BAN CƠ YẾU CHÍNH PHỦ

HỌC VIỆN KỸ THUẬT MẬT MÃ

DELAY VÀ TIMER

Ngành: Công nghệ thông tin

Chuyên ngành: Kỹ thuật phần mềm nhúng và di động

Mã số: 52.48.02.01

Nội dung

- Wall time và System uptime
- Delay
- Timer

Wall time và System uptime

- Mục tiêu
  - Xác định thời điểm xảy ra một sự kiện, hoặc xác định khoảng thời gian giữa hai sự kiện.
  - Thực hiện một tác vụ tại một thời điểm trong tương lai
- Khái niệm
  - **Wall time** (hay còn gọi là **absolute time**) là khoảng thời gian đã trôi qua kể từ thời điểm Epoch (0:00:00 ngày 1/1/1970)
  - **System uptime** (hay còn gọi là **relative time**) là khoảng thời gian đã trôi qua kể từ lúc khởi động hệ thống

Wall time

- Linux kernel đọc wall time từ thiết bị RTC (Real Time Clock)
  - Trong hệ thống x86, RTC được tích hợp trong chip cầu nam
  - Trong hệ thống nhúng, RTC kết nối với SoC thông qua bus I2C hoặc SPI.
- Sau khi đọc được từ RTC, Linux kernel lưu wall time vào biến kiểu cấu trúc **timeval** hoặc **timespec**
  - Số giây đã trôi qua kể từ thời điểm Epoch được lưu trong trường ***tv_sec***
  - Số micro giây (hoặc nano giây) đã trôi qua kể từ giây cuối cùng được lưu trong trường ***tv_usec*** (hoặc ***tv_nsec***)

Wall time

- cấu trúc **timeval** hoặc **timespec**

```
#include <uapi/linux/time.h>
/*
 * cấu trúc timeval biểu diễn khoảng thời gian với độ chính xác micro giây.
 */
struct timeval {
        __kernel_time_t         tv_sec;         /* seconds */
        __kernel_suseconds_t    tv_usec;        /* microseconds */
};

/*
 * cấu trúc timepsec biểu diễn khoảng thời gian với độ chính xác nano giây.
 */
struct timespec {
        __kernel_time_t tv_sec;         /* seconds */
        long            tv_nsec;        /* nanoseconds */
};
```

Wall time

- Để hỗ trợ device driver lấy wall time, Linux kernel cung cấp các hàm sau:

```
#include <linux/timekeeping.h>
/*
 * Chức năng: lấy wall time với độ chính xác lên tới micro giây.
 * Tham số đầu vào:
 *     tv [O]: địa chỉ của cấu trúc timeval dùng để chứa wall time.
 * Giá trị trả về: không có.
 */
void do_gettimeofday(struct timeval *tv);

/*
 * Chức năng: lấy wall time với độ chính xác lên tới nano giây.
 * Tham số đầu vào: không có.
 * Giá trị trả về: wall time.
 */
struct timespec current_kernel_time(void);
```

System uptime

- **System uptime** (hay còn gọi là **relative time**) là khoảng thời gian đã trôi qua kể từ lúc khởi động hệ thống.
  - Cách tính system uptime dựa trên số lần system timer gửi tín hiệu ngắt tới CPU. Trong quá trình khởi động, kernel sẽ thiết lập tần số ngắt cho system timer này. Tần số này có đặc điểm như sau:
  - Tần số này còn được gọi là **tick rate**, ký hiệu là **HZ**, thể hiện số lần system timer gửi tín hiệu ngắt tới CPU trong 1 giây. Chu kì ngắt là khoảng thời gian giữa hai lần ngắt, được gọi là 1 **tick** hoặc 1 **jiffy**. Mối liên hệ: 1 tick = 1 jiffy = 1/**HZ**. Ví dụ, trong Ubuntu 16.04, **HZ** = 250, nên 1 jiffy = 4ms
  - Các hệ thống khác nhau có thể có giá trị **HZ** khác nhau, thường dao động từ 100 đến 1000. Nếu muốn thay đổi **HZ**, thì ta phải thay đổi giá trị của **CONFIG_HZ** trong file *.config* rồi biên dịch lại kernel

