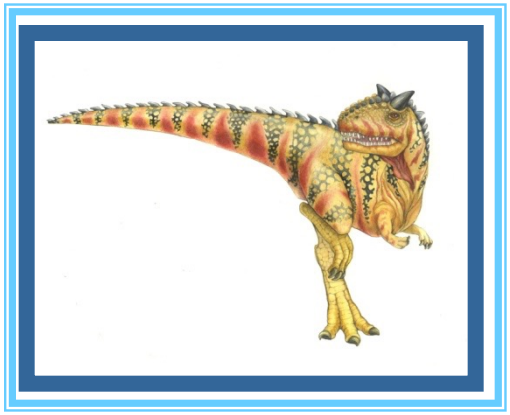
内核模块



内核模块Loadable Kernel Module

本章内容：

- 什么是内核模块

  内核模块机制

  内核模块与应用程序的区别

- 内核模块的使用

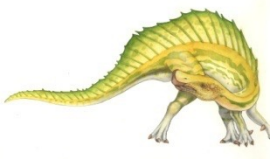
  举例，helloworld.c

  insmod

  lsmod

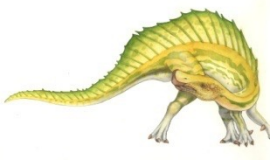
  rmmod

  ksyms

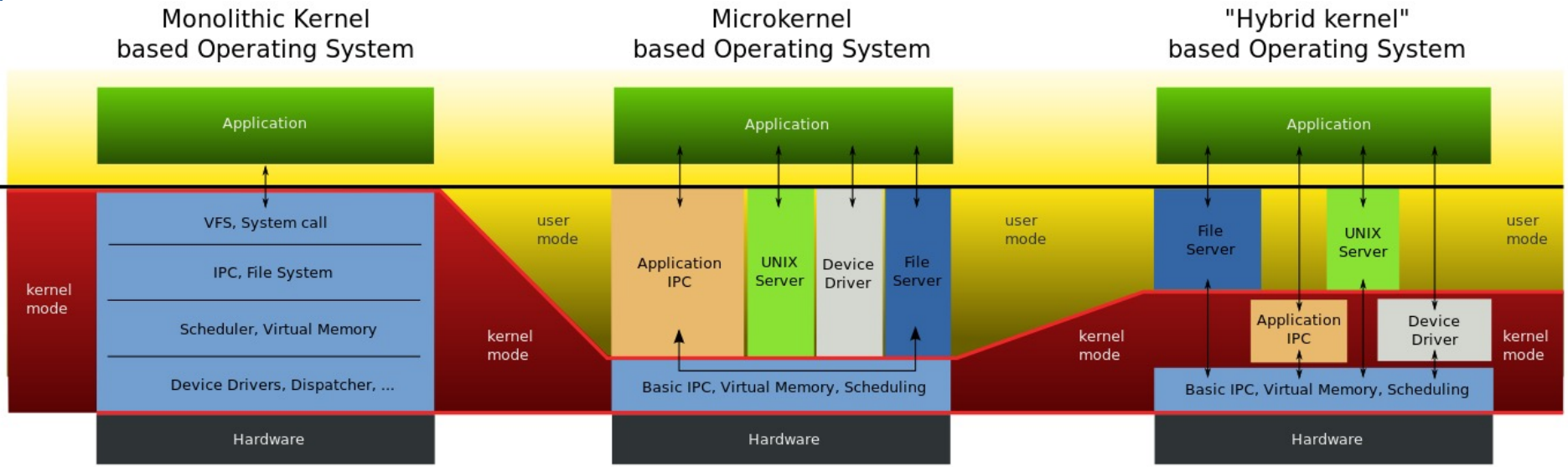


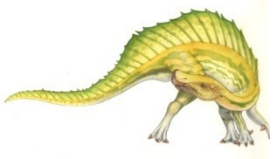
什么是内核模块

- Linux操作系统的内核是单一体系结构(monolithic kernel)的。
- Linux操作系统使用了一种全新的内核模块机制。用户可以根据需要，在不需要对内核重新编译的情况下，模块能动态地装入内核或从内核移出。



