Kernel Modules

Operating Systems

System Calls

Standard interface to allow the kernel to safely handle user requests.

Examples:

- Read from hardware.
- Spawn a new process.
- Get current time.
- Create shared memory.

strace

Linux system call tracer – used to monitor interaction between processes and kernel

- Lots of options available for diagnostics and debugging
  - `-o outfile` saves output to outfile
  - `man strace` for more information

Example:

```
strace -o [outfile] [executable]
```

Part 1

Create an empty C program called "empty.c".

```
> strace -o empty.trace empty
```

Create a new C program with 5 additional system calls called "part1.c"

```
> strace -o part1.trace part1
```

Part1.trace should have 5 more syscalls entries than empty.trace

Kernel

Kernel is responsible for enabling multiple applications to run simultaneously, managing system resources, and interact directly with the hardware.

- Source code is stored in /usr/src
- Kernel image gets installed to /boot/vm/linux-<kernel-name>

Kernel Modules

Kernel modules are portion of the kernel that can be dynamically loaded and unloaded.

Examples:

- USB drivers
- File system drivers
- Disk drivers
- Cryptographic libraries

Why Kernel Modules?

Not every machine needs the same modules

- Different machines uses different drivers

Load only the components that you need

- Smaller system footprint
- Quicker boot time

Dynamically load modules for new devices

- New USB, camera, printer, graphics card, motherboard, etc.

Kernel Programming

Kernel Modules are event-driven.

- Register functions.
- Wait for requests from user-space and service them.
- Server/Client module.

No standard C libraries → Kernel libraries instead.

No floating point support.

Crashes/deadlocks in a module can crash the entire kernel.

- Requires system-wide reboot.

Part 2

Create a kernel module called `my_timer` that displays elapsed time .

- You will create a kernel module that creates a proc file "/proc/timer"
- Every read to the "/proc/timer" will print the current and elapsed time since the last call.
- The file should be removed if the module is unloaded.
- You will only have to use .proc_read to copy data to user space for users.
- Disregard using .proc_write!

You should create the /proc/timer proc file.

Part 2

```
$ cat /proc/timer
current time: 1518647111.760933999

$ sleep 1

$ cat /proc/timer
current time: 1518647112.768429998
elapsed time: 1.007495999

$ sleep 3

$ cat /proc/timer
current time: 1518647115.774925999
elapsed time: 3.006496001

$ sleep 5

$ cat /proc/timer
current time: 1518647120.780421999
elapsed time: 5.005496000
```

Procfs

For security, users cannot access the kernel space directly, as well as the data/info generated in the kernel. Special file system used for information related to the system and it processes.

- Mounted at /proc.
- Acts as an interface to the kernel.
- To see different processes using procfs, run:

```
> ll /proc
```

We will be using the proc file system for Part 2 and Part 3 as the primary interface for you to output to.

We will not be using the kernel log (printk)!

Procfs and Processes

/proc/PID/cmdline

- The command that originally started the process

/proc/PID/cwd

- A symlink to the current working directory of the process

/proc/PID/exe

- A symlink to the original executable file, if it still exists

/proc/PID/fd

- A directory containing a symbolic link for each open file descriptor

Procfs and Processes

/proc/PID/maps

- A text file containing information about mapped files and blocks (like a heap and stack)

/proc/PID/status

- Contains basic information about a process including its run state and memory usage

Hello World Procfs Module

We will create a proc entry /proc/hello upon module load.

- Support read/write to proc entry.
- We can output redirect to /proc/hello to virtually update the /proc/hello file so that users can read from it.
- Remove proc entry upon module exit.
- Demonstrate both .proc_read and .proc_write.

You should **disregard using .proc_write** in your project. This is only a demonstration.

