**CA644 System Software Assignment**

Ruben Vasconcelos

Daniel Rejniak

**Introduction**

The assignment is divided into two section. The first section will describe the process that we have went through to add a new system call to the kernel. The second section will describe the changes that were require to turn simple hello world system call to something more interesting.

System Information

OS ……………………………………………………………………………. Ubuntu

Release ……………………………………………………………………… 16.04

Initial Kernel Version ………………………............................................. 4.13.9

Kernel Version After ………………………………………………………... 4.13.11

**Installing And Compiling Kernel (Part 1)**

1.1 Updating OS & Installing Required Packages

1.2 Defining System Call

1.3 Creating The Makefile

1.4 Adding Hello Directory To Main Makefile

1.5 Updating Syscall Table

1.6 Updating system call header file

1.7 Compiling And Installing New Kernel

1.8 Shell Script

1.9 Confirmation

**New System Call (Part 2)**

2.1 System Call Overview

2.2 Create Functionality

2.3 List Functionality

2.4 Start Functionality

2.5 Delete Functionality

2.6 Error Handling

2.7 Logging

2.8 Security

2.9 Screenshots

**1.1 (Updating OS & Installing Required Packages)**

The initial step is to patch the system and ensure that our OS is running the latest available packages. We also add some additional packages that are required using “apt-get install -y”. Runc dependency is also introduced at this point as this will be required by our custom call later on.

|  |
| --- |
| apt-get update && apt-get upgrade -y && apt-get install -y \  wget \  make \  gcc \  libssl-dev \  libncursesw5-dev \  runc |

Now we can download the kernel from kernel.org. We opted for the latest stable kernel. We download and extract the kernel into usr/src directory.

|  |
| --- |
| # Download kernel kernel\_major\_v="4" kernel\_minor\_v="13.11" kernel\_full\_v="$kernel\_major\_v.$kernel\_minor\_v"  rm -rf /usr/src/linux-$kernel\_full\_v linux-$kernel\_full\_v.tar.gz && wget https://www.kernel.org/pub/linux/kernel/v$kernel\_major\_v.x/linux-$kernel\_full\_v.tar.gz && tar -xvf linux-$kernel\_full\_v.tar.gz -C/usr/src/ && rm -rf linux-$kernel\_full\_v.tar.gz |

**1.2 (Defining System Call)**

Once the kernel has been extracted we went into the kernel directory under usr/src and we have create a hello directory. This directory will store our system call along with Makefile. In the hello directory we have created a file called *hello.c* that defines our system call.

|  |
| --- |
| echo #include <linux/kernel.h>  asmlinkage long sys\_hello(void) {  printk("Hello world\n");  return 0; } |

**1.3 (Creating The Makefile)**

Once the template was created we went onto creating the Makefile that will act as a binder during compilation. In the Makefile we added the following content.

|  |
| --- |
| obj-y := hello.o |

**1.4 (Adding Hello Directory To Main Makefile)**

We need to define the location of our new “hello” directory in the main Makefile so the kernel knows where to find it during the compilation process.

|  |
| --- |
| sed -ri 's/core-y.\*kernel.\*/& hello\//' /usr/src/linux-4.4.1/Makefile |

**1.5 (Updating Syscall Table)**

Now we need to create a new entry in the system call table file since we decided to go for the 64bit version of Ubuntu we need to add the new entry in the following file.

“/usr/src/linux-4.4.1/arch/x86/entry/syscalls/syscall\_64.tbl”

Inside the “syscall\_64.tbl” we added the following entry. Note that we have incremented the value of the last call since all calls in the table need to be unique.

|  |
| --- |
| # 333 common hello sys\_hello |

**1.6 (Updating system call header file)**

The last step is to update the syscall.h file located under “include/linux/syscalls.h”

This file contains declarations for system calls. We have added the following declaration at the end of the file :

|  |
| --- |
| asmlinkage long sys\_hello(void) |

**1.7 (Compiling & Installing New Kernel)**

Now that all the necessary steps for adding the call are done we can proceed to execute the commands that will configure , compile and install our new kernel.

|  |
| --- |
| make menuconfig |

This command builds configuration file for our kernel. In our case we went with the default configurations. There are command such as oldconfig to use the existing kernel configurations and allow us to build upon the existing kernel configurations but we have opted to go with new configurations.

|  |
| --- |
| make & make modules\_install install |

This will compile and install all the kernel modules.

We have encountered multiple issue during the installation process. Firstly, we noticed that we require substantially more RAM to complete the kernel installation in this case minimum 4GB and also over 20GB of virtual disk space was required.

Once all of the commands executed successfully we needed to reboot the system for the changes to take effect. However, before doing that we need to check the current version of the kernel. This will be useful in the following steps as a reference to the old kernel version number.

|  |
| --- |
| uname -r : 4.13.9 (Before) reboot now |

**1.8 (Confirmation)**

Once the system has been rebooted we can perform few basic check to ensure our new kernel has been installed. The first check we ran was to see the new kernel version number. We done that by running the following command :

|  |
| --- |
| uname -r : 4.13.11 (After) |

In the previous step we have ran “uname -r” before installation so now we can compare the version. This is a good indication as we are sure that we are running out latest updated kernel.

