Project 0

Principles of Operating Systems

CMSC 421 — Fall 2023

Project 0

Due by 11:59 PM EDT on Sunday, September 24th

Changelog

August 28, 2023: Initial Version

sept 8, 23: links and file name

Introduction/Objectives:

All of the projects will involve making changes to the Linux kernel and will be conducted within a Virtual Machine (VM) for consistency of results. In order to successfully complete the upcoming projects, it is important to become familiar with compiling the kernel inside the VM and submitting updates.

Project 0 serves as an introductory project, aiming to familiarize you with the necessary procedures. You will be tasked with building a custom version of the Linux kernel, incorporating a modified version string and implementing two additional system calls. Additionally, you will utilize the Git version control system to manage the project's source code.

This project is divided into several parts, each helping you achieve the following objectives:

- Gain familiarity with the tools required for building a custom Linux kernel image.
- Install Linux and build a custom kernel.
- Modify the Linux kernel according to specific requirements.
- Understand the implementation of Linux System Calls.
- Learn how to call system calls from userspace.
- Utilize Git for effective source code revision control.
- Format and submit projects in the designated manner for this course.

Software and Hardware Requirements:

To facilitate the development of projects in this course, we will be using a virtualized environment with the following software and hardware requirements:

1. Virtualization Software:
   - VirtualBox: We will be running the projects within a Virtual Machine (VM) using VirtualBox. This allows us to run a guest operating system inside the host operating system.
   - This is the website to install
   - VirtualBox is available on the machines in the ITE 240 lab running Linux.
2. Hardware Requirements:
   - Relatively modern PC: You should have access to a PC with a 64-bit x86 CPU and a sufficient amount of RAM.
   - CPU Virtualization Extensions: It would be advantageous if your host machine's CPU supports the x86 Virtualization extensions (VT-x for Intel processors or AMD-V for AMD processors). Check your CPU specifications for details.
3. Operating System:
   - Debian 11 (64-bit): For the projects in this course, we will be using the amd64 (64-bit) version of the Debian 11 Linux distribution.

- Custom Kernels: The custom kernels developed in this course will be based on the Linux Kernel version 5.15.91, which is a long-term support release.

For detailed setup instructions based on your operating system, please refer to the following guides that will be in blackboard:

- Windows VirtualBox Setup Guide
- macOS VirtualBox Setup Guide (Intel-based machines)

External Storage Devices and Backups:

External storage devices and backups are crucial for a smooth and secure working experience. Please note the following requirements and recommendations:

1. **External Storage Device:**

   - You must have an external USB hard drive or flash drive if you plan to work in the lab. It is highly recommended to have one even if you do not intend to work in the lab.

   - The external storage device will be used for storing backups of your VM image and should have a capacity of more than 32GiB (gigabytes).

   - Ensure that the device is formatted with a file system that supports files larger than 4GiB.

   - Recommended file systems are NTFS (for Windows compatibility) or a Linux file system such as ext3 or ext4.

   - Note that FAT32 file system is not suitable due to its limitation on file size.

2. **Regular Backups:**

   - Regardless of working in the lab or not, it is highly recommended to regularly back up your VM images.

   - Store your backups on an external hard drive or flash drive, separate from the location where your original data is stored.

   - Since VM images can be large in size, utilizing an external storage device provides ample space and ensures data integrity.

3. **Commit and Push Code:**

   - It is crucial to commit and push your code frequently to a version control system like GitHub.

   - Regularly committing and pushing your code ensures that your project progress is saved and backed up in a remote repository.

   - This practice becomes especially important in case of hardware issues or any unforeseen troubles that may arise.

Installing Linux:

To create the necessary environment for completing the projects in this course, follow these steps, again look up the instructions based on your operating system:

1. **Install VirtualBox 7.0:**

   - If you are using a Linux host OS, install VirtualBox using your package manager.

   👉 Here is the link

2. **Create a Virtual Machine and Install Debian:**

   - Please follow detailed setup instructions based on your operating system. These guides will be in blackboard.

   - During the VM creation process, select the 64-bit version of Debian Linux in the new VM wizard.

   - Adjust the disk size to at least 48 GiB to ensure sufficient space for kernel building, assuming you do not install excessive software within the VM.

   - Allocate an appropriate amount of RAM to the VM, keeping in mind that 4096 MiB is the minimum workable amount. Ideally, do not assign more than half of the total RAM installed on your computer, but a minimum of 8192 MiB is recommended.