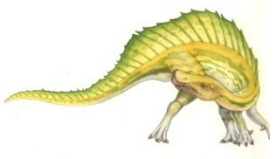
Monolithic Kernel

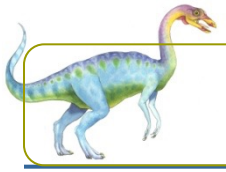




什么是内核模块

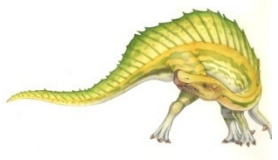
- 模块在内核空间运行，实际上是一种目标对象文件，没有链接，不能独立运行，但是其代码可以在运行时链接到系统中作为内核的一部分运行或从内核中取下，从而可以动态扩充内核的功能
- 这种目标代码通常由一组函数和数据结构组成,如用来实现一种文件系统、一个驱动程序或其他内核上层功能。
- 模块完整叫法：动态可加载内核模块（Loadable Kernel Module LKM）

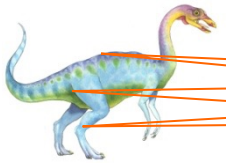




内核模块的优点

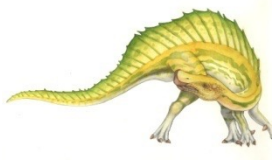
- 使得内核更加紧凑和灵活。
- 修改内核时，不必全部重新编译整个内核。系统如果需要使用新模块，只要编译相应的模块，然后使用insmod将模块插入即可。
- 模块不依赖于某个固定的硬件平台。
- 模块的目标代码一旦被链接到内核，它的作用域和静态链接的内核目标代码完全等价。





