CS353 Linux Kernel

Chentao Wu吴晨涛
Associate Professor
Dept. of CSE, SJTU
wuct@cs.sjtu.edu.cn

2. Linux Kernel Programming Basic

Chentao Wu
Associate Professor
Dept. of CSE, SJTU
wuct@cs.sjtu.edu.cn

Outline

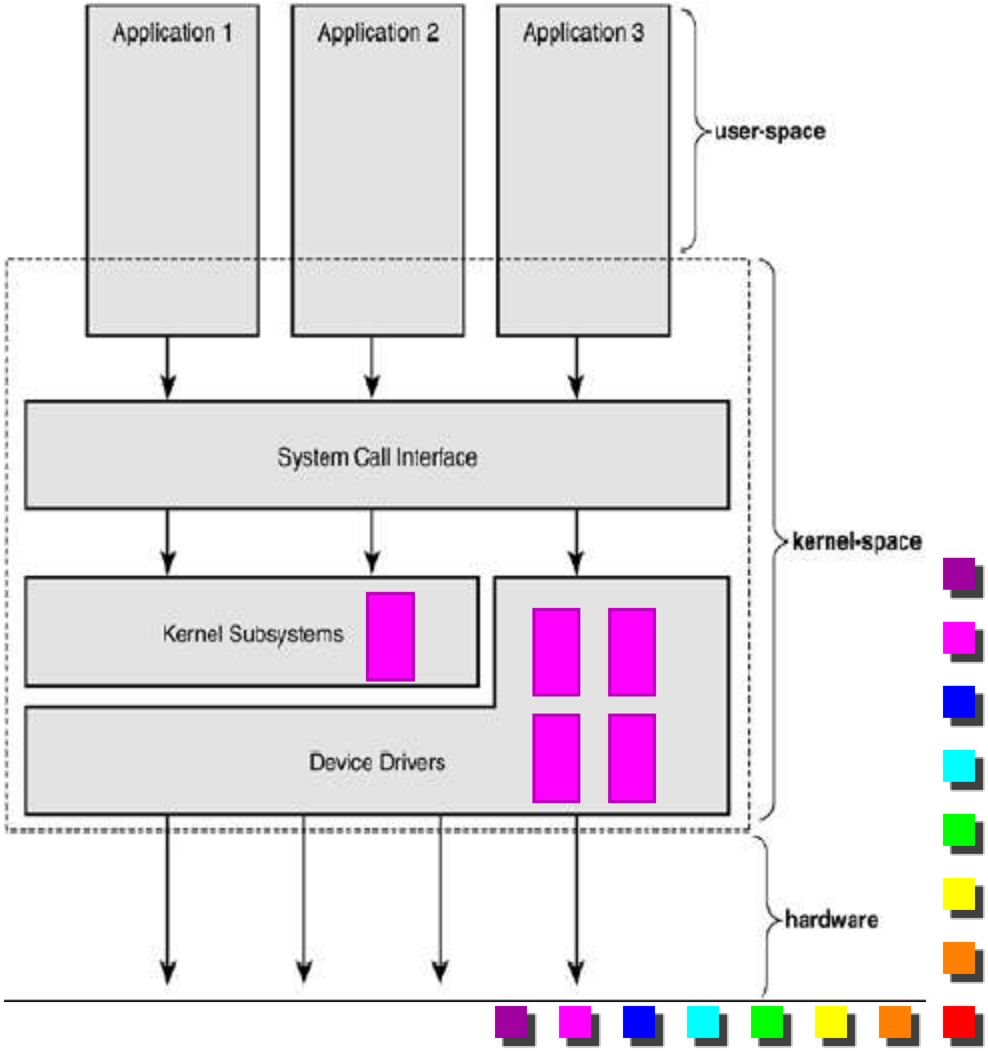
- **Linux Kernel Programming Basic**
- Introduction of the /proc File System
- Module Program in /proc File System

Linux Kernel Programming

- What makes kernel programming different?
  - OS management
    - Process management, memory management
    - File systems
  - Types of devices
    - (Char, Block, SCSI, Net)-based devices → device drivers
  - Loaded as modules or static in the kernel
  - Challenges
    - Portability
    - IPC (Inter-Process Communication)
    - Hardware Management
    - Interface Stability