Headers

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/proc_fs.h>
#include <linux/uaccess.h>
```

Kernel functions for procfs

Memory copy from kernel <…> userspace.

Globals

```
MODULE_LICENSE("GPL");
MODULE_AUTHOR("cop4610t");
MODULE_DESCRIPTION("A simple Linux kernel module");
MODULE_VERSION("1.0");

#define ENTRY_NAME "hello"
#define PERMS 0666
#define PARENT NULL

#define BUF_LEN 100
static struct proc_dir_entry* proc_entry;
static char msg[BUF_LEN];
static int procfs_buf_len;
```

File created in /proc (e.g., /proc/hello)

Max length of read/write message

Pointer to proc entry

Buffer to store read/write message

Variable to hold length of message

Procfile Read

Function called to read /proc file.

```
static ssize_t procfile_read(struct file* file, char* ubuf, size_t count, loff_t
*ppos) {
    printk(KERN_INFO "proc_read\n");
    procfs_buf_len = strlen(msg);
    if (*ppos > 0 || count < procfs_buf_len)
        return 0;

    if (copy_to_user(ubuf, msg,procfs_buf_len))
        return -EFAULT;
    *ppos = procfs_buf_len;
    printk(KERN_INFO "gave to user %s\n", msg);

    return procfs_buf_len;
};
```

Check if data is already read and if space is big enough in user buffer.

Copy data to user buffer.

Update offset.

Return number of characters read.

Procfile Write

Function is called to write to the /proc file.

```
static ssize_t procfile_write(struct file* file, const char* ubuf, size_t count, loff_t* ppos) {
    printk(KERN_INFO "proc_write\n");
    if (count > BUF_LEN)
        procfs_buf_len = BUF_LEN;
    else
        procfs_buf_len = count;

    if (copy_from_user(msg, ubuf, procfs_buf_len)) {
        printk(KERN_WARNING "Failed to copy data from user space\n");
        return -EFAULT;
    }
    printk(KERN_INFO "got from user: %s\n", msg);
    return procfs_buf_len;
}
```

Set procfs_buf_len as min (user message size, buffer length).

Copy message of at most BUF_LEN characters.

Proc Operations

Make sure that this struct is a global variable! (not inside a function)

```
static const struct proc_ops procfile_fops = {
    .proc_read = procfile_read,
    .proc_write = procfile_write,
};
```

Fill in callbacks to read/write functions. (event-driven)

Initialization

```
static int __init hello_init(void) {
    proc_entry = proc_create(ENTRY_NAME, PERMS, 
PARENT, &procfile_fops);
    if (proc_entry == NULL)
        return -ENOMEM;
    return 0;
}
```

Format:
proc_create(<filename>, <permissions>, <parent>, <pointer to fops>

(you should use macros for the parameters)

Exit

```
static void __exit hello_exit(void) {
    proc_remove(proc_entry);
}
```

Module Initialization and Exit

```
module_init(hello_init);
module_exit(hello_exit);
```

Testing

```
> sudo insmod hello.ko
> echo hi > /proc/hello
> cat /proc/hello
> rmmod hello
```

Memory Copying

Kernel → User

- `unsigned long copy_to_user(void__user* to, const void* from, unsigned long size)`

User → Kernel

- `unsigned long copy_from_user(void* to, const void__user* from,unsigned long size)`

Needed because:

- Isolate user space and kernel space.
- User process uses virtual memory.
- Prevents crashing due to inaccessible regions.
- Can handle architecture specific issues.

File Operations

`.proc_read` – this calls the function binded to it everytime the procfile is read (using cat or something)

`.proc_write` – this calls the function binded to it everytime the procfile is written to (like output redirection).

You will primarily use the .proc_read function, as you will only be reading from the file, never writing to it.With the .proc_read, you can write to the /proc file when needed.

Install Header Files

You will need the kernel header files to compile a kernel module.

You should download the header files

```
> sudo apt-get install build-essential linux-
                    headers-`uname -r`
```

Kernel modules rely on the kernel source code to build modules that are compatible with your kernel version.

Header files provide the API that allows your system to build.

Additionally, your makefile tells the kernel build system to build an external module.

Write a Kernel Makefile