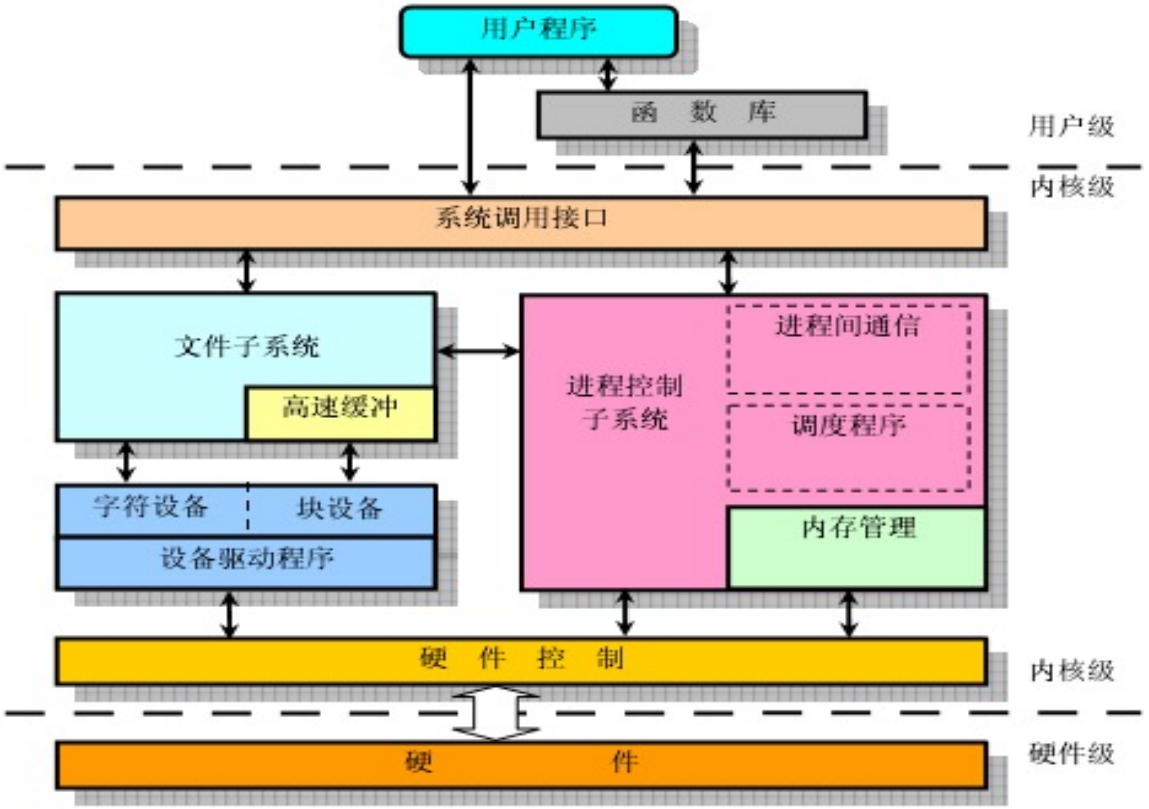
内核模块的缺点

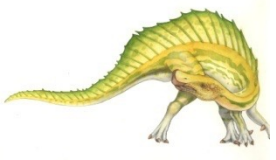
- 由于内核所占用的内存是不会被换出的，所以链接进内核的模块会给整个系统带来一定的性能和内存利用方面的损失。
- 装入内核的模块就成为内核的一部分，可以修改内核中的其他部分，因此，模块的使用不当会导致系统崩溃。
- 为了让内核模块能访问所有内核资源，内核必须维护符号表，并在装入和卸载模块时修改符号表。
- 模块会要求利用其它模块的功能，所以，内核要维护模块之间的依赖性。



内核模块概述

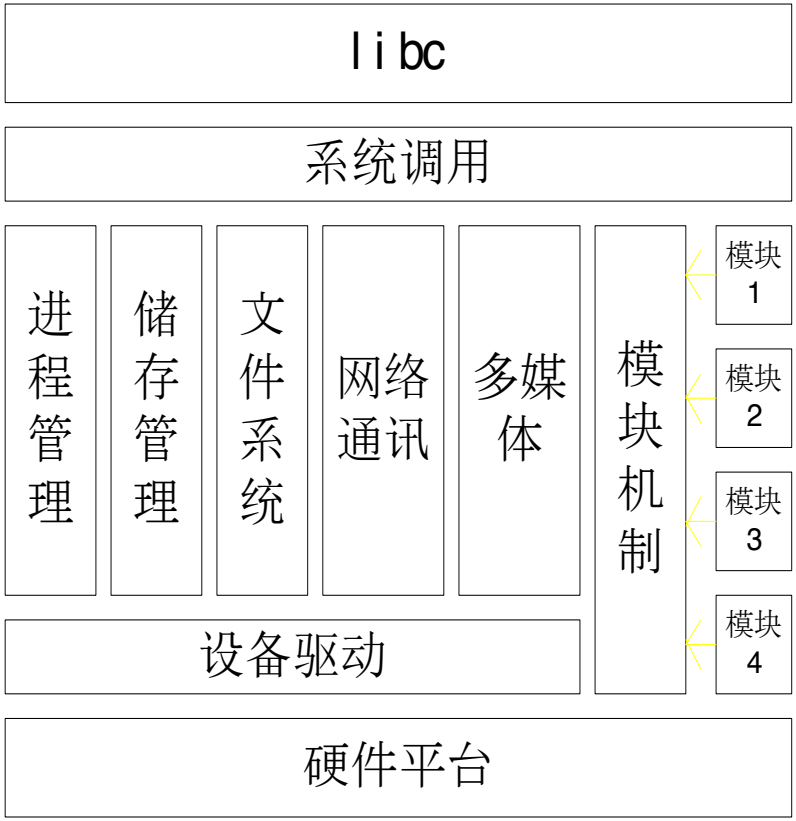
- Linux内核是整体式结构，各个子系统联系紧密，作为一个大程序在内核空间运行。