Module

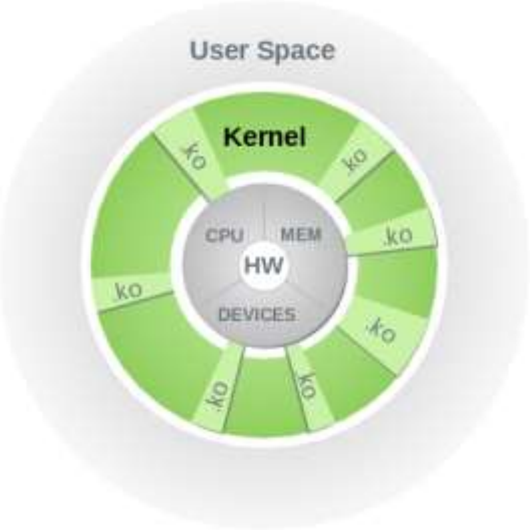
- What's a kernel module?
  - (wiki) An object file that contains code to extend the running kernel;
  - (RedHat) Modules are pieces of code that can be loaded and unloaded into the kernel upon demand.
  - Usage
  - Most current UNIX-like and Microsoft Windows Systems



Advantages and Disadvantages

- Why module? (Advantages)
  - Allowing the dynamic insertion and removal of code from the kernel at run-time.
  - Save memory cost
- Minor Disadvantage
  - Fragmentation Penalty → decrease memory performance

Current Kernel Modules



- # cd /lib/modules/2.6.32-22-generic/
- # find . -name "*.ko"

```
./kernel/arch/x86/kernel/microcode.ko
./kernel/arch/x86/kernel/msr.ko
./kernel/arch/x86/kernel/cpuid.ko
./kernel/arch/x86/kernel/cpu/mcheck/mce-xeon75xx.ko
./kernel/arch/x86/kernel/cpu/mcheck/mce-inject.ko
./kernel/arch/x86/kernel/cpu/cpufreq/p4-clockmod.ko
./kernel/arch/x86/kernel/cpu/cpufreq/speedstep-lib.ko
./kernel/arch/x86/kvm/kvm.ko
... and on and on and on ...
```