System uptime

- Để lưu số lần system timer ngắt CPU kể từ khi khởi động hệ thống, Linux kernel sử dụng biến **unsigned long volatile *jiffies***
  - Biến này có kiểu **unsigned long** nên sẽ có độ dài 32 bit trên các hệ thống 32 bit, và dài 64 bit trên các hệ thống 64 bit. Nếu **HZ** = 1000, biến có độ dài 32 bit sẽ bị tràn trong vòng khoảng 50 ngày, còn biến có độ dài 64 bit sẽ bị tràn sau khoảng 600 triệu năm.
  - Ngoài biến ***jiffies***, Linux kernel cũng cung cấp một biến khác có tên là ***jiffies_64***. Trên hệ thống 64 bit, thì ***jiffies_64*** giống với ***jiffies***. Trên hệ thống 32 bit, thì ***jiffies*** là 32 bit thấp của ***jiffies_64***. Việc sử dụng biến ***jiffies_64*** giúp chúng ta không cần lo lắng về vấn đề tràn. Để đọc được giá trị của biến ***jiffies_64*** trên các hệ thống 32 bit, ta nên sử dụng hàm **u64 get_jiffies_64(void)**.
  - Đây là biến **volatile**, nên trình biên dịch sẽ không tối ưu việc sử dụng biến này trong vòng lặp **for** hoặc **while**. Mỗi lần cần sử dụng ***jiffies***, giá trị của biến này sẽ được đọc lại từ RAM.
  - Lúc khởi động, Linux kernel thiết lập biến này bằng 0. Cứ mỗi khi nhận được tín hiệu ngắt từ system timer, kernel sẽ tăng biến này lên 1 đơn vị. Do đó, system uptime = ***jiffies*** * jiffy = ***jiffies*****/HZ** (giây)

System uptime

- Linux cung cấp các hàm quy đổi sau

```
#include <linux/jiffies.h>

/* chuyển đổi jiffies sang giây (độ chính xác nano giây) */
void jiffies_to_timespec(unsigned long jiffies, struct timespec *val)

/* chuyển đổi jiffies sang giây (độ chính xác micro giây) */
void jiffies_to_timeval(unsigned long jiffies, struct timeval *val)

/*chuyển đổi jiffies sang mili giây */
unsigned int jiffies_to_msecs(const unsigned long j)

/* chuyển đổi jiffies sang micro giây */
unsigned int jiffies_to_usecs(const unsigned long j)

/* chuyển số giây sang số lượng jiffy tương đương */
unsigned long timespec_to_jiffies(struct timespec *val)
unsigned long timeval_to_jiffies(struct timeval *val)

/* chuyển số mili giây sang số lượng jiffy tương đương */
unsigned long msecs_to_jiffies(const unsigned int m)

/* chuyển số micro giây sang số lượng jiffy tương đương */
unsigned long usec_to_jiffies(const unsigned int u)
```

Ví dụ minh họa

- Nội dung chính của ví dụ:
  - Xác định thời điểm tiến trình mở và đóng device file ở dạng system uptime. Hiệu của hai system uptime này sẽ cho biết thời gian mà tiến trình sử dụng device file.
  - Xác định thời điểm đọc dữ liệu và ghi dữ liệu vào device file ở dạng wall time
- Ví dụ được phát triển từ ví dụ trong Chapter trước
  - Các file hầu như không thay đổi, chỉ có file *vchar_driver.c* cần sửa