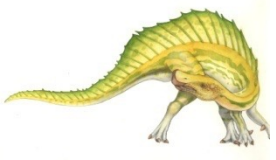
内核模块概述

- 太多的设备驱动和内核功能集成在内核中，内核过于庞大。如何解决？
  - Linux内核引入内核模块机制。通过动态加载内核模块，使得在运行过程中扩展内核的功能。不需要的时候，卸载该内核模块。



内核模块概述

- 内核模块是如何被调入内核工作的?
  - 当操作系统内核需要的扩展功能不存在时，内核模块管理守护进程kmod执行modprobe去加载内核模块。
  - modprobe遍历文件 /lib/modules/$(version)/modules.dep 来判断是否有其它内核模块需要在该模块加载前被加载。
  - 最后modprobe调用insmod先加载被依赖的模块，然后加载该被内核要求的模块。



内核模块概述

- 内核模块的卸载

  当我们不需要内核模块了,为了减少系统资源的开销,需要卸载时使用命令

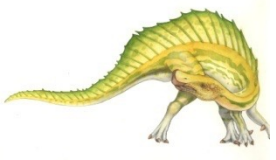
  #rmmod module_name

  或者

  #modprobe –r module_name

- 查看系统已经加载的模块,使用命令

  #lsmod

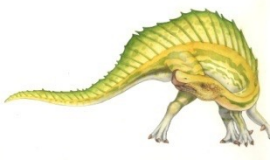


But First

- **Check whether your kernel is configured as below:**

```
Loadable module support  --->
  [*] Enable loadable module support
  [*]   Module unloading
  [ ]   Module versioning support
(EXPERIMENTAL)
  [*]   Automatic kernel module loading
```

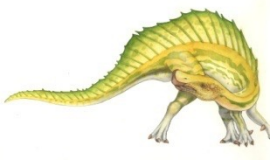
- **If not, reconfigure and recompile**



helloworld.c

helloworld.c

```
#define MODULE
#include <linux/module.h>
int init_module(void)  {
    printk("<1> Hello World!\n");
    return 0;
}
void cleanup_module(void) {
    printk("<1>Goodbye!\n");
}
MODULE_LICENSE("GPL");
```



helloworld.c

```
#include <linux/init.h>  // for module_init()
 #include <linux/module.h>  // must be include
 #include <linux/kernel.h>  // for printk()

 static int __init hello_init(void)
 {
        printk( “Hello world\n” );
        return 0;
 }
 static void __exit hello_exit(void)
 {
       printk( “Good Bye\n” );
 }
 module_init(hello_init);
 module_exit(hello_exit);

 MODULE_LICENSE( “GPL” );
```

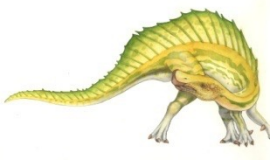
helloworld.c

- 在编译内核模块前，先准备一个Makefile文件：

```
TARGET = helloworld
KDIR = /lib/modules/$(shell uname -r)/build
PWD = $(shell pwd)
   obj-m += $(TARGET).o
default:
   make -C $(KDIR) M=$(PWD) modules
```

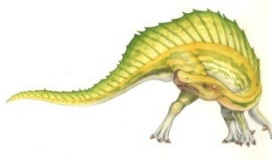
- 然后输入命令make：

```
#make
```



PrintK函数

- printk( )函数
  - printk 函数在 Linux 内核中定义并且对模块可用，为内核提供日志功能，记录内核信息或用来给出警告。与标准 C 库函数 printf 的行为相似。
  - 每个printk() 声明都会带一个优先级。内核总共定义了八个优先级的宏，在linux/kernel.h中定义。若你不指明优先级，DEFAULT_MESSAGE_LOGLEVEL这个默认优先级将被采用。
  - 信息添加到文件 /var/log/messages，可直接查看，或者用命令dmesg查看。在X－windows下的终端insmod一个模块，日志信息只会记录在日志文件中，而不在终端打印。



helloworld.c

- insmod命令加载模块

  # insmod helloworld.ko

  Hello World!

