CIS 657 (POS) fall 2013 Lab 7 –

System call using dynamic kernel linker

# INTRODUCTION

We have created a system call by modifying the kernel. If we modify the kernel, we need to recompile kernel and reboot the machine every time to make syscall work. Now with the help of dynamic kernel linker interface, it is much easier to create, add and use a system call without re-compiling the system every time. We have seen the characteristics common to all KLDs in last lab. For modules that add a syscall, besides the load handler and the DECLARE\_MODULE macro (see 1.5) that must be fulfilled, there are four main parts that are generic (see 1.1-1.4).

## 1.1 Declaring the syscallname\_args structure

For all syscalls, the parameter list seen in the kernel code is as follows:

1. struct thread \*
2. struct syscallname\_args \*

The parameters that one would pass the syscall from user-land are defined in the syscallname\_args structure. For this example, we will have the following syscall arguments:

*struct sc\_example\_args {*

*char \*str; // character string*

*int val; // interger*

*};*

## 1.2 Syscall function

The following is the example syscall, which takes the parameters passed to it (a character string and an integer) and displays them to the currently being used tty (the terminal that is running the program that called the syscall).

*static int sc\_example(struct proc \*p, struct sc\_example\_args \*uap)*

*{*

*char kstr[1024+1]; /\* Holds kernel land copy of uap->str \*/*

*int err = 0; /\* Generic return(err) \*/*

*int size = 0;*

*// Copy the string located at the user land address uap->str to the kernel land address of &kstr.*

*err = copyinstr(uap->str, &kstr, 1024, &size);*

*if (err == EFAULT)*

*return(err);*

*uprintf("The string passed was: %s\n", kstr);*

*uprintf("The value passed was: %d\n", uap->val);*

*return(0);*

*}*

## 1.3 Filling the sysent structure

The next thing we do in our code is fill in a sysent structure for our system call. The sysent structure, defined in /usr/include/sys/sysent.h, is the following:

*struct sysent {*

*int sy\_narg;*

*sy\_call\_t \*sy\_call;*

*};*

There is a sysent structure defined for each system call. 'int sy\_narg' is the variable that defines how many parameters are passed to the system call being defined. In the case of our skeleton code, we have 2 parameters being passed: char \*str and int val. Therefore, we will set sy\_narg to 2. 'sy\_call\_t \*sy\_call' is a function pointer to our static system call. So, in our code we will have the following:

*static struct sysent sc\_example\_sysent= {*

*2, /\* Number of parameters for our system call. \*/*

*sc\_example /\* A function pointer to our new system call. \*/*

*};*

## 1.4 Setting our ‘offset’ variable to NO\_SYSCALL

When we are usually creating a system call and implementing it via something dynamic like a KLD, it is usually not good practice to actually assign a designated slot value. What one \_should\_ do is set the offset value to NO\_SYSCALL. This says: "find next available system calling value."

*static int syscall\_num = NO\_SYSCALL;*

## 1.5 Load Handler Function & SYSCALL\_MODULE macro

We have already completed the necessary parts for implementing a system call, therefore, all we have left to do is write our load handler and call the SYSCALL\_MODULE macro.

*static int sc\_example(struct proc \*p, struct sc\_example\_args \*uap)*

*{*

*char kstr[1024+1];*

*/\* Holds kernel land copy of uap->str \*/*

*int err = 0;*

*/\* Generic return(err) \*/*

*int size = 0;*

*// Copy the string located at the user land address uap->str to the kernel land address of &kstr.*

*err = copyinstr(uap->str, &kstr, 1024, &size);*

*if (err == EFAULT)*

*return(err);*

*uprintf("The string passed was: %s\n", kstr);*

*uprintf("The value passed was: %d\n", uap->val);*

*return(0);*

*}*

Reference: <http://code.google.com/p/ncormier-academic-projects>

Credits: Nicolas Cormier, Oslo, Norway

More about syscalls using KLD: <http://myfreebsd.homeunix.net/freebsd/bsdkern.html>

# Tasks (90)

In this lab, you need to perform the following task in the system call using kernel loadable module:

You also need to attach the user level program which calls newly added system call along with kernel module and makefile.

|  |  |
| --- | --- |
| Requirements | Points |
| Print   * Total Number of processes (5) * Total number of running processes (7.5) * Total number of waiting Processes (7.5)   P.S. Threads in following state can be considered as running  TDS\_CAN\_RUN  TDS\_RUNQ  TDS\_RUNNING | 20 |
| Print total number of page faults since system boot-up | 5 |
| Print memory usage: Active, Inactive, Wired, Cached, Free in Kilobytes (3 for each) | 15 |
| Print Swap usage : Total , Used in kilobytes and Ratio in percentage | 10 |
| Refresh the screen after each second to display latest statistics | 15 |
| Program that compiles correctly | 10 |
| Program that runs without crash | 10 |
| user level program to call newly added syscall | 5 |

# Submission (10)

Create and attach a README (txt/word/pdf) file at the end of the lab. It doesn't need to be comprehensive, but it should at least cover the following content:

* Which tasks are done, and which are not?
* What’s your basic idea to achieve this task?
* Where is your main function?
* Your user level program and what it does?

If you can only finish some of the tasks in this project, please make sure that your code can at least be compiled and installed and also clearly state in the README file about the missing parts of your project.

**Checklist:** To submit your project, you need to:

• Attach the **source code of the kernel module**

• **Makefile**

* **User level program**

• Create and attach a **README** file.

• Send this email to the TAs keeping Dr. Chapin < chapin@syr.edu> in the CC with subject line “CIS657: Lab 7”

Saurabh Sabnis < spsabnis@syr.edu >

Prasoon Pandya < [pdpandya@syr.edu](mailto:pdpandya@syr.edu) >