   - Consider increasing the number of cores dedicated to the VM, keeping it below the total number of physical cores in your PC (excluding HyperThreaded cores). To do this, select the VM in VirtualBox, go to Settings > System > Processor, and adjust the Processor(s) slider accordingly.

3. **Obtain the Debian Network Install CD Image:**

   - Download the Debian network install CD image from the provided link to ensure you install the correct version of Debian.

   > 👉 Debian Image this link

4. **Debian Installation within the VM:**

   - During the installation, you can choose either the normal or graphical install options.

   - Ensure you select the 64-bit installation (matching the downloaded ISO from the previous step).

   - When prompted for partitioning, choose the guided partition option and create one large partition on the disk (including a small swap partition if desired).

   - While installing packages, select at least one graphical environment from the list if you want a GUI. It is recommended to choose Cinnamon or MATE for a lightweight environment that leaves ample space on the virtual hard drive.

   - IMPORTANT: Make sure to UNSELECT the option to install GNOME.

   - Additionally, select the SSH server option.

   - When prompted to install the boot loader, choose to install it on the VM's hard drive (/dev/sda).

> 👉 **For advanced users only**: It is possible to complete all course assignments without installing a GUI in the VM. However, some setup aspects, such as adding your SSH key to your GitHub account, may require additional steps. If you wish to proceed without a GUI, follow the instructions provided for unselecting the "Debian desktop environment" during setup and omitting UI package installations. Use `make menuconfig` instead of `make xconfig` when building the kernel. Keep in mind that importing the SSH key to your GitHub account will require an alternative method, such as using `scp` or `sftp` to copy it from the VM to an easily accessible location. If you are not comfortable with the Linux terminal, it is recommended to install a GUI environment and disregard this advanced section entirely.

Creating an SSH key:

If you have reached this step, it means you have successfully downloaded VirtualBox and installed the Debian operating system on it.

To work on repositories hosted on GitHub, you need to set up an SSH key on your VM and register it with your GitHub account. Follow these steps to create and associate your SSH key:

1. **Open a terminal in your VM and run the following command (replace "yourumbcemail" with your actual UMBC email address):**

   ```
   ssh-keygen -t rsa -b 4096 -C "yourumbcemail@umbc.edu"
   ```

   -Accept the defaults for most prompts during the key generation process.

   - For enhanced security, provide a passphrase when prompted. The passphrase can be any string of your choice, but make sure not to forget it. You will need the passphrase whenever you use the git command line tool to access resources in your GitHub account.

   - It is also possible to skip using a passphrase, but it is generally recommended for improved security.

   - Do not change the filename or location where the key is stored from the default location.

2. **Associate your SSH key with your GitHub account:**

   - Open a web browser (Iceweasel or Firefox) inside your VM and visit github.com.

- Log in to your GitHub account.

  - Go to the SSH settings section of your GitHub user settings.

  - Use the "New SSH key" option, give the key a title (e.g., "421 VM Key"), and run the following command in a terminal:

```
cat ~/.ssh/id_rsa.pub
```

- Carefully copy and paste the output of the command into the Key box on the GitHub page.

- Click the "Add SSH key" button to save and associate the SSH key with your GitHub account.

> 👉 If you run into issues using your SSH key after adding it to your GitHub account properly, you may need to add it to the SSH agent. Follow the instructions on this GitHub documentation page to do so.

Obtaining Root Privileges:

> 👉 Before you begin the process of building a custom kernel, it's important to understand what root privileges are. In Linux, the root user has full access to the system and can execute any command or perform any action. This level of access is necessary for system administration tasks, but it's important to use root privileges with caution, as it can potentially compromise the security of your system.

To execute commands requiring root privileges in the Linux installation within the VM, you have a few options. Follow these methods to elevate your user privileges:

1. Using su:
   - Open a regular terminal and enter the following command:

   ```
   su -
   ```

   - When prompted, enter your root password.
   - You can now execute any commands that require root privileges.
   - To return to your regular user account, type:

   ```
   exit
   ```

2. Using sudo:
   - Once you have installed the initial set of packages mentioned in the instructions, you can use the sudo program to run commands with elevated privileges.
   - Open a terminal and run a command using sudo by prefixing it with `sudo`:

   ```
   sudo command
   ```

   - When prompted, enter your regular user password.
   - This allows you to perform specific commands with root privileges without switching to the root user entirely.