- 使用lsmod命令查看 模块

  # lsmod

  Module Size Used by

  helloworld 464 0 (unused)

  …

- rmmod 命令卸载模块

  # rmmod helloworld

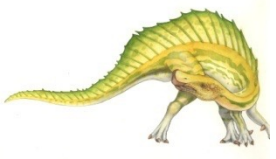
  Goodbye!

注意事项

写内核程序时需要注意

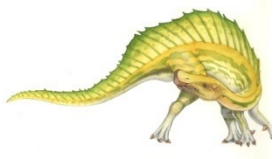
- 内核编程时不能访问C库。
- 内核编程时必须使用GNU C。
- 内核编程时缺乏像用户空间那样的内存保护机制。
- 内核编程时浮点数很难使用。
- 内核只有一个很小的定长堆栈。
- 由于内核支持异步中断、抢占和SMP，因此必须时刻注意同步和并发。
- 要考虑可移植性的重要性。

让我们仔细考察一下这些要点，所有这些东西在内核开发中必须时刻牢记。



其他

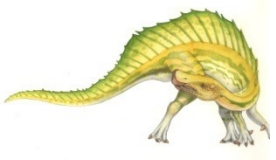
- 内核模块证书和内核模块文档说明
  - 2.4内核后，引入识别代码是否在GPL许可下发布的机制。在使用非公开的源代码产品时会得到警告。通过宏MODULE_LICENSE(“GPL”)，设置模块遵守GPL证书，取消警告信息。
  - 宏MODULE_DESCRIPTION()用来描述模块的用途。
  - 宏MODULE_AUTHOR()用来声明模块的作者。
  - 宏MODULE_SUPPORTED_DEVICE() 声明模块支持的设备。
  - 这些宏都在头文件linux/module.h定义。使用这些宏只是用来提供识别信息。



insmod 命令

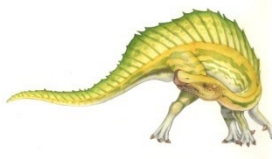
- 调用insmod程序将把需要插入的模块以目标代码的形式插入到内核中。
- 在插入的时候，insmod会自动运行在init_module()函数中定义的过程。注意，只有超级用户才能使用这个命令
- 格式：

  insmod [path]modulename.ko



insmod 命令

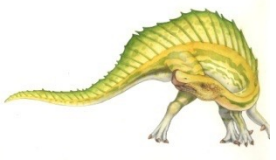
- insmod程序完成下面一系列工作：
  1. 从命令行中读入要链接的模块名，通常是扩展名为“.ko”，elf格式的目标文件。
  2. 确定模块对象代码所在文件的位置。通常这个文件都是在lib/modules的某个子目录中。
  3. 计算存放模块代码、模块名和module对象所需要的内存大小。
  4. 在用户空间中分配一个内存区，把module对象、模块名以及为正在运行的内核所重定位的模块代码拷贝到这个内存里。其中，module对象中的init域指向这个模块的入口函数重新分配到的地址；exit域指向出口函数所重新分配的地址。
  5. 调用init_module()，向它传递上面所创建的用户态的内存区的地址，其实现过程我们已经详细分析过了。
  6. 释放用户态内存，整个过程结束。



lsmod命令

- lsmod 显示当前系统中正在使用的模块信息。
- 实际上这个程序的功能就是读取/proc文件系统中的文件/proc/modules中的信息。所以这个命令和cat /proc/modules等价。
- 格式：

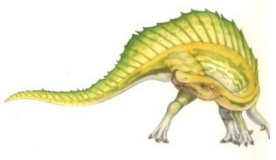
  lsmod



rmmod命令

- rmmod将已经插入内核的模块从内核中移出
- rmmod会自动运行在cleanup_module()函数中定义的过程
- 格式：

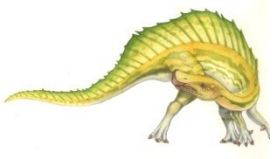
  rmmod [path]modulename



ksyms

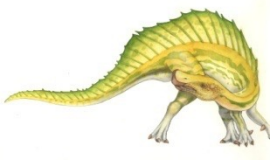
- 显示内核符号和模块符号表的信息，可以读取/proc/kallsyms文件。

  cat /proc/kallsyms



modprobe 命令

- modprobe是自动根据模块之间的依赖性插入模块的程序
- 按需装入的模块加载方法会调用这个程序来实现按需装入的功能。举例来讲，如果模块A依赖模块B，而模块B并没有加载到内核里，当系统请求加载模块A时，modprobe程序会自动将模块B加载到内核。