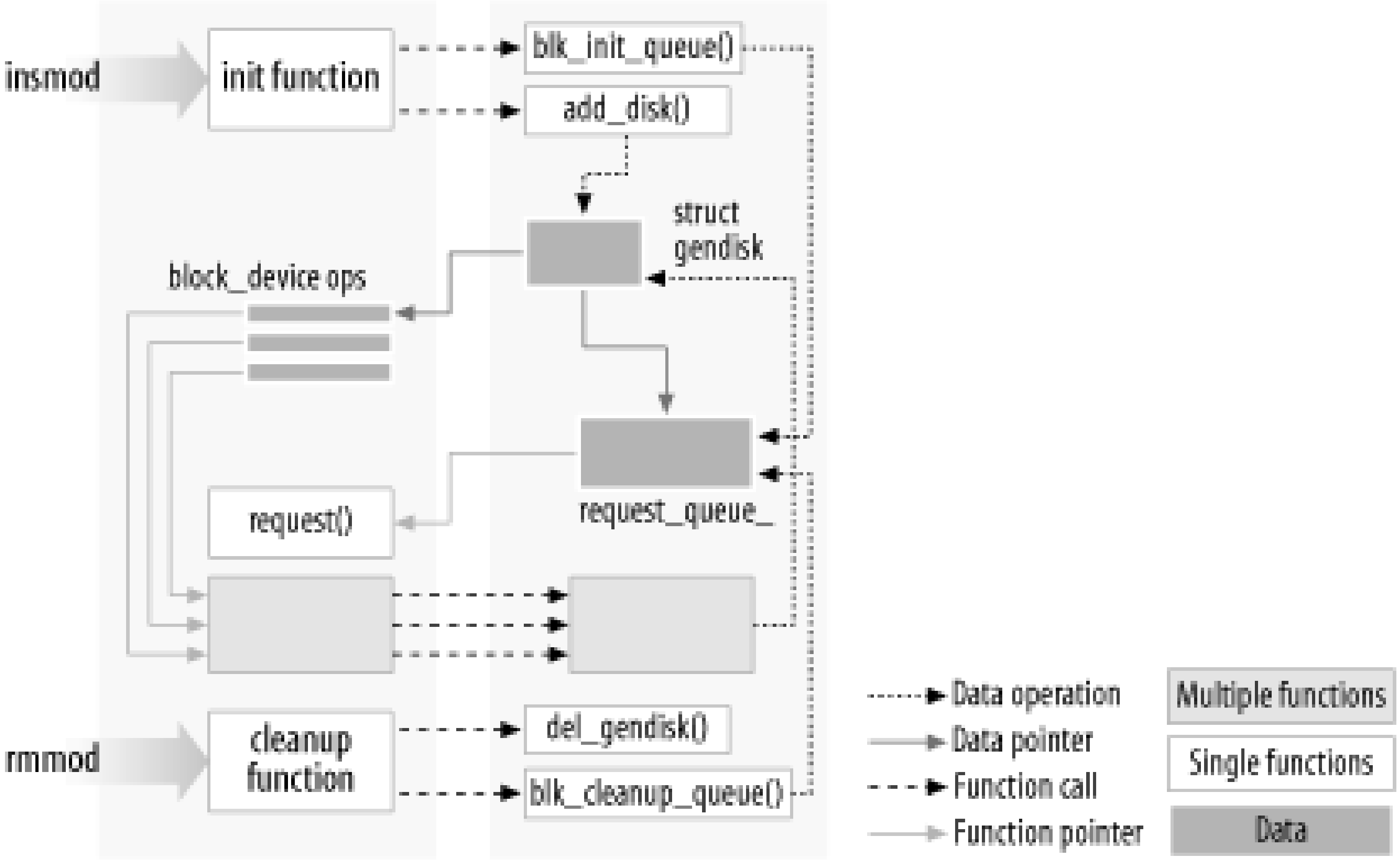
Current Loadable Modules

- # lsmod

```
--------------------------------------------------------------------------------
Module                           Size    Used by
cryptd                           8116      0
aes_x86_64                       7912      1
aes_generic                     27607      1
aes_x86_64 nls_iso8859_1         4633      1
nls_cp437                        6351      1
vfat                            10802      1
fat                             55350      1vfat
binfmt_misc                      7960      1
ppdev                            6375      0
... etc. ...
--------------------------------------------------------------------------------
```

- # cat /proc/modules

An Example of A Module Design



A Simple Module

```
#include <linux/kernel.h>   /* Needed for KERN_INFO */
#include <linux/module.h>  /* Needed by all modules */
#include <linux/init.h>  /*Needed for init and cleanup functions*/
Entrance→          static int __init hello_init(void)
                   {
                   printk(KERN_INFO "Hello world\n");
                   return 0;
                   }
Exit→              static void __exit hello_exit(void)
                   {
                   printk(KERN_INFO "Goodbye world\n");
                   }
                   module_init(hello_init);
                   module_exit(hello_exit);
```

loglevels:

KERN_EMERG = “<0>”
KERN_ALERT
KERN_CRIT
KERN_ERR
KERN_WARNING
KERN_NOTICE
KERN_INFO
KERN_DEBUG

A Real Module in VFS (Minix FS)

```
/* linux/fs/minix/inode.c
 ......*/
#include <linux/module.h>
#include "minix.h"
#include <linux/buffer_head.h>
#include <linux/slab.h>
#include <linux/init.h>
#include <linux/vfs.h>
......
static struct file_system_type minix_fs_type = {
.owner = THIS_MODULE,
.name = "minix",
.get_sb = minix_get_sb,
.kill_sb = kill_block_super,
.fs_flags = FS_REQUIRES_DEV,
};
```

```
static int __init init_minix_fs(void)
{
int err = init_inodecache();
if (err)
goto out1;
err = register_filesystem(&minix_fs_type);
if (err)
goto out;
return 0;
out:
destroy_inodecache();
out1:
return err;
}
static void __exit exit_minix_fs(void)
{
unregister_filesystem(&minix_fs_type);
destroy_inodecache();
}
```

Compiling Modules (1)

- Makefile
  - obj-m:=hello.o

- Multiple source files
  - obj-m:=hello.o
  - hello-objs := a.o b.o

hello.c→hello.o

```
obj-m:=hello.o
all:
	make -C /lib/modules/$(shell uname -r)/build M=$(shell pwd) modules
clean:
	make -C /lib/modules/$(shell uname -r)/build M=$(shell pwd) clean
```

This command starts by changing its directory to the one provided with the –C option (that is, your kernel source directory). There it finds the kernel’s top-level makefile. The M= option causes that makefile to move back into your module source directory before trying to build the modules target. This target, in turn, refers to the list of modules found in the obj-m variable.