```
obj-m += hello.o
ccflags-y := -g -Wno-error
KDIR := /lib/modules/$(shell uname -r)/build

all:
	$(MAKE) -C $(KDIR) M=$(PWD) modules
clean:
	$(MAKE) -C $(KDIR) M=$(PWD) clean
```

Required object files for this module.

Overwrites cflags that were used for compiling the kernel.

Contains the symbolic link to the source code of the specific kernel version and building rules.

Writing a Kernel Makefile

You can then run the kernel makefile:

```
> make
```

You should get a kernel object file that you can load into your kernel, a .ko (kernel object) file.

Loading a Kernel Module

To load a kernel module, you will need the kernel object file (.ko) that was generated from your makefile.

Run this command:

```
sudo insmod [kernel_object_file]
```

For example, to run the

```
sudo insmod hello.ko
```

Checking Kernel Module is Loaded

To check all loaded kernel modules:

```
> lsmod
```

To check a specific kernel module is loaded:

```
> lsmod | grep [module_name]
> lsmod | grep hello
```

Print Proc File

To print out a proc file, you can cat it out:

```
> cat /proc/[file]
> cat /proc/timer
```

To see it continuously update every [seconds], you can use this command:

```
> watch -n [seconds] cat /proc/[file]
```

Make sure that you have your kernel module loaded.

Unloading a Kernel Module

To unload a kernel module (no .ko extension):

```
> sudo rmmod [module name]
> sudo rmmod [kernel object name]
```

For example:

```
> sudo rmmod hello
```

Kernel Headers

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

- Module stuff

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

- Module stuff

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

- Proc files

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

- User space access for memory

Kernel Headers

`#include <linux/mutex.h>`
- Locks

`#include <linux/list.h>`
- Linked lists

`#include <linux/string.h>`
- String functions

`#include <linux/kthread.h>`
- Kthreads

And many more you can find at: https://elixir.bootlin.com/linux/latest/source

printk

Similar to printf, but only prints to the kernel log.

Takes log level and format string as parameters

```
printk(KERN_INFO “hello %s\n”, str_var);
```

Note that there is no comma between the log level and string argument.

- **For our project, we will not be using printk. We will primarily be using the /proc file mechanism.**
- **Using printk only for debugging your program.**

printk

Kernel log viewable through multiple ways:

```
> cat /var/log/kern.log
> dmesg
> tail -f var/log/kern.log
```

(real-time)

printk

Log levels:

- KERN_EMERG – Emergency condition, kernel likely crashed
- KERN_ALERT – Alert that requires immediate attention
- KERN_CRIT – Critical error message
- KERN_ERR – Error message
- KERN_WARNING – Warning message
- KERN_NOTICE – Normal, but noteworthy message
- KERN_INFO – Informational message
- KERN_DEBUG – Debug message

kmalloc

Dynamically allocate memory in the kernel, similar to malloc in the user-space.

```
void * kmalloc(size_t size, gfp_t flags);
char * char_ptr = kmalloc(sizeof(char) * 20, GFP_KERNEL)
```

Remember to restrict kernel memory allocation.

- Can block important functions.
- Can crash kernel if improperly handled.
- Kernel has limited access to memory.

Ref: https://docs.kernel.org/core-api/mm-api.html#c.kmalloc

kmalloc

Flags:

Below is a brief outline of the most useful GFP flags:

- GFP_KERNEL – Allocate normal kernel ram. May sleep.
- GFP_NOWAIT– Allocation will not sleep.
- GFP_ATOMIC– Allocation will not sleep. May use emergency pools.

Also it is possible to set different flags by OR'ing in one or more of the following additional flags:

- __GFP_ZERO – Zero the allocated memory before returning.
- __GFP_HIGH – This allocation has high priority and may use emergency pools.
- __GFP_NOFAIL – Indicate that this allocation is in no way allowed to fail.
- …

Ref: https://docs.kernel.org/core-api/memory-allocation.html#memory-allocation

Kernel Documentation

- https://docs.kernel.org/
- Demo how to use this web