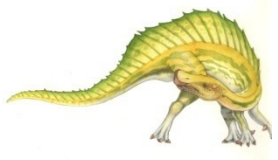
带参数的Hello World模块

- 模块参数
  - 内核允许对模块指定参数，这些参数可在装载模块时改变。在运行**insmod**或者**modprobe**命令时给出参数的值。

    insmod hellop.ko howmany=10 whom="Mom"
  - 如何定义实现模块参数呢？
  - 要传递参数给模块，首先将获取参数值的变量声明为全局变量。然后使用宏**module_param**来声明

    ```
    int myint = 3;

    module_param(myint, int，0);
    ```

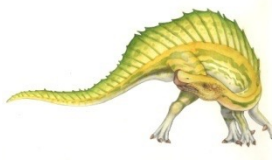


带参数的Hello World模块

- 模块参数
  - module_param(name,type,perm);

    perm是一个权限值,控制谁可以存取模块参数在 sysfs 中的表示。

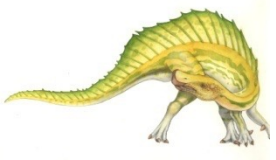
    perm 被设为 0, 就根本没有 sysfs 项
  - 这个宏定义应当放在任何函数之外, 典型地是出现在源文件的前面。
  - 应该总是为变量赋初值。



带参数的Hello World模块

- 模块参数
  - 宏MODULE_PARM_DESC() 用来注解该模块可以接收的参数。该宏两个参数：变量名和一个对该变量的描述。
  - 模块可以用这样的命令行加载：

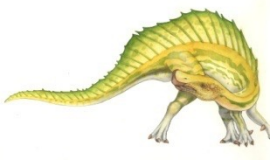
    ./insmod mymodule.ko myvariable=2



带参数的Hello World模块

- 模块参数
  - 声明一个数组参数：

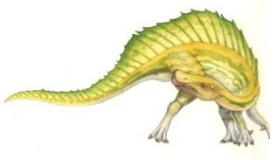
    module_param_array(name,type,num,perm);
    - name 数组的名子(也是参数名)
    - type 数组元素的类型
    - num 是数组元素的个数,模块加载者拒绝比数组能放下的多的值。2.6.9传递数组个数变量名，2.6.11传递数组个数变量的地址。
    - perm 是通常的权限值. 如果数组参数在加载时设置。



带参数的Hello World模块

- 参数数组的定义：
  - static int test[5] = {1,2,3,4,5};
  - static int num =5;
  - module_param(num,int,0);
  - module_param_array(test,int,num,0);
  - MODULE_PARM_DESC(test, "test array");
- 参数数组的加载方式：

insmod test.ko test=6,7,8,9,10 num=5



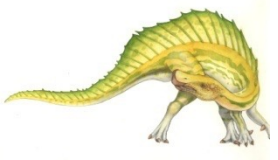
让你的Module变成Kernel的一部分

- 把模块加到Kernel的代码树:

选择一个需要把你的模块加入进去的Kernel子目录,如/$(KERNEL_DIR)/drivers/char, 在此子目录的Makefile中的适当的位置加入

```
obj-$(CONFIG_HELLO_WORLD) +=
  $(your_module_dir_name)/
```

这一句让Kernel被Kbuild系统编译的时候进入到您的模块的Makefile去执行



让你的Module变成Kernel的一部分

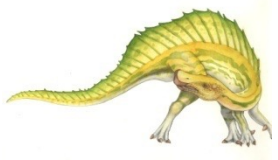
- 把模块加到Kernel的代码树:

在模块的目录中新建一个Kconfig文件,内容为

```
config  HELLO_WORLD
        tristate “A helloworld sample”
        default m
        help
            this is a test module sample for
            study, it just prints hello world on
            console
```

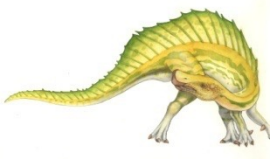
在/$(KERNEL_DIR)/drivers/char/Kconfig中的适当位置加入

```
source “drivers/char/$(your_module_dir_name)/Kconfig”
```



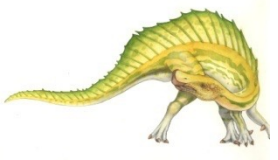
新模块的配置


话 编辑 查看 书签 设置 帮助
config - Linux/arm 2.6.38.5 Kernel Configuration
Character devices
Arrow keys navigate the menu. <Enter> selects sub
Highlighted letters are hotkeys. Pressing <Y> inc
<M> modularizes features. Press <Esc><Esc> to exi
for Search. Legend: [*] built-in [ ] excluded
<M> A hello sample (NEW)
[ ] Support for binding and unbinding console
[*] /dev/kmem virtual device support
[*] Non-standard serial port support
< > Computone IntelliPort Plus serial suppor
< > Comtrol RocketPort support

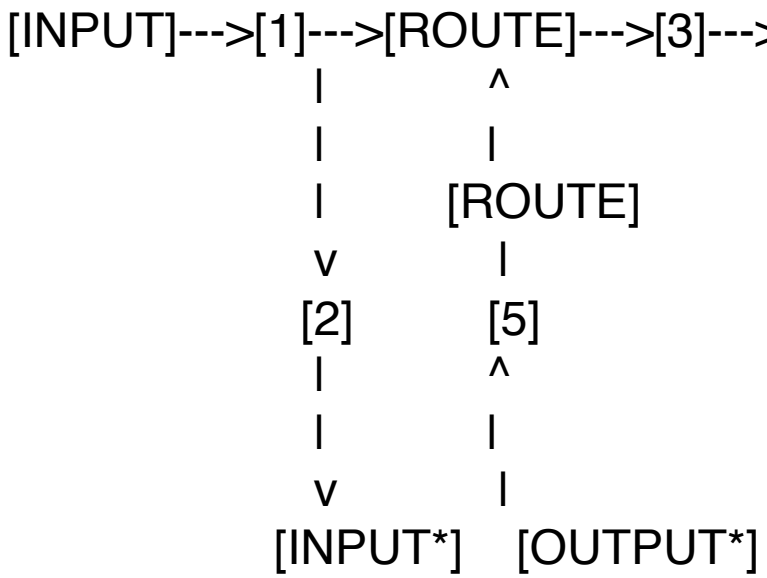


A Mini Firewall

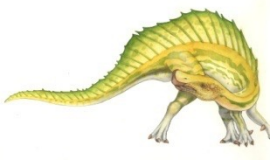
- Netfilter is a packet filtering subsystem in the Linux kernel stack and has been there since kernel 2.4.x. Netfilter's core consists of five hook functions declared in linux/netfilter_ipv4.h
- We try to overwrite the hooks defined in Netfilter



Network Hooks

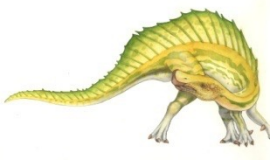


[1] NF_IP_PRE_ROUTING
[2] NF_IP_LOCAL_IN
[3] NF_IP_FORWARD
[4] NF_IP_POST_ROUTING
[5] NF_IP_LOCAL_OUT
[*] Network Stack



Hook Function Messages

- NF_ACCEPT: accept the packet (continue network stack trip)
- NF_DROP: drop the packet (don't continue trip)
- NF_REPEAT: repeat the hook function
- NF_STOLEN: hook steals the packet (don't continue trip)
- NF_QUEUE: queue the packet to userspace