Lastly we create a simple c program that will make use of our new system call.

|  |
| --- |
| #include <stdio.h> #include <linux/kernel.h> #include <sys/syscall.h> #include <unistd.h>  Int main() {  Long int amm = syscall(333);  printf("Sys call returned %ld\n", amma);  Return 0; } |

To compile the simple test program we ran the following :

|  |
| --- |
| gcc test.c ./a.out Dmesg |

Dmesg command allows us to see what was printed by our system call, the expected result at this point is to see “Hello World” printed in the kernel log. Once we see the message in the log we can confirm that the system call is working as it should.

Note : Once the kernel is initially compiled we only need to run *make* and *make\_modules\_install install* for the changes to take effect. There is no need to go through the entire installation process again. This process takes substantially less time than initial compilation as only the changes component are recompiled.

To speed up the compilation process further we made use of “-j” flag that runs the commands in parallel. This functionality is incorporated into the shell script that we created to automate the process where it automatically detects available cores to speed the work and make the compilation a lot quicker.

**1.9 (Shell Script)**

Updating and configuring new kernel with all the configuration is a very manual process to automate this process we decided to create a shell file that will automate this process. This automated process is very useful when it comes to development since the initial kernel configuration can take a lot of time to configure.

Add more info about the shell file here...

**New System Call**

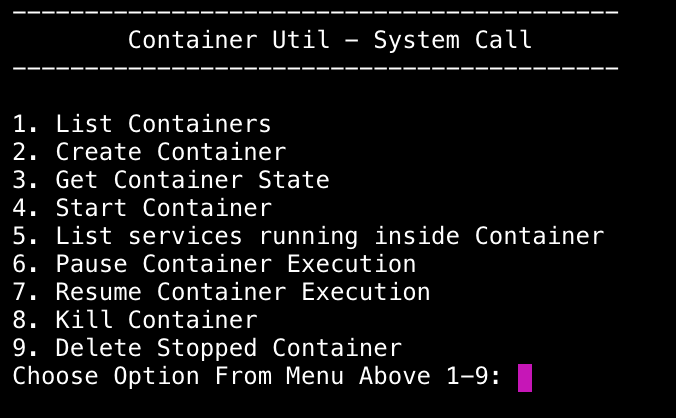
**2.1 (System Call Overview)**

We decided to create a helper utility that will allow developers to create and manage linux container through user friendly interface. Docker does not need to be installed on the operating systems as we leverage linux container that are available on native linux operating systems.

There are two key parts to our design, the first part is a front-end application that provides a menu where users can choose variety of options such as create , list, start, delete and list services inside the container.. User does not need to know the actual commands since the command are encapsulated conveniently under each menu option.

The second part is the actual system call that receives requests from the front-end application, in a secure manner it processes incoming requests. Once the request has been processed by the kernel the system call securely returns the feedback back to the front-end application.

Below is a screenshot of applications main menu:



System call is designed to take two parameters. The first parameter is the option parameter and the second parameter is the container id. Combination of the parameter can trigger required behavior inside the system call.

Below is a constructor of the system call that accepts two parameters.

*sys\_call(int container\_action, char\* container\_id)*

Possible system call requests could be :

*sys\_call(1,0) - This will trigger list container function*

*sys\_call(2,test) - This will trigger the start container function with id “test”*

**2.2 (Create Functionality)**

Createfunctionality allows the user to create a new container, user simply provides the container id and a new container is provisioned. We decided to keep the create functionality on the front-end of the application rather than having it in the kernel due to security concerns. When the user would would be allowed to create a container inside the kernel he could modify the .config and introduce a trojan horse attack by becoming root. We also require the user to be root when creating the container.

**2.3 (List Functionality)**

List functionality allows the user to view all the available container. Additional information such as container id, state, pid bundle and date are also provided. This functionality is very convenient to monitor available container. There are no security concerns around this functionality so any non sudo user is able to use this functionality.

**2.4 (Start Functionality)**

This functionality allows the user to start the container by simply providing the container id that can be obtained from the list container option described in section 2.2. The request is sent to the system call where it's being processed.

**2.5 (Delete Functionality)**

Once the user is finished with the container it can be simply removed by choosing the delete option and providing the container id.

**2.6 (Error Handling)**

When each of the system calls executes its end value is always returned to the front-end application 0 for success and 1 for error. That way we can establish if the outgoing request has been processed successfully. We built a responsive mechanism where any errors are written to the standard error “2>1” and returned to the user as an indication to what could have went wrong in the system call.

**2.7 (Logging)**

Since the calls are made from user land to the kernel as a system admin we want to be able to obtain information about the incoming requests. We can achieve that by using the “current” struct which is made available to the system call. That struct contains information such as the PID of calling process and the actual process name. This information is then logged into the log which can be viewed by execution the following command

|  |
| --- |
| dmesg |

We also log the user inputs and the executional behavior of the system call, this is useful information that helps to determine the interaction of a process with our system call.

**2.8 (Security)**

Several precautions need to be put in place when dealing with input from the user that will reach the system call. Firstly, we need to make sure that the content of the inputted string does not birch any of the security requirements. Several aspects are checked such as:

* Char length
* Null characters
* Tab characters
* Space characters
* Pointer reference
* Invalid characters

Use of fgets in the front end of the application is more secure than use of fscan as fgets takes care of many validation check including buffer overflow check.

Second security check is in the system call itself in an event where some security role could be breached the system call will perform pre check before processing the incoming values.

**2.9 (Screenshots)**