Ví dụ minh họa

- tham chiếu tới thư viện *<linux/timekeeping.h>* và *<linux/jiffies.h>*

```
#include <linux/uaccess.h> /* thu vien nay chua cac ham trao doi du lieu giua user va kernel */
#include <linux/ioctl.h> /* thu vien nay chua cac ham phuc vu ioctl */
#include <linux/proc_fs.h> /* thu vien nay chua cac ham tao/huy file trong procfs */
#include <linux/timekeeping.h> /* thu vien nay chua cac ham de lay wall time */
#include <linux/jiffies.h> /* thu vien nay chua cac ham de lay system uptime */

#include "vchar_driver.h" /* thu vien mo ta cac thanh ghi cua vchar device */
```

Ví dụ minh họa

- để lưu lại thời điểm tiến trình mở device file, ta thêm trường ***start_time*** vào cấu trúc **vchar_drv**

```
@@ -47,6 +49,7 @@ struct _vchar_drv {
            vchar_dev_t * vchar_hw;
            struct cdev *vcdev;
            unsigned int open_cnt;
+           unsigned long start_time;
    } vchar_drv;
```

- Để biết được tiến trình đã sử dụng device file trong bao lâu, ta tính số tick đã trôi qua kể từ lúc mở cho tới lúc đóng device file, sau đó chuyển số tick này sang dạng **timeval**

Ví dụ minh họa

```
/* cac ham entry points */
static int vchar_driver_open(struct inode *inode, struct file *filp)
{
        vchar_drv.start_time = jiffies;
        vchar_drv.open_cnt++;
        printk(KERN_INFO "Handle opened event (%d)\n", vchar_drv.open_cnt);
        return 0;
}

static int vchar_driver_release(struct inode *inode, struct file *filp)
{
        struct timeval using_time;
        jiffies_to_timeval(jiffies - vchar_drv.start_time, &using_time);
        printk(KERN_INFO "The driver is used in %ld.%ld seconds\n", using_time.tv_sec, using_time.tv_usec/1000);
        printk(KERN_INFO "Handle closed event\n");
        return 0;
}
```

Ví dụ minh họa

- Để xác định thời điểm đọc và ghi dữ liệu vào device file dưới dạng wall time

```
@@ -207,8 +214,9 @@ static ssize_t vchar_driver_read(struct file *filp, char __user *user_buf, size_
 {
         char *kernel_buf = NULL;
         int num_bytes = 0;
+        struct timespec rd_ts = current_kernel_time();

-        printk(KERN_INFO "Handle read event start from %lld, %zu bytes\n", *off, len);
+        printk(KERN_INFO "Handle read event start from %lld, %zu bytes at %ld.%ld from Epoch\n", *off, len, rd_ts.tv_sec, rd_ts.tv_nsec/1000000);

         kernel_buf = kzalloc(len, GFP_KERNEL);
         if(kernel_buf == NULL)
@@ -230,7 +238,10 @@ static ssize_t vchar_driver_write(struct file *filp, const char __user *user_buf
 {
         char *kernel_buf = NULL;
         int num_bytes = 0;
-        printk(KERN_INFO "Handle write event start from %lld, %zu bytes\n", *off, len);
+        struct timeval wr_tv;
+
+        do_gettimeofday(&wr_tv);
+        printk(KERN_INFO "Handle write event start from %lld, %zu bytes at %ld.%ld from Epoch\n", *off, len, wr_tv.tv_sec, wr_tv.tv_usec/1000);

         kernel_buf = kzalloc(len, GFP_KERNEL);
         if(copy_from_user(kernel_buf, user_buf, len))
```

Delay

- Linux kernel hỗ trợ các hàm để device driver trì hoãn một tác vụ lại một khoảng thời gian. Khi sử dụng các kỹ thuật này, device driver sẽ tạm ngừng hoạt động trong khoảng thời gian đó (blocking). Có hai kiểu trì hoãn: busy waiting và sleep waiting.
  - Khi muốn trì hoãn một tác vụ trong một khoảng thời gian lớn hơn 1 jiffy (long delay), chúng ta nên ưu tiên sử dụng các kỹ thuật thuộc kiểu sleep waiting hơn, để tránh làm suy giảm hiệu năng của hệ thống.
  - Khi muốn trì hoãn trong một khoảng thời gian nhỏ hơn 1 jiffy (short delay), chúng ta chỉ có thể sử dụng các kỹ thuật thuộc kiểu busy waiting