3. Using sudo with -s flag:
   - Alternatively, you can use the `sudo -s` command to open a root shell.

- Open a terminal and enter the following command:

```
sudo -s
```

- When prompted, enter your regular user password.
- Now, any subsequent commands you run within this root shell will have root privileges.
- To exit the root shell and return to your regular user account, type:

```
exit
```

It is crucial to pay careful attention to which steps require root privileges. In the instructions provided, we explicitly mention the steps that should be run as root. For any other commands, it is important to run them using your normal user account and not the root shell. Failure to follow this guidance may result in issues with your build.

Building a Custom Linux Kernel:

To obtain the Linux kernel sources and build a customized kernel, follow the steps provided in the document carefully. Read all instructions multiple times before starting and make sure all commands run successfully. It is recommended to take notes and underline steps. If something went wrong, do not proceed and go back to find out what happened.

To build a customized Linux kernel, carefully follow these steps:

1. **Update all packages on your installation:**
   - Open a terminal as **root** and run the following commands:

   ```
   apt-get update
   apt-get upgrade
   apt-get dist-upgrade
   apt-get install sudo
   usermod --append --groups sudo YourVMUsername
   reboot
   ```

   - Wait for the VM to reboot. After rebooting, you will be able to use `sudo` for running commands as root. Remember to prefix each command with `sudo` if you are running multiple commands at once as root.
2. **Install required software packages for kernel development:**
   - Open a terminal as **root** and install the required packages with the following command:

   ```
   apt-get install build-essential valgrind nano patch diffutils curl fakeroot git pkg-config gedit libssl-dev libncurses5-dev libe
   lf-dev bison flex bc dkms rsync clang-format dwarves qtbase5-dev qtchooser qt5-qmake qtbase5-dev-tools
   ```

3. **Obtain the Linux Kernel sources and unpack them:**
   - Open a terminal as **root** and run the following commands:

   ```
   cd /usr/src
   chmod 777 .
   ```

   - Switch to your **normal user** account and run the following commands in the terminal:

   ```
   cd /usr/src
   git config --global user.email yourumbcemail@umbc.edu
   git config --global user.name "Your Full Name"
   git clone git@github.com:UMBC-CMSC421/linux.git fall23-project0
   cd fall23-project0
   git remote remove origin
   ```

```
git remote add origin git@github.com:UMBC-CMSC421/fall23-project0-yourusername.git
git push -u origin main
```

- The sources in the `/usr/src/fall23-project0` directory will be referred to as the **working copy** of the kernel for the rest of the project.

> 👉 If the git push -u origin main command returns an error message about the repository does not exist or you do not have permission to access it, then make sure that the repository has been created (that is to say that you clicked on the link on the project assignment on Blackboard and that your repository has been created), that you have correctly created and added your SSH key to GitHub, and that you are not trying to run the command as root. Also ensure that you properly run the two git remote commands inside the */usr/src/fall23-project0*

4. **Build and install your customized kernel:**
   - Read the "Compiling a Kernel" chapter of the Debian Administrator's Handbook for additional insights.
   - Review "Linux Kernel in a Nutshell" for further understanding.
   - To ensure that our customized kernel does not interfere with the modules or other files of any kernels installed by the Debian package manager. Provide a unique version string for your kernel in the format `5.15.67-fall23-project0-USERNAME`. USERNAME is your UMBC username (your UMBC email username — NOT your Github username, if they differ). Edit the **Makefile** in the `/usr/src/fall23-project0` directory and modify the line starting with `EXTRAVERSION`. Anything after the equal sign on that line (excluding leading whitespace) will be appended to the kernel's version number.

   ```
   EXTRAVERSION = hello
   ```

   - The above change would result in your kernel's version string to 5.15.67hello.

> ✋ Before adding **any functionality**, I would suggest that you compile the kernel and boot it. This will allow you to know that the original kernel works. I added this step to help you narrow down any errors that may occur. Before moving on

   - Implement a "**Hello World**" system call to the kernel, performing the system call implementation and kernel modification steps in the tutorial found as a separate file in this directory. It is called “CMSC421 Project 0 Hello World” You should be sure to run the test program and verify that the system call was able to run successfully **only after you build, install, and reboot into your new kernel**. You will not be able to test the kernel until you reboot your VM after building it.
   - Implement a **memory copy system** call to the kernel, following the guide found in this directory called “CMSC421 Project 0 Adding a memcpy system call”. As with the syscall added in the previous step, you will only be able to test this system call after building your kernel, installing it, and rebooting into the new kernel.
   - Configure and compile the custom kernel and its modules. This step will likely take a while:

   ```
   cd /usr/src/Fall23-Project0
   make mrproper
   make xconfig
   make bindeb-pkg
   ```

     - During the `make xconfig` step, configure the "CRC32c CRC algorithm" or "CRC32c (Castagnoli, et al.) Cyclic Redundancy-Check" option to be built-in to the kernel (checkmark, not a dot). Save the configuration file with its default filename.