Hook Registration

- **To register our own hook, we must use the structure defined in linux/netfilter.h**

```
struct nf_hook_ops
{
        struct list_head list;
        nf_hookfn *hook;
        int pf;
        int hooknum;
        int priority;
};
```

The Simple Version

```
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/netfilter.h>
#include <linux/netfilter_ipv4.h>
static struct nf_hook_ops netfilter_ops_in; /* NF_IP_PRE_ROUTING */
static struct nf_hook_ops netfilter_ops_out; /* NF_IP_POST_ROUTING */
/* Function prototype in <linux/netfilter> */
unsigned int main_hook(unsigned int hooknum,
                       struct sk_buff **skb,
                       const struct net_device *in,
                       const struct net_device *out,
                       int (*okfn)(struct sk_buff*))
{
return NF_DROP; /* Drop ALL Packets */
}
```

The Simple Version

```
int init_module()
{
        netfilter_ops_in.hook             =      main_hook;
        netfilter_ops_in.pf               =      PF_INET;
        netfilter_ops_in.hooknum          =      NF_IP_PRE_ROUTING;
        netfilter_ops_in.priority         =      NF_IP_PRI_FIRST;
        netfilter_ops_out.hook            =      main_hook;
        netfilter_ops_out.pf              =      PF_INET;
        netfilter_ops_out.hooknum         =      NF_IP_POST_ROUTING;
        netfilter_ops_out.priority        =      NF_IP_PRI_FIRST;
        nf_register_hook(&netfilter_ops_in); /* register NF_IP_PRE_ROUTING hook
*/
        nf_register_hook(&netfilter_ops_out); /* register NF_IP_POST_ROUTING
hook */
return 0;
}
void cleanup()
{
nf_unregister_hook(&netfilter_ops_in); /*unregister NF_IP_PRE_ROUTING hook*/
nf_unregister_hook(&netfilter_ops_out); /*unregister NF_IP_POST_ROUTING
hook*/
}
```

A Better One

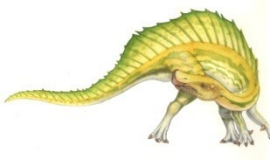
```
#include <linux/ip.h>
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/netdevice.h>
#include <linux/netfilter.h>
#include <linux/netfilter_ipv4.h>
#include <linux/skbuff.h>
#include <linux/udp.h>
static struct nf_hook_ops netfilter_ops;
static unsigned char *ip_address = "\xC0\xA8\x00\x01"; \\192.168.0.1
static char *interface = "lo";          \\loopback interface
unsigned char *port = "\x00\x17";  \\23, telnet port
struct sk_buff *sock_buff;
struct udphdr *udp_header;
```

A Better One

```
unsigned int main_hook(unsigned int hooknum,
                       struct sk_buff **skb,
                       const struct net_device *in,
                       const struct net_device *out,
                       int (*okfn)(struct sk_buff*))
{
  if(strcmp(in->name,interface) == 0){ return NF_DROP; }

  sock_buff = *skb;
  if(!sock_buff){ return NF_DROP; }
  if(!(sock_buff->nh.iph)){ return NF_DROP; }
  if(sock_buff->nh.iph->saddr == *(unsigned int*)ip_address){ return NF_DROP; }


if(sock_buff->nh.iph->protocol != 17){ return NF_DROP; }
udp_header = (struct udphdr *)(sock_buff->data + (sock_buff->nh.iph->ihl *4));
if((udp_header->dest) == *(unsigned short*)port){ return NF_DROP; }
return NF_ACCEPT;
}
```



A Better One

```
int init_module()
{
        netfilter_ops.hook              =       main_hook;
        netfilter_ops.pf                =       PF_INET;
        netfilter_ops.hooknum           =       NF_IP_PRE_ROUTING;
        netfilter_ops.priority          =       NF_IP_PRI_FIRST;
        nf_register_hook(&netfilter_ops);

return 0;
}
void cleanup_module() { nf_unregister_hook(&netfilter_ops); }
```