

Kernel Compilation, Kernel Module, and System Call Implementation

1 Introduction

The objective of this assignment is to familiarize yourself with the Linux kernel source code. Specifically, you will:

- Compile your own Linux kernel.
- Create your own kernel module.
- Add a new kernel system call and then test this system call from a user-space program.

All the assignment's steps must be performed using your **Debian virtual machine** from programming assignment 1. The notions from the course involved in this assignment are the following:

- Linux source code exploration and compilation.
- Installing and running a modified kernel, as well as creating and testing a module.
- The **printk** kernel function.
- User space / kernel space communication using a system call.

2 Finding and Booting Your “Known Good” Kernel

Start the Debian VM, log in, open a terminal window, and execute the following command to find the exact version of the kernel you are currently running: **uname -r**

Write down the result somewhere and remember this exact kernel version for the rest of the semester. This is your “known good” kernel which is known to work correctly. Also execute **ls -l /boot** and you should see one “vmlinuz” file in the /boot directory that has the exact same version in its name. This is the file containing your “known good” kernel. Never delete or modify this file in any way.

Use Debian's battery icon menu (upper right corner of the Debian desktop) to restart the virtual machine. **As soon as** you see the white-on-blue-green menu of the GRUB boot loader, press the down arrow key on the keyboard to select the second menu entry named “Advanced options for Debian GNU/Linux”. You have only 5 seconds to do this! Then press **Enter** to get a sub-menu. In the sub-menu, select the “known good” kernel, based on the version you learned just above (but not the “recovery mode” one, which is similar to Microsoft Windows's “safe mode”, which is not what we want here) and press **Enter** to boot this “known good” kernel. From now on this is what you will do **every time** you boot or reboot the VM, to make sure you always know exactly which kernel you are booting! If later you have a problem when testing your own modified “working” kernel (see below), such as a kernel that panics (crashes) on boot, you will always be able to boot this “known good” kernel instead to get back to work.

3 Compiling Your Own “Working” Kernel

- 1) Before compiling your own “working” kernel, you must first install some tools and the Linux kernel source code. So, after booting the “known good” kernel (as indicated above), log in, open a terminal window, and execute the following command to install all the software required to configure and

compile the Linux kernel source code:

sudo apt install build-essential pahole libelf-dev libncurses-dev libssl-dev flex bison ([build-essential](#) is a software package that depends on all the development libraries and header files and basic development tools that you need, so installing it guarantees that all those things are present on your VM; [pahole](#) and [libelf-dev](#) are a set of tools and a library to handle the [ELF standard file format for executable files](#); [libncurses-dev](#) is a library to create [text-based user interfaces](#), which is used by “make menuconfig” below; [libssl-dev](#) is a library that provides [cryptographic functions and secure networking](#); and [flex](#) and [bison](#) are a [lexer / scanner generator](#) and [parser generator](#), respectively).

- 2) Install the Linux kernel source code: **sudo apt install linux-source**

This automatically selects the right version of the Linux kernel source code, based on the version of the “known good” kernel which is currently executing on the VM. Then execute **ls -l /usr/src** and you should see one file named “linux-source-X.Y.tar.xz” (“[tar](#)” is the standard file archive format for Unix, and “[xz](#)” is one among many compressed file formats on Unix; other common ones are “[gz](#)” and “[bz2](#)”; together “tar” and “xz” give you a compressed file archive, similar in spirit to the “zip” compressed file archive format which is used a lot on Microsoft Windows).

- 3) Execute **cd** to make sure that you are in your home directory. Then decompress and extract the kernel source code you just installed: **tar -xavf /usr/src/linux-source-X.Y.tar.xz** (the **-x** option means “extract”; **-a** means auto-compress or auto-decompress, as the case may be; **-v** means “verbose”, to see the names of the extracted files as the extraction happens; **-f** is followed by the name of the compressed archive file). After doing this, you should have in your home directory a new directory named “linux-source-X.Y” (use **ls** to see it).

Use **du -s -h linux-source-X.Y** to see how big the Linux kernel source code is.