-- Linux Device Drivers (3rd Edition)

Compiling Modules (2)

- Run the Module

Command:

```
# make
# make clean
```

Another Type of Makefile

homework1.c→homework

```
obj-m := homework1.o
KDIR := /lib/modules/$(shell uname -r)/build
PWD := $(shell pwd)
all:
	make -C $(KDIR) M=$(PWD) modules
clean:
	rm *.o *.ko *.mod.c Module.symvers
modules.order -f
```

Commands For Modules

- #insmod hello.ko /*insert a module*/
- #rmmod hello.ko /*delete a module*/
- #lsmod /*list the current modules*/
- #modinfo hello.ko /*list the information of a module*/
- #modprobe /*insert or delete a module, and handle the dependencies between modules*/

Module Parameters

```
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/init.h>
#include <linux/moduleparam.h>
static int test;
module_param(test, int, 0644);
void hello_foo(void)
{
printk("Hello\n");
}
EXPORT_SYMBOL(hello_foo);

static int __init hello_init(void)
{
printk(KERN_INFO "Hello world\n");
printk(KERN_INFO "Params: test:%d;\n",test);
return 0;
}
```

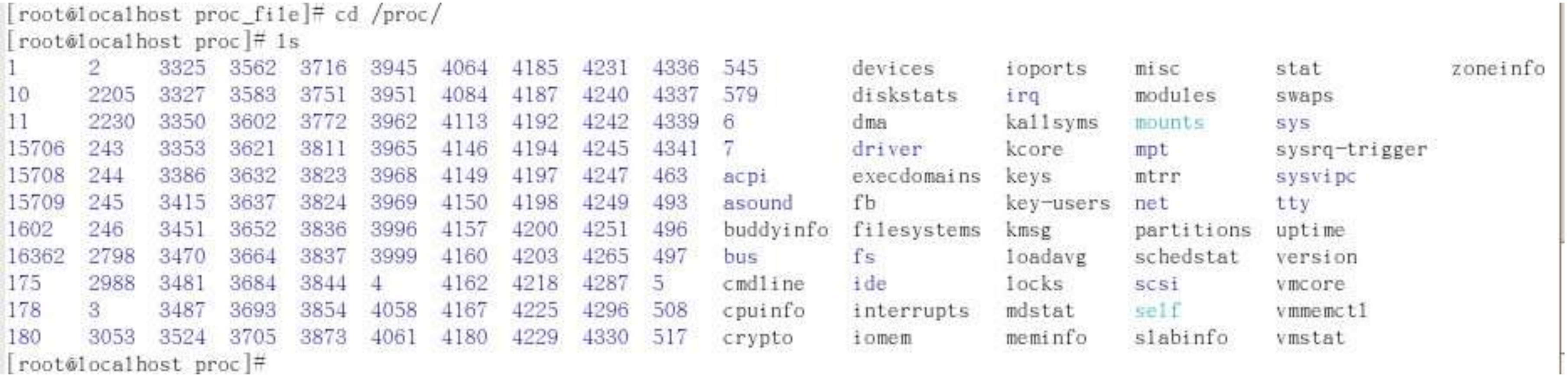
```
static void __exit hello_exit(void)
{
printk(KERN_INFO "Goodbye world\n");
}
MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("Test");
MODULE_AUTHOR("xxx");
module_init(hello_init);
module_exit(hello_exit);
```

```
#insmod hello.ko test=10
```

Outline

- Linux Kernel Programming Basic
- **Introduction of the /proc File System**
- Module Program in /proc File System