- If you have allocated multiple cores to your VM, you can use the `j` flag with `make bindeb-pkg` to utilize multiple cores (e.g., `make -j2 bindeb-pkg` if you allocated

Once the kernel has been built, you need to install the kernel image. Follow these steps:

5. Configure and compile the custom kernel and its modules. This step will likely take quite a while. Run the following commands in a terminal

   (make the change mentioned below during the make xconfig step and be

   sure to save the configuration file to its default filename):

   - Open a terminal as **root** and navigate to the `/usr/src` directory.

     1. Run the following command, substituting your UMBC username in both places:

        ```
        dpkg -i linux-image-5.15.67-fall23-project0-USERNAME+_5.15.67-fall23-project0-USERNAME+-1_amd64.deb
        ```

        If you encounter any issues with this command, ensure that you are in the correct directory, have root privileges, have successfully built the kernel without any compiler errors, and have correctly changed the version string. If you have built the kernel multiple times, the `+-1` at the end of the filename may increment. Make sure to install the newest kernel image with the highest number.

6. Your working copy of the kernel should now be built and installed. The installation process should have updated your bootloader configuration and added the new kernel entry, making it the default option.
7. Reboot your VM and select your customized kernel from the GRUB boot menu.
8. After the reboot, open a terminal and run the following command to check if your custom kernel boots properly:

```
uname -a
```

9. Verify that the output from the `uname` command matches the correct version string as specified above.
10. If the output from the `uname` command does not match the version string format provided, you can try rebuilding the kernel by running `make bindeb-pkg`. This command will only rebuild the kernel without executing any other make steps (such as `make xconfig`). After rebuilding, install the new kernel using `dpkg` as root. You should not need to run `make mrproper` again unless instructed otherwise by an instructor or TA.
11. To verify that the Hello World system call works correctly from userspace, follow the steps provided in the "Testing Your New System Call" section of the Hello World system call tutorial. If the system call does not work properly, ensure that you have followed all the steps in the tutorial and the compilation steps of this assignment. Double-check that you have not made any mistakes in creating the test program. One error could be adding a space when typing things out.

Submission Instructions:

Follow these steps to submit your project:

Verify the changes in the local git repository:

- Open a terminal in the `/usr/src/fall23-project0` directory.
- Run `git status` and ensure that any modified files are listed under "Changes not staged for commit".
- Run `git diff` to confirm that all your changes to the kernel source code are accurately displayed.

Add and commit your changes:

- Use the following commands to add and commit your modifications (replace `FilesYouModifiedOrAdded` with the relevant filenames):

```
git status
git add FilesYouModifiedOrAdded
git commit #Give a *good* commit message to describe your changes
```

Verify the commit in the local repository:

- Run `git log` and confirm that your commit appears at the top of the list.
- Double check by login into your Github account on the website and check your repository with the changes are there.

Push the changes to your GitHub repository:

✋ Execute the command `git push origin main` to push the changes to your GitHub account ONLINE

- Verify on the GitHub website that your changes have been successfully pushed to the repository.

Fill out the form with your GitHub username:

✋ Complete the form to provide your GitHub username to the TAs. Ensure that you are signed into your UMBC account when filling out the form. **LINK HERE**

Congratulations! You have completed Project 0 Setting Up!

Part 2 : Hello World System Call

☐ **Part 2: Adding a System Call to the Linux Kernel**

Introduction

Later in the semester, there will be an additional project that will add several system calls to add interesting functionality to the Linux kernel. In preparation for this task, this guide has been developed to show you how to go about adding a simple system call to the kernel. This guide will focus on the 64-bit x86 architecture, however similar steps could be taken on other CPU architectures that are supported by the Linux kernel.