Delay

- device driver cũng hay phải thực thi một tác vụ nào đó ở một thời điểm trong tương lai theo 2 cách:
  - **blocking**: việc sử dụng các kỹ thuật thuộc loại này khiến device driver bị tạm dừng hoạt động một thời gian trước khi thực thi một tác vụ. Nói một cách khác, tác vụ đó sẽ bị trì hoãn một khoảng thời gian. Do đó, loại này còn có tên là **delay**.
  - **non-blocking**: device driver sẽ đăng ký với Linux kernel rằng: tác vụ F cần được thực thi ở thời điểm T trong tương lai. Sau khi đăng ký xong, device driver tiếp tục thực thi các tác vụ khác. Khi đến giờ, kernel sẽ lập lịch thực thi tác vụ F

Long delay

- Busy waiting
  - Giải pháp đơn giản nhất là device driver liên tục chờ cho tới khi biến ***jiffies*** vượt quá một giá trị thì device driver mới tiếp tục thực hiện

```
/*
 * thời điểm later là 10 jiffy nữa kể từ thời điểm hiện tại
 * nếu HZ = 250, thì có nghĩa là 40ms nữa kể từ thời điểm hiện tại
 */
unsigned long later = jiffies + 10;

//driver chờ trong vòng 10 jiffy.
while (jiffies <= later);

//driver thực thi một tác vụ nào đó sau khi chờ
```

Long delay

- Giải pháp này trước đơn giản, nhưng lại không hiệu quả. Bởi vì, CPU phải chờ đợi một cách vô ích. Thêm vào đó, cách triển khai trên sẽ bị sai nếu kết quả (***jiffies*** + 10) bị tràn. Để khắc phục vấn đề tràn, Linux kernel cung cấp các macro

Macro so sánh	**Ý nghĩa**
time_after(a,b)	Trả về **true** nếu a > b (thời điểm a sau thời điểm b)
time_before(a,b)	Trả về **true** nếu a < b (thời điểm a trước thời điểm b)
time_after_eq(a,b)	Trả về **true** nếu a >= b (thời điểm a không ở trước thời điểm b)
time_before_eq(a,b)	Trả về **true** nếu a <= b (thời điểm a không ở sau thời điểm b)

Long delay

- Nếu muốn làm một việc gì đó sau N giây nữa kể từ thời điểm hiện tại, ta có thể lập trình như sau:

```
/*
 * giá trị jiffies tương ứng với thời điểm hiện tại
 * giá trị later tương ứng với 2 giây nữa kể từ thời điểm hiện tại
 */
unsigned long later = jiffies + 2*HZ;

//chờ cho tới khi nào giá trị của jiffies vượt qua later
while (time_before(jiffies, later));

//đã đủ 2 giây, thực hiện tác vụ cần làm
```

Long delay

- Sleep waiting
  - Thay vì bắt CPU phải chờ một cách vô ích, Linux cung cấp hàm [schedule_timeout]

```
#include<linux/sched.h>
/*
 * Chức năng: khi gọi hàm này, device driver sẽ bị tạm dừng hoạt động (còn gọi
 *            là đi ngủ) trong một khoảng thời gian. Khi đó, Linux kernel sẽ
 *            lập lịch cho CPU thực thi driver khác, hoặc tiến trình khác.
 * Tham số đầu vào:
 *    timeout [I]: device driver sẽ đi ngủ trong khoảng thời gian @timeout.
 *                 Giá trị này tính theo đơn vị jiffy.
 * Trả về:
 *    Khi hết khoảng thời gian @timeout, Linux kernel lập lịch để CPU thực thi
 *    device driver trở lại (còn gọi là thức dậy), và hàm này trả về 0.
 *    Trong trường hợp device driver thức dậy trước thời hạn, hàm này trả về
 *    số jiffy còn lại.
 * Chú ý: Trước khi gọi hàm này, ta nên gọi hàm set_current_state để thiết
 *        lập trạng thái lúc ngủ của device driver là TASK_INTERRUPTIBLE hay
 *        TASK_UNINTERRUPTIBLE.
 *        Nếu là TASK_UNINTERRUPTIBLE, device driver ngủ trong khoảng thời gian
 *        ít nhất là @timeout.
 *        Nếu là TASK_INTERRUPTIBLE, device driver có thể sẽ ngủ trong khoảng
 *        thời gian nhỏ hơn @timeout.
 */
signed long schedule_timeout(signed long timeout);
```

