BAN CƠ YẾU CHÍNH PHỦ

HỌC VIỆN KỸ THUẬT MẬT MÃ

PHƯƠNG PHÁP PHÁT TRIỂN TRÌNH ĐIỀU KHIỂN

Ngành: Công nghệ thông tin

Chuyên ngành: Kỹ thuật phần mềm nhúng và di động

Mã số: 52.48.02.01

Nội dung

- Xây dựng và thực thi trình điều khiển
- Các kỹ thuật gỡ lỗi
- Quản lý bộ nhớ thiết bị
- Giao tiếp với phần cứng
- Xử lý ngắt

Xây dựng và thực thi trình điều khiển

- Thiết lập môi trường kiểm thử
- Lập trình trình điều khiển
- Biên dịch và nạp
- Khởi tạo và kết thúc
- Các tham số trình điều khiển
- Không gian làm việc (user, kernel)

Thiết lập môi trường kiểm thử

- Lấy các dòng chính thống của nhân trực tiếp từ website gốc (kernel.org)
- Lỗi trong mã nhân có thể dẫn đến sự cố của tiến trình người dùng hoặc toàn bộ hệ thống
- Thử nghiệm nhân
  - Trên một hệ thống không chứa dữ liệu
  - Trên máy ảo

Lập trình trình điều khiển

- Ví dụ:

```
#include <linux/init.h>
#include <linux/module.h>
MODULE_LICENSE("Dual BSD/GPL");
static int hello_init(void)
{
	printk(KERN_ALERT "Hello, world\n");
	return 0;
}
static void hello_exit(void)
{
	printk(KERN_ALERT "Goodbye, cruel world\n");
}
module_init(hello_init);
module_exit(hello_exit);
```

Lập trình trình điều khiển

- Mô-đun gồm hai chức năng
  - hello_init được gọi khi mô-đun được tải vào nhân
  - hello_exit được gọi khi mô-đun bị gỡ khỏi nhân
- module_init và module_exit sử dụng các macro nhân đặc biệt để chỉ ra vai trò của hai chức năng này
- Hàm printk trong kernel space có vai trò giống printf trong user space
- KERN_ALERT là hằng chỉ ra mức độ ưu tiên của thông báo
- MODULE_LICENSE("Dual BSD/GPL"); cho biết nhân có giấy phép miễn phí

Biên dịch và nạp

- Biên dịch
  - Dùng makefile (bài trước)
- Nạp
  - Dùng insmod <tenfile>.ko
    - ✓lsmod hoạt động bằng cách đọc tệp tin ảo /proc/module
    - ✓Thông tin về các mô-đun hiện đang được tải cũng có thể được tìm thấy trong hệ thống tệp tin ảo sysfs dưới /sys/module
- Gỡ
  - rmmod <tenfile>.ko
- Tên của các lời gọi hệ thống được bắt đầu bằng sys_
- Lời gọi hệ thống được định nghĩa trong kernel/module.c

Khởi tạo và kết thúc

- Hàm khởi tạo
  - static int __init initialization_function(void)
    {
        /* Initialization code here */
    }
    module_init(initialization_function);
- Kết thúc
  - Mỗi mô-đun không cần thiết cũng yêu cầu chức năng dọn dẹp, hủy đăng ký giao diện và trả lại tất cả tài nguyên cho hệ thống trước khi mô-đun bị xóa
  - static void __exit cleanup_function(void)
    {
        /* Cleanup code here */
    }
    module_exit(cleanup_function);

Các tham số trình điều khiển

- Cho phép truyền tham số cho trình điều khiển
  - Tăng khả năng tùy biến
- Giá trị tham số này có thể được chỉ định tại thời điểm tải bằng insmod

  ismod hollop howmany=10 whom = “Mom”

  ✓Hai tham số: một giá trị nguyên được gọi là howmany và một chuỗi ký tự được gọi là whom