The /proc



```
[root@localhost proc_file]# cd /proc/
[root@localhost proc]# ls
1      2     3325  3562  3716  3945  4064  4185  4231  4336  545        devices      ioports    misc        stat           zoneinfo
10     2205  3327  3583  3751  3951  4084  4187  4240  4337  579        diskstats    irq        modules     swaps
11     2230  3350  3602  3772  3962  4113  4192  4242  4339  6          dma          kallsyms   mounts      sys
15706  243   3353  3621  3811  3965  4146  4194  4245  4341  7          driver       kcore      mpt         sysrq-trigger
15708  244   3386  3632  3823  3968  4149  4197  4247  463   acpi       execdomains  keys       mtrr        sysvipc
15709  245   3415  3637  3824  3969  4150  4198  4249  493   asound     fb           key-users  net         tty
1602   246   3451  3652  3836  3996  4157  4200  4251  496   buddyinfo  filesystems  kmsg       partitions  uptime
16362  2798  3470  3664  3837  3999  4160  4203  4265  497   bus        fs           loadavg    schedstat   version
175    2988  3481  3684  3844  4     4162  4218  4287  5     cmdline    ide          locks      scsi        vmcore
178    3     3487  3693  3854  4058  4167  4225  4296  508   cpuinfo    interrupts   mdstat     self        vmmemctl
180    3053  3524  3705  3873  4061  4180  4229  4330  517   crypto     iomem        meminfo    slabinfo    vmstat
[root@localhost proc]#
```

The /proc

- A pseudo file system
- Real time, resides in the virtual memory
- Tracks the processes running on the machine and the state of the system
- A new /proc file system is created every time your Linux machine reboots
- Highly dynamic. The size of the proc directory is 0 and the last time of modification is the last bootup time.

Other features

- /proc file system doesn't exist on any particular media.
- The contents of the /proc file system can be read by anyone who has the requisite permissions.
- Certain parts of the /proc file system can be read only by the owner of the process and of course root. (and some not even by root!!)
- The contents of the /proc are used by many utilities which grab the data from the particular /proc directory and display it.
- eg : top, ps, lspci ,dmesg etc

Tweak kernel parameters

- /proc/sys : Making changes in this directory enables you to make real time changes to certain kernel parameters.
- eg : /proc/sys/net/ipv4/ip_forward
- It has default value of "0" which can be seen using 'cat'.
- This can be changed in real time by just changing the value stored in this file from "0" to "1", thus allowing IP forwarding

Files in /proc

- buddyinfo
- cmdline
- cpuinfo
- crypto
- devices
- diskstats
- dma
- execdomains
- fb
- filesystems
- interrupts
- iomem
- ioports
- kcore
- kmsg
- loadavg
- locks
- mdstat
- meminfo
- misc
- modules
- mounts
- mtrr
- partitions
- pci
- self
- slabinfo
- stat
- swaps
- sysrq-trigger
- uptime
- version
- vmstat

Details of some files in /proc

- buddyinfo

  Contains the number of free areas of each order for the kernel buddy system
- cmdline

  Kernel command line
- cpuinfo

  Information about the processor(s).(Human readable)
- devices

  List of device drivers configured into the currently running kernel (block and character).
- dma

  Shows which DMA channels are being used at the moment.
- execdomains

  Execdomains, related to security

Details of some files in /proc

- fb

  Frame Buffer devices.

- filesystems

  Filesystems configured/supported into/by the kernel.

- interrupts

  Number of interrupts per IRQ on the x86 architecture.

- iomem

  This file shows the current map of the system's memory for its various devices

- ioports

  provides a list of currently registered port regions used for input or output communication with a device

/proc/kcore

- This file represents the physical memory of the system and is stored in the core file format.

- Unlike most /proc files, kcore does display a size. This value is given in bytes and is equal to the size of physical memory (RAM) used plus 4KB.

- Its contents are designed to be examined by a debugger, such as gdb, the GNU Debugger.

- Only the root user has the rights to view this file.

Details of some files in /proc (1)

- kmsg
  - Used to hold messages generated by the kernel. These messages are then picked up by other programs, such as `klogd`
- loadavg
  - Provides a look at load average
  - The first three columns measure CPU utilization of the last 1, 5, and 10 minute periods.
  - The fourth column shows the number of currently running processes and the total number of processes.
  - The last column displays the last process ID used.
- locks
  - Displays the files currently locked by the kernel

Details of some files in /proc (2)

- mdstat
  - contains the current information for multiple-disk, RAID configurations
- meminfo
  - One of the more commonly used /proc files
  - It reports back plenty of valuable information about the current utilization of RAM on the system
- misc
  - This file lists miscellaneous drivers registered on the miscellaneous major device, which is number 10
- modules
  - Displays a list of all modules that have been loaded by the system
- mounts
  - This file provides a quick list of all mounts in use by the system

Details of some files in /proc (3)

- mtrr
  - This file refers to the current Memory Type Range Registers (MTRRs) in use with the system
- partitions
  - Very detailed information on the various partitions currently available to the system
- pci
  - Full listing of every PCI device on your system
- slabinfo
  - Information about memory usage on the slab level
- stat
  - Keeps track of a variety of different statistics about the system since it was last restarted

Details of some files in /proc (4)

- swap
  - Measures swap space and its utilization
- uptime
  - Contains information about how long the system has on since its last
  - restart
- version
  - Tells the versions of the Linux kernel and gcc, as well as the version of
  - Red Hat Linux installed on the system.

