;

    /* this line is in jiq_init() */
INIT_WORK(&jiq_work, jiq_print_wq);
INIT_DELAYED_WORK(&jiq_work_delay, jiq_print_wq);
```

（2）提交工作

```
int schedule_work(&jiq_work);/*对于work_struct结构*/
int schedule_delayed_work(&jiq_work_delay, delay);/*对于delayed_work结构*/
/*返回值的定义和 queue_work 一样*/
```

若需取消一个已提交给工作队列入口项，可以使用 cancel_delayed_work和cancel_work_sync， 但刷新共享队列需要一个特殊的函数：

```
void flush_scheduled_work(void);
```

因为不知道谁可能使用这个队列，因此不可能知道 flush_schduled_work 返回需要多长时间。

ARM9 s3c2440AL 实验

jit模块：jit

jiq模块：jiq

实验数据：

```
[Tekkaman2440@SBC2440V4]#cd /lib/modules/
[Tekkaman2440@SBC2440V4]#insmod jit.ko
[Tekkaman2440@SBC2440V4]#head -6 /proc/currentime
0x0002b82f 0x000000010002b82f 1191.119051
                                    1191.115000000
0x0002b82f 0x000000010002b82f 1191.119204
                                    1191.115000000
0x0002b82f 0x000000010002b82f 1191.119230
                                    1191.115000000
[Tekkaman2440@SBC2440V4]#dd bs=20 count=5 < /proc/jitbusy
   201604 201804
```

```
   201804 202004
   202004 202204
   202204 202404
   202404 202604
5+0 records in
5+0 records out
[Tekkaman2440@SBC2440V4]#dd bs=20 count=5 < /proc/jitsched
   212640 212840
   212840 213040
   213040 213240
   213240 213440
   213440 213640
5+0 records in
5+0 records out
[Tekkaman2440@SBC2440V4]#dd bs=20 count=5 < /proc/jitqueue
   218299 218499
   218499 218699
   218699 218899
   218899 219099
   219099 219299
5+0 records in
5+0 records out
[Tekkaman2440@SBC2440V4]#dd bs=20 count=5 < /proc/jitschedto
   228413 228613
   228613 228813
   228813 229013
   229013 229213
   229213 229413
5+0 records in
5+0 records out
[Tekkaman2440@SBC2440V4]#cat /proc/jitimer
   time delta inirq pid cpu command
   236945 0 0 832 0 cat
   236955 10 1 0 0 swapper
   236965 10 1 0 0 swapper
   236975 10 1 0 0 swapper
   236985 10 1 0 0 swapper
   236995 10 1 0 0 swapper
[Tekkaman2440@SBC2440V4]#cat /proc/jitasklet
   time delta inirq pid cpu command
   238437 0 0 833 0 cat
   238437 0 1 3 0 ksoftirqd/0
   238437 0 1 3 0 ksoftirqd/0
   238437 0 1 3 0 ksoftirqd/0
   238437 0 1 3 0 ksoftirqd/0
   238437 0 1 3 0 ksoftirqd/0
[Tekkaman2440@SBC2440V4]#cat /proc/jitasklethi
   time delta inirq pid cpu command
   239423 0 0 834 0 cat
   239423 0 1 3 0 ksoftirqd/0
   239423 0 1 3 0 ksoftirqd/0
   239423 0 1 3 0 ksoftirqd/0
   239423 0 1 3 0 ksoftirqd/0
   239423 0 1 3 0 ksoftirqd/0
[Tekkaman2440@SBC2440V4]#insmod jiq.ko
[Tekkaman2440@SBC2440V4]#cat /proc/jiqwq
    time delta preempt pid cpu command
   405005 0 0 5 0 events/0
   405005 0 0 5 0 events/0
   405005 0 0 5 0 events/0
```

```
   405005 0 0 5 0 events/0
   405005 0 0 5 0 events/0
   405005 0 0 5 0 events/0
   405005 0 0 5 0 events/0
   405005 0 0 5 0 events/0
   405005 0 0 5 0 events/0
   405005 0 0 5 0 events/0
   405005 0 0 5 0 events/0
[Tekkaman2440@SBC2440V4]#cat /proc/jiqwqdelay
    time delta preempt pid cpu command
   406114 1 0 5 0 events/0
   406115 1 0 5 0 events/0
   406116 1 0 5 0 events/0
   406117 1 0 5 0 events/0
   406118 1 0 5 0 events/0
   406119 1 0 5 0 events/0
   406120 1 0 5 0 events/0
   406121 1 0 5 0 events/0
   406122 1 0 5 0 events/0
   406123 1 0 5 0 events/0
   406124 1 0 5 0 events/0
[Tekkaman2440@SBC2440V4]#cat /proc/jiqtimer
    time delta preempt pid cpu command
   420605 0 0 853 0 cat
   420805 200 256 0 0 swapper
[Tekkaman2440@SBC2440V4]#cat /proc/jiqtasklet
    time delta preempt pid cpu command
   431905 0 256 3 0 ksoftirqd/0
   431905 0 256 3 0 ksoftirqd/0
   431905 0 256 3 0 ksoftirqd/0
   431905 0 256 3 0 ksoftirqd/0
   431905 0 256 3 0 ksoftirqd/0
   431905 0 256 3 0 ksoftirqd/0
   431905 0 256 3 0 ksoftirqd/0
   431905 0 256 3 0 ksoftirqd/0
   431905 0 256 3 0 ksoftirqd/0
   431905 0 256 3 0 ksoftirqd/0
   431905 0 256 3 0 ksoftirqd/0
```


上一篇：Linux设备驱动程序学习（8）-分配内存
下一篇：Linux设备驱动程序学习（11）-中断处理

感谢楼主这一系列文章，收益颇多

回复 | 举报

评论热议

请登录后评论。

登录 注册