Các tham số trình điều khiển

- Các tham số được khai báo với macro module_param, được định nghĩa trong moduleparam.h
- module_param có ba tham số: tên của biến, loại của nó và mặt nạ quyền được sử dụng cho mục nhập sysfs

```
static char *whom = "world";
static int howmany = 1;
module_param(howmany, int, S_IRUGO);
module_param(whom, charp, S_IRUGO);
```

Một số kiểu giá trị được hỗ trợ như:

- Bool, invbool (giá trị là true, false. Với kiểu invbool là đảo ngược giá trị)
- Charp: con trỏ char
- Int, long, short, uint, ulong, ushort: Giá trị số.
- Module_param_array (name, type, num, perm): Khai báo mảng.

Các kỹ thuật gỡ lỗi

- Hỗ trợ gỡ lỗi trong nhân
- Gỡ lỗi dựa trên in kết quả
- Gỡ lỗi dựa trên truy vấn
- Gỡ lỗi dựa trên quan sát
- Bộ gỡ lỗi và công cụ liên quan

Hỗ trợ gỡ lỗi trong nhân

- Dựa trên một số cấu hình cho nhân
  - CONFIG_DEBUG_SLAB: Tùy chọn quan trọng này bật một số loại kiểm tra trong các hàm cấp phát bộ nhớ nhân
  - CONFIG_DEBUG_PAGEALLOC: Các trang đầy đủ được xóa khỏi không gian địa chỉ nhân khi được giải phóng
  - CONFIG_INIT_DEBUG: Các mục được đánh dấu bằng __init (hoặc __initdata) sẽ bị loại bỏ sau khi khởi tạo hệ thống hoặc thời gian tải mô-đun
  - …

Gỡ lỗi dựa trên in kết quả

```
printk(KERN_DEBUG "Here I am: %s:%i\n", __FILE__,
__LINE__);
printk(KERN_CRIT "I'm trashed; giving up on %p\n", ptr);
```

Có tám chuỗi loglevel cần quan tâm, được xác định trong <linux/kernel.h>. Liệt kê chúng theo thứ tự giảm dần mức độ nghiêm trọng:

- KERN_EMERG: Dùng cho các tin nhắn khẩn cấp, thường là những tin nhắn trước sự cố.
- KERN_ALERT: Khi đòi hỏi phải hành động ngay lập tức.
- KERN_CRIT: Điều kiện quan trọng, thường liên quan tới lỗi phần cứng hoặc phần mềm nghiêm trọng.
- KERN_ERR: Cảnh báo các điều kiện lỗi. Trình điều khiển thiết bị thường dùng để báo các khó khăn về phần cứng.
- KERN_WARNING: Cảnh báo về một số lỗi không nghiêm trọng với hệ thống.
- KERN_NOTICE: Vấn đề là bình thường nhưng cần lưu ý. Một số liên quan tới bảo mật được cảnh báo ở cấp độ này.
- KERN_INFO: Thông tin, nhiều trình điều khiển in thông tin về phần cứng tìm thấy tại thời điểm khởi động.
- KERN_DEBUG: Dùng để gỡ lỗi.

Gỡ lỗi dựa trên truy vấn

- sử dụng nhiều printk có thể làm chậm đáng kể hệ thống
- Gỡ lỗi dựa trên truy vấn
  - Dựa trên truy vấn file log của hệ thống
  - Sử dụng hệ thống tệp tin /proc, sử dụng trình điều khiển ioctl và xuất các thuộc tính thông qua sysfs
  - tệp tin /proc là một hệ thống đặc biệt, được tạo bởi phần mềm, được nhân sử dụng để xuất thông tin ra ngoài
  - Ioctl là một lời gọi hệ thống hoạt động trên một mô tả tệp tin
    - ✓cần một chương trình khác để phát hành ioctl và hiển thị kết quả

Gỡ lỗi dựa trên quan sát

- Lệnh strace là một công cụ mạnh mẽ hiển thị tất cả các lời gọi hệ thống trên chương trình trong không gian người dùng