The system call added to the kernel in this guide is trivial, but it demonstrates the process in a relatively simple manner. All the system call developed in this guide does is print a simple "Hello World!" string to the kernel's log. The second part will add a memcpy system call. Don't worry, we give you the code.

System Call Implementation

To start out, make a new directory in the root of your kernel source tree. For the purposes of this example, let's just call the directory **project0(/usr/src/fall23-project0)**. Create a new file in that directory called hello.c. **(/usr/src/fall23-project0/hello.c)**.

In the hello.c file created above, let's implement the system call itself. Copy/paste the following code into the source file...

```
#include <linux/kernel.h> #include <linux/syscalls.h>

SYSCALL_DEFINE0(hello) {
printk("Hello World!\n"); return 0;
}
```

As mentioned before, this system call is trivial. All it does is print the string "Hello World!" to the kernel's message log, not to the screen. We do not have a direct printf to use within kernel space. While it would be possible to use the write system call to print to the calling process' stdout, I'll leave that as an exercise to the reader.

There are a couple of things to notice about this system call code. First, you should note that it returns 0 in all cases (as a long). This notifies the calling process that the system call hopefully succeeded no errors occurred. Any positive return value from a system call is treated as a successful return. If the system call were to indicate an error occurred, a negative number should be returned — specifically a negated value from the <errno.h> header file should be returned, indicating the type of error. The system C library will automatically take that negative value and put the appropriate value in the errno variable in user-space (and return -1 from the function). Another point to notice is that the system call is defined in a slightly strange looking way. While it is possible to define the code for system calls with more standard-looking declarations, due to various changes added to the kernel source code for working around the Meltdown and Spectre vulnerabilities, it is better to do things this way now. This type of declaration helps to ensure that the system call can be used on other architectures than just x86-64, where system calls may be annotated in some special way. The SYSCALL_DEFINE0 part is a macro that expands out to make the declaration of the function. It will define the function to return a long, which is used to return any errors the function may encounter, and it will also be declared to have 0 arguments. There are similar macros like SYSCALL_DEFINE1 that annotate that the function will take one argument (and so on for other numbers of arguments too).

After creating the system call, we must create a Makefile in the syscall's directory to build the code when the kernel is built. Since the kernel has a relatively nice build system,
the Makefile is extremely short and simple. Copy and paste the following into a new file named Makefile in the project0 directory:

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

With that, we have completed all "new" code that needs to be added to the kernel.

Kernel Modifications for the New System Call

To ensure that our new system call code gets compiled, we must first add in the

new project0 directory (/usr/src/fall23-project0) to the normal kernel build system. To do this, open the root level Makefile under /usr/src/fall23-project0. Look for the following section of the Makefile (which should start at line 1150 on a clean copy of the 5.15.67 kernel source code):

```
ifeg ($(KBUILD_EXTMOD),)

core-y   += kernel/ certs/ mm/ fs/ ipc/ security/ crypto/ block/

vmlinux-dirs  := $(patsubst %/,%,$(filter %/, $(init-y) $(init-m) \
              $(core-y) $(core-m) $(drivers-y) $(drivers-m) \
              $(net-y) $(net-m) $(libs-y) $(libs-m) $(virt-y)))
```

You must modify the core-y line to add our new directory. Modify it to look like this:

```
core-y   += kernel/ certs/ mm/ fs/ ipc/ security/ crypto/ block/ fall23-project0/
```

Next, open the *include/linux/syscalls.h* file. We must modify this file to provide the prototype of the system call we've added. This would be needed, for instance, if we were to need to call the system call from other kernel code or system calls. It isn't strictly necessary for this system call, but it is still a good idea. Go to the bottom of the file and add the following right before the #endif:

```
asmlinkage long sys_hello(void);
```

Finally, we must add the new system call to the system call table. Open
the arch/x86/entry/syscalls/syscall_64.tbl file, go to the end, and add the following after the comment that appears at the end of the file.

```
548      common   hello       sys_hello
```

In this file, the first column represents the system call number. The second column indicates which ABI the system call is a part of. For x86-64, your options are "common", "64", and "x32". Most of the time, you want to use the "common" option to make it so the system call will work regardless of which mode is currently in use. The third column is the descriptive name of the system call. The last column is the name of the actual function you've created. Every column except the second one (the ABI column) should have unique values for every system call.

You have now finished modifying the kernel. All that's left now is to build and install your new kernel, reboot into it, and test it out. I'll leave the building, installing, and rebooting steps as an exercise to the reader here.

