CSUS, College of Engineering and Computer Science
Department of Computer Science
CSC/CPE 139 – Operating System Principles

Programming Assignment 1 – Linux Kernel Modules

Acknowledgement: This project is authored by Prof. Don Heller and Prof. Bhuvan Urgaonkar.

Description: In this project, you will learn about the Linux system calls that enable a process to be loaded. **fork** and **exec** are the most prominent system calls in creating a process's address space and loading the desired code/data for that process, but process loading consists of several additional steps. In this project, you will document these steps (and learn the associated system calls) and answer specific questions regarding process loading.

The first part of the assignment will focus on compiling and running a small program to gather information about process loading. We will provide you with a UNIX tarball containing the code and the files necessary to build the code. You will be required to build the code on a Linux system to learn the build mechanism. You will run the resultant binary under system call tracing tools to collect the system calls executed. In addition to the traces, you will document process loading at the level of system calls.

In the second part of the project, you will have to answer some questions that will require some more focused investigation into process loading. For example, you will have to examine the binary to identify the causes of certain process loading operations (don't worry, I will provide some guidance below).

Hopefully, this project will enable you to become more comfortable with the Linux system that we will be using and provide you will an initial familiarity with how systems get your programs to run.

Follow these instructions:

1. Download the following tarball to your local account file space. You should have one file **proj1.tgz**.

2. We are going to build the Linux version of the program. **tar** is a rather complex command -- use **man tar** if you want to learn more.

   First, create the project directory proj1: **tar xvfz proj1.tgz**

3. Next, build the binary version of the project program on a Linux system:

   The program binary must be built using **makefile.txt**. For information about compiling via Makefiles look at **man make**.

   Important: You must be on a Linux machine. You can determine the operating system your machine is running via the command **uname -a**.

4. Now, we are going to run the program to generate the system call traces. Linux uses **strace** to trace the system calls used by a process (among other things). From the proj1 directory, run the binary using the following command:

   $ **strace -o strace.linux ./pr1_c89_32**

   This command will generate file **strace.linux** containing a sequence of system calls, including their argument values. Save this trace somewhere safe, as you will need them later.

5. Using **strace.linux**, you need to describe how the process loading is executed. You may use **man** pages to determine what a system call does, but you will probably also need to search for web resources that provide additional guidance. You should be as precise as possible (given some limitations as described below). Perform the following tasks using the **strace.linux** file:
   - Define (in your own words) the function of each unique system call in the trace. There should be about 10 unique system calls in the trace.

     Using **brk** as an example, I would see that the **man** page describes the system call as one that "changes the data segment size", but just returns the end of the current data segment when sent a 0 (actually the **man** page doesn't say that, but you would have looked at some web resources and found that that is true). As a result you could say (but will use your own words), that "brk either returns the current end of the process's data segment or changes (usually by extending) the end of the data segment. The kernel thus backs the data segment with the corresponding physical memory."

   - Write a description of the process loading protocol in terms of the sequence of system calls in **strace.linux**. With the exception of **mmap**, **mprotect**, and **munmap** you should be able to provide a detailed description of what the system call is doing and what object it is operating on. This must answer the question: What is the *purpose* of the operation and the object (file) it is operating on?

     For **mmap** determine whether the system call is mapping data from a file (specify the file), providing a block of zero'd memory, or causing memory to be unmapped.

     munmap and **mprotect** affect specific memory segments in the process's address space. Identify the memory segment by specifying which **mmap** command mapped the memory initially. You can simply identify the **mmap** command by number (don't have to write it out). One **mprotect** appears not to correspond to the **mmap**'d memory.

6. In addition, answer the following question to gain a basic understanding of the structure of a Linux binary.
   - The libraries **libc.so.*** and **libm.so.*** are loaded (via **mmap**) after the executable by a mechanism known as *dynamic linking*. Define dynamic linking (in your own words), including a contrast with static linking.

7. This program is also a good example of a C program, and provides some insights into what an address space looks like.

This program prints out several program variables (data) and functions (code) to help you understand how different variables and code correspond to different areas of an address space. The program prints the following information:

Address (Memory_Size) Variable_Name Value

As Lecture 3 discusses, an address space consists of a code segment (text), global data, dynamic data (heap), and local data (on the stack). How the variables/functions are defined should give you some clue as their segment in the address space. You should then see if the addresses correspond to your understanding. You can then answer the following questions:

Which variables are global? Which variables are local? What is difference in their memory locations?

Allocate a variable using malloc(). Where is this variable located in the address space relative to the others?

Where are local variables allocated? How about function arguments (argv, argc)? What about the environment variables? You should familiarize yourself with environment variables.

Add your own variables to the program and see where they are allocated.