The numerical named directories

- The various directories in /proc are the processes that were running at the instant a snapshot of the /proc file system was taken.
- The contents of all the directories are the same as these directories contain the various parameters and the status of the corresponding process.
- You have full access only to the processes that you have started.

A typical process directory

- cmdline : it contains the whole command line used to invoke the process. The contents of this file are the command line arguments with all the parameters (without formatting/spaces).
- cwd : symbolic link to the current working directory
- environ : contains all the process-specific environment variables
- exe : symbolic link of the executable
- maps : parts of the process' address space mapped to a file.

A typical process directory(contd.)

- fd : this directory contains the list file descriptors as opened by the particular process.
- root : symbolic link pointing to the directory which is the root file system for the particular process
- status : information about the process

Other Subdirectories in /proc

- /proc/self : link to the currently running process
- /proc/bus : contains information specific to the various buses available on the system
  - eg : for ISA, PCI, and USB buses, current data on each is available in /proc/bus/<bus type directory>
  - Individual bus directories, signified with numbers, contains binary files that refer to the various devices available on that bus
  - *devices* file : USB root hub on the motherboard:

Subdirectories (cont...)

- /proc/driver : specific drivers in use by kernel
  - rtc : output from the driver for the Real Time Clock
- /proc/fs : specific filesystem, file handle, inode, dentry and quota information
- /proc/ide : information about IDE devices
  - Each IDE channel is represented as a separate directory, such as /proc/ide/ide0 and /proc/ide/ide1
  - *drivers* file : version number of the various drivers
  - Device directories : data like cache, capacity, driver, geometry, media, model, settings

Subdirectories (cont...)

- /proc/irq : used to set IRQ to CPU affinity
  - smp_affinity : which CPUs handle that specific IRQ
- /proc/net : networking parameters and statistics
  - arp — kernel's ARP table. Useful for connecting hardware address to an IP address on a system.
  - dev — Lists the network devices along with transmit and receive statistics.
  - route — Displays the kernel's routing table.
- /proc/scsi : like /proc/ide it gives info about scsi devices

/proc/sys

- allows you to make configuration changes to a running kernel
- Changing a value within a /proc/sys file is done by the 'echo' command
- Any configuration changes made thus will disappear when the system is restarted

/proc/sys subdirectories

- /proc/sys/dev : provides parameters for particular devices on the system
  - cdrom/info : many important CD-ROM parameters
- /proc/sys/fs
- /proc/sys/kernel
  - acct — Controls the suspension of process accounting based on the percentage of free space available on the filesystem containing the log

/proc/sys subdirectories (cont....)

- ctrl-alt-del — Controls whether [Ctrl]-[Alt]-[Delete] will gracefully restart the computer using init (value 0) or force an immediate reboot without syncing the dirty buffers to disk (value 1).
- domainname — Allows you to configure the system's domain name, such as domain.com.
- hostname — Allows you to configure the system's host name, such as host.domain.com.
- threads-max — Sets the maximum number of threads to be used by the kernel, with a default value of 4095.

/proc/sys subdirectories (cont....)

  - The random directory data related to generating random numbers for the kernel.
  - panic — Defines the number of seconds the kernel will postpone rebooting the system when a kernel panic is experienced. By default, the value is set to 0, which disables automatic rebooting after a panic.
- /proc/sys/net
- /proc/sys/vm : facilitates the configuration of the Linux kernel's virtual memory (VM) subsystem

/proc File System Entries

- To use any of the procfs functions, you have to include the correct header file! #include <linux/proc_fs.h>
- struct proc_dir_entry* **create_proc_entry**(const char* *name*, mode_t *mode*, struct proc_dir_entry* *parent*);
  - This function creates a regular file with the name *name*, the mode *mode* in the directory *parent*.
  - To create a file in the root of the procfs, use NULL as *parent* parameter.
  - When successful, the function will return a pointer to the freshly created *struct proc_dir_entry*
  - *foo_file = create_proc_entry("foo", 0644, example_dir);*

Creating a Directory and a Symlink

- *struct proc_dir_entry** ***proc_mkdir****(const char* name, struct proc_dir_entry* parent);*
  - Create a directory *name* in the procfs directory *parent*.
- struct proc_dir_entry* **proc_symlink**(const char* *name*, struct proc_dir_entry* *parent*, const char* *dest*);
  - This creates a symlink in the procfs directory *parent* that points from *name* to *dest*. This translates in userland to ln -s *dest name*.