Long delay

- hàm **ssleep** và **msleep**.

```
#include<linux/delay.h>
/*
 * Chức năng: khi gọi các hàm này, device driver sẽ đi ngủ trong một khoảng thời gian.
 *            Khi đó, Linux kernel sẽ lập lịch cho CPU chuyển sang thực thi driver khác,
 *            hoặc tiến trình khác.
 * Tham số đầu vào:
 *     seconds  [I]: khoảng thời gian device driver đi ngủ, tính theo đơn vị giây.
 *     millisecs[I]: khoảng thời gian device driver đi ngủ, tính theo mili giây.
 * Trả về: không có.
 */

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

Long delay

- Nếu muốn tạm dừng device driver trong một khoảng thời gian cho tới khi xảy ra một sự kiện E, ta sử dụng các hàm sau:

```
#include <linux/wait.h>
/*
 * Chức năng: khi gọi các hàm này, device driver sẽ đi ngủ cho tới khi
 *            - hoặc là xuất hiện sự kiện mong đợi
 *            - hoặc là hết thời gian
 * Tham số:
 *    wq        : hàng đợi các sự kiện mà device driver này mong chờ.
 *    condition: sự kiện mong đợi (được biểu diễn dưới dạng biểu thức C).
 *               Device driver đi ngủ cho tới khi biểu thức @condition
 *               trở thành đúng.
 *    timeout   : thời gian chờ đợi, tính theo đơn vị jiffy
 * Giá trị trả về:
 *    Trả về 0 nếu biểu thức @condition vẫn sai sau khoảng thời gian @timeout.
 *    Trả về 1 nếu biểu thức @condition chuyển sang đúng sau khoảng @timeout.
 *    Trả về số jiffy còn lại nếu biểu thức @condition chuyển sang đúng trước
 *    thời hạn.
 */
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)
```

Short delay

- Các kỹ thuật đã trình bày trong phần trên chỉ cho phép tạm dừng device driver khoảng thời gian tối thiểu bằng 1 jiffy. Nhưng khi device driver làm việc với phần cứng, thì nó thường chỉ cần tạm dừng khoảng vài micro giây, tức là nhỏ hơn 1 jiffy
  - Trong những trường hợp như vậy, Linux cung cấp các hàm sau:

```
#include<linux/delay.h>

/* tạm ngừng device driver ở cấp độ nano giây */
void ndelay(unsigned long nsecs);

/* tạm ngừng device driver ở cấp độ micro giây */
void udelay(unsigned long usecs);

/* tạm ngừng device driver ở cấp độ mili giây */
void mdelay(unsigned long msecs);
```

- Chú ý:
  - Các hàm trên đều trì hoãn theo kiểu busy waiting. Nghĩa là CPU chỉ chờ một cách vô ích trước khi làm công việc tiếp theo.
  - Không thể áp dụng các kỹ thuật kiểu sleep waiting để thực hiện short delay

Ví dụ minh họa

- Yêu cầu
  - khi tiến trình đọc/hoặc ghi dữ liệu vào device file, driver tương ứng sẽ tạm ngừng hoạt động trong 10 giây trước khi trả về kết quả cho tiến trình
- Thực hiện
  - thêm hàm các hàm có tác dụng trì hoãn vào trong hàm **vchar_driver_read** và **vchar_driver_write** của device driver

Ví dụ minh họa