Ví dụ: Khi chạy lệnh `strace ls/dev>/dev/scull0` sẽ hiện thị như sau (những dòng cuối của kết quả trả về):

```
open("/dev", O_RDONLY|O_NONBLOCK|O_LARGEFILE|O_DIRECTORY) = 3
fstat64(3, {st_mode=S_IFDIR|0755, st_size=24576, ...}) = 0
fcntl64(3, F_SETFD, FD_CLOEXEC) = 0
getdents64(3, /* 141 entries */, 4096) = 4088
[...]
getdents64(3, /* 0 entries */, 4096) = 0
close(3) = 0
[...]
```

Bộ gỡ lỗi và công cụ liên quan

- Sử dụng gdb
- Bộ gỡ lỗi nhân kdb
- Bản vá kgdb
- Công cụ dò vết trong Linux

Quản lý bộ nhớ thiết bị

- Cấp phát và thu hồi bộ nhớ
- kmalloc là một công cụ mạnh mẽ và dễ dàng thao tác vì nó tương tự như malloc. Hàm này nhanh và không xóa bộ nhớ mà nó thu được. Vùng được phân bổ vẫn giữ nội dung trước đó

```
#include <linux/slab.h>
void *kmalloc(size_t size, int flags);
```

Đối số đầu tiên cho kmalloc là kích thước của khối được phân bổ. Đối số thứ hai, các cờ phân bổ, nó kiểm soát hành vi của kmalloc theo một số cách.

- Một số cờ
  - GFP_ATOMIC: phân bổ bộ nhớ từ các trình xử lý ngắt và các đoạn mã khác bên ngoài bối cảnh của tiến trình. Không bao giờ SLEEP.
  - GFP_KERNEL: cách thức phân bổ bình thường. Có thể SLEEP
  - GFP_USER: phân bổ các trang trong không gian người dùng. Có thể SLEEP.
- Nếu bản sử dụng là portal thì không thể lớn hơn 128 KB, nếu cần lượng lớn hơn thì không nên sử dụng kmalloc

Quản lý bộ nhớ thiết bị

- Bộ nhớ đệm cho thiết bị
  - Nếu thực sự cần bộ nhớ đệm lớn, cách tốt nhất là phân bổ nó bằng cách yêu cầu bộ nhớ khi khởi động

Cấp phát bộ nhớ thời gian khởi động được thực hiện bằng cách gọi một trong các hàm sau:

```
#include <linux/bootmem.h>
void *alloc_bootmem(unsigned long size);
void *alloc_bootmem_low(unsigned long size);
void *alloc_bootmem_pages(unsigned long size);
void *alloc_bootmem_low_pages(unsigned long size);
```

Giải phóng bộ nhớ:

```
void free_bootmem (addr dài không dấu, kích thước dài không dấu);
```

Giao tiếp với phần cứng

- Địa chỉ vật lý
  - = Số thứ tự của byte (một vị trí); được xác định bởi mạch giải mã
- Địa chỉ vật lý của byte nhớ = số thứ tự của byte trong không gian địa chỉ bộ nhớ
- Cổng vào/ra = địa chỉ thanh ghi của khối điều khiển/thiết bị trong không gian vào/ra
- Hệ thống có thể dùng chung không gian bộ nhớ và vào/ra hoặc tách biệt

Giao tiếp với phần cứng

- Sử dụng cổng vào ra
  - **Cấp phát cổng I/O**
    - ✓Nhân cung cấp giao diện đăng ký để cho phép trình điều khiển yêu cầu các cổng mà nó cần. Hàm chủ request_region:

```
#include <linux/ioport.h>
struct resource *request_region(unsigned long first, unsigned long n,const
char *name);
```

    - ✓Khi hoàn thành với thiết lập cổng I/O, chúng cần được trả về hệ thống với:

```
void release_region(unsigned long start, unsigned long n);
```