Removing an Entry

- void **remove_proc_entry**(const char* *name*, struct proc_dir_entry* *parent*);
  - Removes the entry *name* in the directory *parent* from the procfs.
  - Be sure to free the *data* entry from the struct proc_dir_entry before remove_proc_entry is called

Advantages & Disadvantages of the /proc File System

- Advantages
  - Coherent, intuitive interface to the kernel
  - Great for tweaking and collecting status info
  - Easy to use and program for
- Disadvantages
  - Certain amount of overhead, must use fs calls
    - Alleviated somewhat by sysctl() interface
  - User can possibly cause system instability

Outline

- Linux Kernel Programming Basic
- Introduction of the /proc File System
- **Module Program in /proc File System**

Creating the /proc file in modules (1)

read_write:

create file: /proc/net/hyperCard

```
struct proc_dir_entry *create_proc_entry(
const char *name,
mode_t mode,
struct proc_dir_entry *parent )
```

```
struct proc_dir_entry *entry = NULL;
entry = create_proc_entry("hyperCard",
S_IFREG|S_IRUGO|S_IWUSR, &proc_net);
if (!entry) {
printk(KERN_ERR "unable to create /proc/net/hyperCard\n");
return -EIO;
} else {
entry->read_proc = hypercard_proc_read;
entry->write_proc = hypercard_proc_write;
}
```

Creating the /proc file in modules (2)

```
struct proc_dir_entry {
unsigned int low_ino;
unsigned short namelen;
const char *name;
mode_t mode;
nlink_t nlink;
uid_t uid;
gid_t gid;
loff_t size;
struct inode_operations * proc_
const struct file_operations *
get_info_t *get_info;
struct module *owner;
struct proc_dir_entry *next, *p
void *data;
read_proc_t *read_proc;
write_proc_t *write_proc;
atomic_t count; /* use
int deleted; /* dele
void *set;
};
```

Creating the /proc file in modules (3)

read-only:

create file: /proc/test-file

struct proc_dir_entry *create_proc_read_entry

(

const char *name, → **"test-file"**

mode_t mode, → **the access mode of the file (0444)**

struct proc_dir_entry *base, → **the location of the file (NULL)**

read_proc_t *read_proc, → **function called when the file is read**

void *data → **data used by read_proc function**

);

Creating the /proc file in modules (4)

```
int (*read_proc)(char *page, char **start, off_t offset, int count, int
      *eof, void *data);
```

- →page   points to the buffer to write your data
- →start   is used only if your data is beyond one page
- →offset   means where to start from
- →count   bytes to read
- →eof   points to an integer that is set by the driver to indicate EOF
- →data   driver-specific data pointer for internal bookkeeping, return the number of bytes actually placed in the page buffer

Creating the /proc file in modules (5)

linux/proc_fs.h

```
struct proc_dir_entry *create_proc_read_entry(...)
```

```
struct proc_dir_entry {
unsigned int low_ino;
unsigned short namelen;
const char *name;
mode_t mode;
nlink_t nlink;
uid_t uid;
gid_t gid;
loff_t size;
struct inode_operations * proc_iops;
```

Creating the /proc file in modules (6)

```
const struct file_operations * proc_fops;
get_info_t *get_info;
struct module *owner;
struct proc_dir_entry *next, *parent, *subdir;
void *data;
read_proc_t *read_proc;
write_proc_t *write_proc;
atomic_t count; /* use count */
int deleted; /* delete flag */
void *set;
};
```