```
@@ -222,6 +224,9 @@ static ssize_t vchar_driver_read(struct file *filp, char __user *user_buf, size_
        if(kernel_buf == NULL)
                return 0;

+       set_current_state(TASK_UNINTERRUPTIBLE);
+       schedule_timeout(10*HZ);
+
        num_bytes = vchar_hw_read_data(vchar_drv.vchar_hw, *off, len, kernel_buf);
        printk(KERN_INFO "read %d bytes from HW\n", num_bytes);

@@ -247,6 +252,9 @@ static ssize_t vchar_driver_write(struct file *filp, const char __user *user_buf
        if(copy_from_user(kernel_buf, user_buf, len))
                return -EFAULT;

+       mdelay(10000); //cho theo kieu busy-wait
+       //ssleep(10); //cho theo kieu sleep
+
        num_bytes = vchar_hw_write_data(vchar_drv.vchar_hw, *off, len, kernel_buf);
        printk(KERN_INFO "writes %d bytes to HW\n", num_bytes);
```

Kernel Timer

- **Giới thiệu**
  - Linux kernel cũng cung cấp các kỹ thuật thuộc loại non-blocking, bao gồm kernel timer, tasklet và workqueue
  - kernel timer tương tự như một cái đồng hồ báo thức
- **Biểu diễn kernel timer trong Linux**
- **Cách sử dụng kernel timer**
- **Ví dụ minh họa**

Biểu diễn kernel timer trong Linux

- Linux kernel sử dụng cấu trúc **timer_list** để biểu diễn kernel timer. Cấu trúc này chứa một số trường quan trọng sau:

```
#include <linux/timer.h>
/*
 * @expires: thời điểm mà ta muốn thực thi tác vụ (hàm @function) trong tương lai.
 *           Thời điểm này được tính từ lúc khởi động hệ thống, theo đơn vị jiffy.
 * @function: hàm này sẽ được thực thi khi system uptime = @expires.
 * @data: tham số truyền cho hàm @function. Nếu ta muốn truyền nhiều thông tin cho
 *        hàm @function, ta có thể gộp các thông tin đó vào một cấu trúc, sau đó
 *        ép địa chỉ của cấu trúc này sang kiểu unsigned long, rồi gán cho trường
 *        @data.
 */
struct timer_list {
    …
    unsigned long expires;
    void (*function)(unsigned long);
    unsigned long data;
    …
}
```

Cách sử dụng kernel timer

- Linux kernel cung cấp một số hàm giúp khởi tạo, đăng ký, sửa đổi và xóa bỏ kernel timer.
  - Trước khi sử dụng các hàm này, chúng ta cần khai báo một kernel timer tương tự như khai báo biến thông thường
    - *struct timer_list my_ktimer;*
- Sau khi khai báo, chúng ta gọi hàm **init_timer** để khởi tạo kernel timer. Ví dụ:

```
/*
 * Chức năng:
 *    Khởi tạo kernel timer.
 * Tham số đầu vào:
 *    timer [O]: địa chỉ của kernel timer cần khởi tạo.
 * Giá trị trả về: không có.
 * Chú ý: Hàm này cần được gọi trước khi sử dụng các hàm khác.
 */
void init_timer(struct timer_list *timer)

/* Ví dụ */
init_timer(&my_ktimer);
```

Cách sử dụng kernel timer

- Sau khi khởi tạo xong, chúng ta tiến hành cấu hình cho kernel timer. Quá trình cấu hình bao gồm:
  - Thiết lập thời điểm kernel timer sẽ báo thức. Chú ý: thời điểm này được tính từ lúc khởi động hệ thống, theo đơn vị jiffy.
  - Chỉ định hàm nào sẽ được thực thi khi kernel timer báo thức.
  - Chỉ định dữ liệu sẽ được truyền cho hàm nói trên.
- Ví dụ về quá trình cấu hình kernel timer như sau:

```
/* kernel timer sẽ báo thức trong 5s nữa */
my_ktimer.expires = jiffies + 5*HZ;

/* hàm handle_ktimer_func(unsigned long data) được định nghĩa ở đâu đó */
my_ktimer.function = handle_ktimer_func;

/* biến data thường là một cấu trúc dữ liệu */
my_ktimer.data = (unsigned long)&data;
```

Cách sử dụng kernel timer

- Sau khi cấu hình xong, chúng ta gọi hàm **add_timer** để đăng ký kernel timer với Linux kernel. Ví dụ:

```
/*
 * Chức năng:
 *    Hàm này đăng ký kernel timer với Linux kernel.
 *    Sau đó, kernel timer này bắt đầu hoạt động.
 *    Lúc này, ta nói kernel timer đang ở trạng thái ACTIVE.
 * Tham số:
 *    timer [I]: kernel timer được đăng ký với Linux kernel.
 * Giá trị trả về: không có.
 * Chú ý:
 *     Hàm này nên được gọi sau quá trình cấu hình kernel timer.
 *
 *     Khi đến thời điểm báo thức, hàm @function sẽ được thực thi.
 *     Sau đó, kernel timer sẽ ngừng hoạt động. Lúc này, ta nói
 *     kernel timer đang ở trạng thái INACTIVE.
 */
void add_timer(struct timer_list *timer)

/* Ví dụ */
add_timer(&my_ktimer);
```

Cách sử dụng kernel timer

- Có thể sử dụng hàm **mod_timer** nếu muốn thay đổi thời điểm báo thức của kernel timer

```
/*
 * Chức năng:
 *   Thay đổi thời điểm báo thức của kernel timer.
 *   Hoạt động của hàm này có thể được hiểu như sau:
 *      mod_timer(ktimer, new_expires) {
 *              del_timer(timer);
 *              timer->expires = new_expires ;
 *              add_timer(timer);
 *      }
 * Tham số:
 *   timer   [I]: kernel timer cần được thay đổi thời điểm báo thức.
 *   expires [I]: thời điểm báo thức mới. Thời điểm này được tính
 *                từ lúc khởi động hệ thống, đơn vị là jiffy.
 * Giá trị trả về:
 *   Nếu kernel timer đang ở trạng thái ACTIVE, hàm này trả về 1.
 *   Nếu kernel timer đang ở trạng thái INACTIVE, thì hàm này trả
 *   về 0. Sau đó, kernel timer sẽ chuyển sang trạng thái ACTIVE.
 * Chú ý:
 *   Hàm này thường được gọi trong @function để thực thi một
 *   tác vụ định kì.
 */
int mod_timer(struct timer_list *timer, unsigned long expires);

/* Ví dụ */
mod_timer(&my_ktimer, jiffies + new_delay);
```

Cách sử dụng kernel timer

- Khi không cần dùng kernel timer nữa, ta gọi hàm **del_timer** để hủy bỏ

```
/*
 * Chức năng:
 *   Hàm này hủy một kernel timer.
 * Tham số:
 *   timer [I]: kernel timer sẽ bị hủy bỏ.
 * Giá trị trả về:
 *   Nếu kernel timer đang ở trạng thái INACTIVE, trả về 0.
 *   Nếu kernel timer đang ở trạng thái ACTIVE, trả về 1.
 */
int del_timer(struct timer_list *timer);