Testing Your New System Call

To test out your new system call, you must build a user-space program to call it. Create a new file (in your home directory) and copy/paste the following content into it:

```
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <errno.h>
#include <linux/kernel.h>
#include <sys/syscall.h>

#define  NR_hello 548 long hello_syscall(void) {

  return syscall( NR_hello);

}

int main(int argc, char *argv[]) {

  long rv;

  rv = hello_syscall(); if(rv < 0) {

  perror("Hello syscall failed");

  }

  else {

  printf("Hello syscall ran successfully, check dmesg output\n");

  }

  return 0;

  }
```

Note that if you had a different number for your system call in the previous step (i.e, you're not using a clean copy of the 5.15.67 kernel source), then you may need to adjust

the __NR_hello line. The only interesting part of this code is the part where the system call is called. This is done by making use of the syscall macro, provided by the C library. This function-like macro accepts an integer argument to note what system call to perform, along with any number of arguments to pass to the system call. So, if the hello system call accepted an argument (with a name, for instance), you would simply add that argument to the list passed there, after __NR_hello.

Compile the program with your system C compiler (gcc) and run it. The program will print out the success message if the syscall runs correctly, and you should see the "Hello World!" message if you look at the output running the dmesg program as root. If you get an error message from the program (which should usually say "Function not implemented"), make sure that you have made all the changes in the guide properly, built the modified kernel, installed it, and rebooted into it before running the test program.

If you've gotten this far successfully, congratulations, you've written your first Linux kernel system call!

Further exploration

The system call developed in this tutorial is rather boring and doesn't really do anything of major interest. There are ways this could be improved. For instance, you could use

the write system call within the kernel to write the greeting string to the user-space program's standard output. Another, more interesting modification would be to modify the system call to accept a string to greet the user by name. This would complicate the system call quite a bit, as you would have to handle proper copying of the argument into kernel- space, however it would make a very worthwhile exercise for anyone who wants to develop a non-trivial system call.

Part 3 Memory Copy System Call

☐ Part 3: **Adding a Memory Copy System Call**

Adding a Memory Copy System Call

The memcpy() function

In C, a commonly used function from the system's C library is the memcpy() function. This function takes in two pointers and a size and copies the given number of bytes from one pointer to the other of the pointers. A very simple (albeit slow, performance-wise) version of this function might be implemented in the following manner:

```
void *memcpy(void *dest, const void *src, size_t n)
{
  unsigned char *d = (unsigned char *)dest;
  const unsigned char *s = (unsigned char *)src;
  size_t i;
  unsigned char tmp;

  for(i = 0; i < n; ++i)
  {
    tmp = s[i];
    d[i] = tmp;
  }

  return dest;
}
```

This function is implemented in the system's C library (or more likely by the compiler's runtime) on most systems in user-space. Also, there is a version of this function implemented in the Linux kernel for use in kernel code (which only is designed to copy from a kernel-space pointer to another kernel-space pointer).

In this part of Project 0, you will be implementing a system call to do the same job as the userspace version of memcpy(). This part of the assignment is designed to introduce you to the concept of kernel and user memory spaces and how to copy data back and forth between them. This will be **extremely** useful knowledge for future projects in this class.

Kernel and User Memory Spaces

In the Linux kernel (and most other operating system kernels as well), there is a very clear distinction between kernel memory and user memory. While there is no physical hardware difference between the two (both are resident in the RAM of your system), they are treated very differently by software. Also, each process' own user memory space must be kept separate from other processes. Largely, this is a security and protection feature. For an idea of why this is important, think about it this way. There are several different processes running in the same memory and at the same time on a modern system. Assume one of these processes is interacting with a hardware encryption engine and processing data using secret keys. It is important in such situations to protect the memory containing secret keys and prevent other processes from gaining access to the keys. The kernel must also keep its memory separate for similar reasons (and to ensure no programs can circumvent the protections outlined above).