    - ✓Hàm cho phép trình điều khiển thiết bị có thể kiểm tra xem việc thiết lập cổng I/O có sẵn sàng:

```
int check_region(unsigned long first, unsigned long n);
```

Giao tiếp với phần cứng

- Sử dụng cổng vào/ra

```
unsigned inb(unsigned port);
void outb(unsigned char byte, unsigned port);
```

Đọc hoặc ghi các cổng byte (rộng 8 bit). Đối số port được định nghĩa là unsigned long cho một số nền tảng và unsigned short cho các nền tảng khác. Kiểu trả về của inb cũng khác nhau giữa các kiến trúc.

```
unsigned inw(unsigned port);
void outw(unsigned short word, unsigned port);
```

Các hàm truy cập tới các cổng 16 bit. Chúng không cho phép biên dịch cho nền tảng S390, chỉ hỗ trợ cho I/O byte.

```
unsigned inl(unsigned port);
void outl(unsigned longword, unsigned port);
```

Các hàm truy cập cổng 32bit. Longword được khai báo là unsigned long hoặc insigned int theo từng nền tảng.

Chú ý: Không có hoạt động cho cổng I/O 64 bit nào được định nghĩa.

Giao tiếp với phần cứng

- Sử dụng bộ nhớ vào ra
  - Bộ nhớ I/O là một vùng nhớ (cấp phát trong RAM) mà thiết bị được sử dụng
- Cấp phát và ánh xạ bộ nhớ I/O

Vùng nhớ I/O phải được cấp phát trước khi sử dụng. Giao diện cấp phát vùng bộ nhớ (được định nghĩa trong <linux/ioport.h>) như sau:

```
struct resource *request_mem_region(unsigned long start,
unsigned long len, char *name);
```

Hàm này cấp phát một vùng nhớ len bytes. Bắt đầu từ start, nếu thành công, con trỏ khác NULL được trả về, nếu không sẽ trả về NULL. Việc cấp phát bộ nhớ I/O được liệt kê trong /proc/iomem.

Vùng bộ nhớ cần được giải phóng khi không dùng:

```
void release_mem_region(unsigned long start, unsigned
long len);
```

Hàm (đã cũ) để kiểm tra tính khả dụng của vùng bộ nhớ I/O:

```
nt check_mem_region(unsigned long start, unsigned long
len);
```

Giao tiếp với phần cứng

- Vộ nhớ I/O không được phép truy cập trực tiếp
  - Phải có bước ánh xạ, sử dụng hàm ioremap. Dựa vào hàm này, trình điều khiển có thể truy cập vào bất kỳ địa chỉ bộ nhớ I/O nào

```
#include <asm/io.h>
void *ioremap(unsigned long phys_addr, unsigned long
size);
void *ioremap_nocache(unsigned long phys_addr, unsigned
long size);
void iounmap(void * addr);
```

Giao tiếp với phần cứng

- Truy cập bộ nhớ I/O

Để đọc bộ nhớ I/O, sử dụng một trong các cách sau:

```
unsigned int ioread8(void *addr);
unsigned int ioread16(void *addr);
unsigned int ioread32(void *addr);
```

Ở đây, addr phải là một địa chỉ lấy từ ioremap. Giá tị trả về là giá trị đã độc từ bộ nhớ I/O.

Một số hàm tương tự để ghi vào bộ nhớ I/O:

```
void iowrite8(u8 value, void *addr);
void iowrite16(u16 value, void *addr);
void iowrite32(u32 value, void *addr);
```

Xử lý ngắt

- Kiến thức về ngắt: số hiệu ngăt, ISR, phân loại, hoạt động
- Cài đặt bộ xử lý ngắt
- Thực hiện bộ xử lý ngắt
- Chia sẻ ngắt
- Vào/ra dựa trên ngắt