/* Ví dụ */
del_timer(&my_ktimer);
```

Ví dụ minh họa

- Ví dụ này sẽ trình bày cách sử dụng các hàm của Linux kernel để khởi tạo, đăng ký và thay đổi một kernel timer. Ý tưởng chính của ví dụ này là: sau khi lắp vchar driver vào kernel, cứ 10 giây một lần, vchar driver sẽ hiển thị cặp số tự nhiên đối nhau tăng dần

Ví dụ minh họa

- Từ ví dụ trước, các file hầu như không thay đổi, chỉ có file *vchar_driver.c* cần sửa lại
  - trong cấu trúc **vchar_drv**, ta thêm trường ***vchar_ktimer*** để biểu diễn kernel timer được sử dụng trong vchar driver.
  - Ta cũng định nghĩa cấu trúc **vchar_ktimer_data_t** để đóng gói các thông tin cần truyền cho hàm **timer_list.function**.

```
@@ -50,8 +51,14 @@ struct _vchar_drv {
            struct cdev *vcdev;
            unsigned int open_cnt;
            unsigned long start_time;
+           struct timer_list vchar_ktimer;
    } vchar_drv;

+ typedef struct vchar_ktimer_data {
+           int param1;
+           int param2;
+ } vchar_ktimer_data_t;
+
    /******************************** device specific - START
    /* ham khoi tao thiet bi */
    int vchar_hw_init(vchar_dev_t *hw)
```

Ví dụ minh họa

- trong hàm **vchar_driver_init**, ta khởi tạo, cấu hình và đăng ký kernel timer với Linux kernel

```
@@ -423,6 +463,11 @@ static int __init vchar_driver_init(void)
                goto failed_create_proc;
        }

+       /* khoi tao, cau hinh va dang ki kernel timer voi Linux kernel */
+       init_timer(&vchar_drv.vchar_ktimer);
+       configure_timer(&vchar_drv.vchar_ktimer);
+       add_timer(&vchar_drv.vchar_ktimer);
+
        printk(KERN_INFO "Initialize vchar driver successfully\n");
        return 0;
```

Ví dụ minh họa

- Hàm **configure_timer** sẽ:
  - Thiết lập thời điểm báo thức là sau 10s nữa.
  - Chỉ định hàm **handle_timer** sẽ được gọi khi đến giờ.
  - Cấu trúc **data** sẽ được truyền cho hàm **handle_timer** khi nó được thực thi
- Các hàm **configure_timer** và **handle_timer** được triển khai như slide sau

```
@@ -361,6 +368,39 @@ static struct file_operations proc_fops =
        .write   = vchar_proc_write,
};

+ static void handle_timer(unsigned long data)
+ {
+         /* lay du lieu */
+         vchar_ktimer_data_t *pdata = (vchar_ktimer_data_t*)data;
+         if(!pdata) {
+                 printk(KERN_ERR "can not handle a NULL pointer\n");
+                 return;
+         }
+
+         /* xu ly du lieu */
+         ++pdata->param1;
+         --pdata->param2;
+         printk(KERN_INFO "Information: %d & %d\n", pdata->param1, pdata->param2);
+
+         /*
+          * Neu muon kernel timer tiep tuc hoat dong thi can
+          * thay doi thoi diem bao thuc.
+          */
+         mod_timer(&vchar_drv.vchar_ktimer, jiffies + 10*HZ);
+ }
```

```
+ static void configure_timer(struct timer_list *ktimer)
+ {
+         static vchar_ktimer_data_t data = {
+                 .param1 = 0,
+                 .param2 = 0
+         };
+         ktimer->expires = jiffies + 10*HZ;
+         ktimer->function = handle_timer;
+         ktimer->data = (unsigned long)&data;
+ }
```

Ví dụ minh họa

- trong hàm **vchar_driver_exit**, ta gọi hàm **del_timer** để hủy kernel timer này

```
@@ -444,6 +489,9 @@ static int __init vchar_driver_init(void)
    /* ham ket thuc driver */
    static void __exit vchar_driver_exit(void)
    {
+           /* huy kernel timer */
+           del_timer(&vchar_drv.vchar_ktimer);
+
            /* huy file /proc/vchar_proc */
            remove_proc_entry("vchar_proc", NULL);
```