Example (1)

```
#include <linux/module.h>
#include <linux/init.h>
#include <linux/proc_fs.h>
static int hello_call (char *buf, char **start, off_t off, int count, int *eof, void *data)
{
printk(KERN_INFO "This is a test for proc file.\n");
int len = sprintf(buf, "%s", "Message from a Linux kernel module.\n");
return len;
}
static int __init hello_init (void)
{
struct proc_dir_entry *entry;
entry = create_proc_read_entry ("hello", 0444, NULL, hello_call, NULL);
if(!entry)
return -1;
else{
Printk (KERN_INFO "Proc_read_entry created successfully.\n");
return 0;
}
}
```

Example (2)

```
static void __exit hello_exit(void)
{
remove_proc_entry("hello", NULL);
printk(KERN_INFO "Goodbye.\n");
}
module_init(hello_init);
module_exit(hello_exit);
MODULE_LICENSE("GPL");
```

```
struct proc_dir_entry *create_proc_read_entry
(
const char *name,
mode_t mode,
struct proc_dir_entry *base,
read_proc_t *read_proc,
void *data
);
```

Other Functions

- proc_mkdir creates a new directory.
- proc_mkdir_mode creates a new directory whose access mode can be explicitly specified.
- proc_symlink generates a symbolic link.
- remove_proc_entry deletes a dynamically generated entry from the proc directory.

```
struct proc_dir_entry *create_proc_read_entry
(
const char *name,
mode_t mode,
struct proc_dir_entry *base,
read_proc_t *read_proc,
void *data
);
struct proc_dir_entry *create_proc_entry
( const char *name,
mode_t mode,
struct proc_dir_entry *parent
);
```

Printk()

```
int printk(const char * fmt,…);
```

```
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/init.h>
static int __init hello_init(void)
{
printk(KERN_INFO "Hello world\n");
return 0;
}
static void __exit hello_exit(void)
{
printk(KERN_INFO "Goodbye world\n");
}
module_init(hello_init);
module_exit(hello_exit);
```

Output the text:

```
#dmesg | tail -5
```

loglevels:

```
--------------------
KERN_EMERG = “<0>”
KERN_ALERT
KERN_CRIT
KERN_ERR
KERN_WARNING
KERN_NOTICE
KERN_INFO
KERN_DEBUG
```

Modules VS. Programs

Modules

- module_init() and module_exit() functions
- Avaialble functions
  - printk()
  - Kernel Space
  - Written in C (typically)

Programs

- main() function
- Avaialble functions
  - printf()
  - User Space
  - Written in C, C++, Java, etc.

Tips

- Modules can only use APIs exported by kernel and other modules
  - No libc
  - Kernel exports some common APIs
- Modules run in ring 0
  - Security holes
- Modules are part of kernel
  - Modules can control the whole system
  - As a result, damage the whole system
- Modules basically can't be written with C++
- Determine which part should be in kernel

Commands Frequently Used (1)

Create a patch

a) Assume under directory /root/work

b) Copy the source tree of kernel here, or untar the source codes.

Assume the tree is linux-2.6.17;

c) Copy the source tree by #cp -r linux-2.6.17 linux-2.6.17-fix;

d) Change files under linux-2.6.17-fix;

e) After changes are done; #cd /root/work

f) #diff -Nraup linux-2.6.17 linux-2.6.17-fix>patch_name.patch

g) If you once build kernel under your new tree,

pls. delete the obj by #make mrproper before run #diff

Apply a patch

a) Assume your working directory of new kernel is /root/work/new_tree;

b) #cd /root/work/new_tree

c) #patch -p1</root/work/patch_name.patch

Commands Frequently Used (2)

Create cross-index for code lookup: two approaches

a) Tags:

i. #cd /root/work/linux-2.6.17

ii. #make tags #you just need run it once

iii.When you look for a symbol/variable/definition,
run #vim -t definition_name.

b) cscope:

i. buildup cscope database: (Only need once)

1. #cd /root/work/linux-2.6.17

2. #find . -name "*.[chS]">files

3. #find . -name "*Makefile*">>files

4. #cscope -bk -f cc.db -I files

ii. Lookup symbol:

1. #cscope -d -f cc.db -p10

2. You will see a small menu. Query symbol/file/head_files

Project 2A:
Module Program

Module program

- Compile a kernel and run it in the system
  - Module 1

    – Load/unload the module can output some info
  - Module 2

    – Module accepts a parameter (an integer)

    – Load the module, output the parameter's value
  - Module 3

    – Module creates a proc file, reading the proc file returns some info
- More detailed information is shown in Student Experimental Handbook.