Most functions within the Linux kernel's minimal C library are designed solely for operating on data in the kernel's memory. The memcpy() function in the kernel is only designed to copy from one kernel pointer to another — not to copy to or from the memory of a userspace process. There are, however, several special purpose functions in the kernel's C library that are designed to work with userspace pointers, however usually only one at a time. Below is a sampling of such functions, including comments explaining how they work (note that these declarations are somewhat simplified from the actual kernel code).

```
/* Do a very quick check whether a userspace memory access of the
given size is likely to succeed or not. Returns non-zero if the access
appears to be valid. This is not a guarantee that the access will succeed
-- you should always check the return values of the functions you use to
access the user memory as you use them. */

int access_ok(const void  user *ptr, unsigned long size);

/* Copy a simple value from the userspace pointer ptr to x.
Used as follows, assuming ptr is a userspace pointer:

unsigned char value; if(get_user(value, ptr)) {
handle_error_condition();
}

The variable value will then contain the data that was at
the userspace pointer ptr.
*/

#define get_user(x, ptr) /* ... */

/* Copy a simple value from x to the userspace pointer ptr. Used in a similar manner to get_user above. */

#define put_user(x, ptr) /* ... */

/* Copy a block of data (of size n bytes) from the userspace pointer from to the kernel-space pointer to, returning the number of bytes
should always be 0 on success! (You *must* check the return value or you will get a compiler warning) */

unsigned long copy_from_user(void *to, const void  user *from, unsigned long n);

/* Copy a block of data of size n bytes from the kernel-space pointer from to the userspace pointer to, returning the number of bytes *

unsigned long copy_to_user(void  user *to, const void *from, unsigned long n);
```

There are a few other useful operations that are available as well regarding copying/checking strings, but they are left out of this discussion for now.

Any system calls that accesses userspace memory **must** ensure that any such accesses are valid and use these memory access functions **correctly** to ensure the safety of the operations they perform. This means checking all userspace pointers (those tagged

with _user) before accessing them with the access_ok, and ensuring that any accesses with *get_user, put_user, copy_to_user, or copy_from_user* succeed, and acting accordingly if they fail (for instance, by returning an appropriate error code, such as -*EFAULT*.

System Call Implementation

Your task in this part of the assignment is to take the version of memcpy from earlier in this document and modify it to be a system call. You will add that system call to your kernel, build, install, reboot into said kernel, and test that your function works as required. Start by adding a new file to your project0 directory (*/usr/src/fall23-project0*) within your working copy of the kernel called copy.c, and adding copy.o to the obj-y line in the Makefile you should have already created in that directory. Then go through the steps to add the new system call to the system call table (as number 549) and to the *syscalls.h* file as well.

Your system call must be declared in the following manner in copy.c:

```
SYSCALL_DEFINE3(memory_copy, unsigned char  user *,
                to, unsigned char  user *, from, int, size)
```

You should use the *access_ok* function in the kernel to check both pointers before copying any data. You can then use the *get_user* and *put_user* macros to read each byte from userspace and write each byte back, respectively. You will need to include the <*linux/kernel.h*>, <*linux/uaccess.h*>, <*linux/syscalls.h*>,
and <*linux/errno.h*> header files at the top of copy.c to use any of these functions. If any checks fail (from *access_ok, get_user,*

or *put_user*), you should stop trying to copy data and immediately return *-EFAULT*. If all data is copied correctly, return 0 at the end of the function (rather than the destination pointer as the userspace memcpy does). Recall that in C, that a pointer to the ith element at a pointer *ptr* (or the *ith* element of an array, equivalently) can be obtained in the following two ways (use whichever you would like in your *get_user* and *put_user* calls:

```
&ptr[i]

/* OR */

ptr + i
```

Testing Your System Call

As with the hello world system call earlier, you should test your new memory copy system call to ensure that it works properly. You should test your function with a variety of pointers, both good and bad (for instance, try copying to or from a NULL pointer to see what happens. You should notice that when you copy to or from a bad pointer that the system call always returns *-1*. If you look at the *errno* variable in your program (*#include <errno.h>* in your test program to get access to it), you will find that it should be set to the constant *EFAULT* (also in errno.h) when an error occurs.

Cleaning up:

After submitting your project, it is recommended to clean up your VM to avoid running out of space. Run the following command in a terminal:

```
cd /usr/src/Fall23-Project0
make mrproper
```

To avoid losing points on this project, please ensure the following:

- Submit your changes to your GitHub repository before the project due date.
- Avoid making excessive unnecessary changes to the kernel sources.
- Not submitting a read me doc with your implementation.
- Include all the required changes specified in the project instructions.
- Properly implement the required changes without errors.
- Provide a descriptive and meaningful commit message.
- Exclude any extraneous files from the submission.
- Fill out the form with your GitHub username to inform the TAs.
- Please make sure to comply with these requirements to avoid any point deductions.