Cài đặt bộ xử lý ngắt

Các hàm được khai báo trong <linux/interrupt.h>, thực hiện việc đăng ký giao diện ngắt:

```
int request_irq(unsigned int irq, irqreturn_t
(*handler)(int, void *, struct pt_regs *), unsigned long
flags, const char *dev_name, void *dev_id);
```

- Một số tham số của hàm request_irq
  - `unsigned int irq`: Số hiệu ngắt được yêu cầu.
  - `irqreturn_t (*handler)(int, void *, struct pt_regs *)`: Con trỏ trỏ tới hàm đang xử lý.
  - `unsigned long flags`: Tùy chọn mặt nạ bit liên quan tới quản lý ngắt.
  - `const char *dev_name`: Chuỗi được truyền cho request_irq được sử dụng trong /proc/interrupt để hiển thị chủ sở hữu của ngắt.
  - `void *dev_id`: Con trỏ được dùng để chia sẻ đường ngắt. Là mã định danh duy nhất sử dụng cho các được ngắt được giải phón và trình điều khiển cũng dùng để trỏ tới các vùng dữ liệu riêng của nó. Nếu không được dùng nó sẽ là NULL.

Sử dụng ngắt

Mã trong short đáp ứng ngắt bằng lời gọi do_gettimeofday và in thời gian hiện thời trong kích cỡ trang bộ đệm. Sau đó nó đánh thức tiến trình đang đọc vì bây giờ nó có dữ liệu có thể đọc.

```
irqreturn_t short_interrupt(int irq, void *dev_id, struct
pt_regs *regs)
{
      struct timeval tv;
      int written;
      do_gettimeofday(&tv);
/* Write a 16 byte record. Assume PAGE_SIZE is a multiple of
16 */
      written = sprintf((char *)short_head,"%08u.%06u\n",
           (int)(tv.tv_sec % 100000000), (int)(tv.tv_usec));
      BUG_ON(written != 16);
      short_incr_bp(&short_head, written);
      wake_up_interruptible(&short_queue); /* awake any
reading process */
      return IRQ_HANDLED;
}
```

Chia sẻ ngắt

- Chia sẻ ngắt = một ngắt phục vụ từ 2 thiết bị trở lên
  - Phần cứng hiện đại đã được thiết kế để có thể chia sẻ các ngắt
  - Bus PCI yêu cầu thực hiện việc này
  - Nhân linux hỗ trợ chia sẻ ngắt trên tất cả các bus, ngay cả khi chia sẻ theo truyền thống không được hỗ trợ
- Chia sẻ ngắt được chèn thông qua request_irq, giống như không chia sẻ chỉ khác hai điều:
  - Bit SA_SHIRQ phải chỉ định trong đối số flags khi yêu cầu ngắt.
  - Đối số dev_id phải là duy nhất. Bất kỳ con trỏ trong không gian địa chỉ của mô-đun sẽ làm việc nếu dev_id không được đặt là NULL.
- Khi nhân nhận được ngắt, tất cả các trình xử lý ngắt đã đăng ký sẽ được gọi
  - Trình xử lý dùng chung phải có khả năng phân biệt giữa các ngắt cần xử lý và các ngắt khác

Chia sẻ ngắt

- Tất cả các trình xử lý được chèn cho cùng một số ngắt trên cùng dòng của /proc/interrupts

```
CPU0
0: 892335412 XT-PIC timer
1: 453971 XT-PIC i8042
2: 0 XT-PIC cascade
5: 0 XT-PIC libata, ehci_hcd
8: 0 XT-PIC rtc
9: 0 XT-PIC acpi
10: 11365067 XT-PIC ide2, uhci_hcd, uhci_hcd, SysKonnect
SK-98xx, EMU10K1
11: 4391962 XT-PIC uhci_hcd, uhci_hcd
12: 224 XT-PIC i8042
14: 2787721 XT-PIC ide0
15: 203048 XT-PIC ide1
NMI: 41234
LOC: 892193503
ERR: 102
MIS: 0
```

HỎI - ĐÁP