- 4) Change the name of this new “linux-source-X.Y” directory: **mv linux-source-X.Y pa2**
- 5) Execute **cd pa2** to move into the top directory of your Linux kernel source code. Use **ls** to have a look at the different directories there (see slides 20-21 in the “Introduction to Linux” lecture notes on Canvas for an explanation of the content of those directories).
- 6) Next we need to create a configuration file for the kernel (see slides 34 and 36 of the “Introduction to Linux” lecture notes). The simplest method is to just copy and then modify the configuration file for the “known good” kernel which is currently executing on your VM. Again, the command **uname -r** tells you exactly which kernel version you are currently executing on the VM, so we can use the result of this command to copy the right kernel configuration file from the /boot directory into the current directory containing your Linux kernel source code: **cp /boot/config-\$(uname -r) .config**
- 7) We can now modify this kernel configuration file using: **make menuconfig**

You should then get a text-based user interface (see slide 35 in “Introduction to Linux”) which allows you to easily modify your kernel configuration file:

- a) Press **Enter** to go into the “General setup” sub-menu, press the down arrow twice on your keyboard to go down two lines, then **Enter** to select “Local version - append to kernel release”. In the new text window that appears, enter a minus sign **-** followed by your login name (which is also your Stevens login name), followed by “-pa2”. For example: **-pmeunier-pa2** (replace my login name with yours, obviously). Do **not** add any space before, inside, or after this text. This piece of text is going to be added to the version of the Linux kernel you are going to compile, so that later you can easily recognize which kernel is yours. This will be your “working” kernel. Press **Enter** to select “OK” and you should be back in the previous sub-menu, with the text you just

typed is now showing between () parentheses in front of the name of the “Local version - append to kernel release” sub-menu. Press the right arrow on your keyboard to select “Exit” and then **Enter** to go back to the previous top-level menu.

- b) Select Save and press **Enter**. In the new window, check that the default “.config” file name is correct and press **Enter** to save your new configuration file. Then select “Exit” to quit the “menuconfig” user interface and go back to the usual Unix shell.
- c) Double check the differences between the original kernel configuration file and the one you just modified: **diff /boot/config-\$(uname -r) .config**

In the output you should see that the kernel variable CONFIG_LOCALVERSION is now defined to be the piece of text “-pmeunier-pa2” (except with your own login name) while before it was the empty string (ignore any change for CONFIG_CC_VERSION_TEXT and for SALT, SIG, and KEY stuff, these changes are irrelevant). You are now ready to compile the Linux kernel source code for your “working” kernel!

- 8) Execute **nproc** to see how many processors your VM is using (this should be the same number you indicated in the settings of the VM in VirtualBox). The more the better.

You might also want to go into the settings of your host operating system (Microsoft Windows or Apple macOS) and make sure that the power settings of your computer are set for maximum performance (for example, in Microsoft Windows, click on the battery icon at the right end of the bottom taskbar, then move the slider to “Best performance”); you can undo this setting later again after you are finished compiling the kernel.

Also make sure that your host OS is not set up to automatically suspend or shut down your host computer if you do not move the mouse or whatnot for a while. Again you can undo this later when you are finished compiling the kernel.

If you are using a laptop computer, also make sure that it is plugged into an electric socket, otherwise your host OS might automatically throttle your computer’s performance to save the battery.

Finally, click with your right mouse button on the Debian desktop background, select “Settings”, then “Power”, then click on “Automatic suspend” and turn off both “On Battery Power” and “Plugged In”; this will guarantee that Debian does not suspend itself while you are in the middle of compiling. Then close the settings window.

Now we are ready to compile the whole Linux kernel source code using all the VM’s processors in parallel, to speed up things: **make -j \$(nproc) all**


Wait an hour or two (or three). You will know the compilation is over when your computer’s fan stops making more noise than usual... If for some reason you need to stop the compilation before it is finished, just press Ctrl-c on the keyboard. You can then re-start the compilation later using the same command again, and it will automatically restart from where it left off.

If for some reason the compilation fails, it might be hard to see why it failed because you are compiling different parts of the kernel code in parallel. In that case, you can execute just **make all** to restart the compilation from where it failed but this time in sequential mode on a single CPU, which will make it much easier to see what the problem is. Then contact your nearest Course Assistant.

If you look at the output of the **make** command while it is compiling the code, you will see different abbreviations: CC is when compiling a C file to get a “something.o” object file (a file containing binary CPU instructions), LD is when linking (gluing) multiple object files together, AR is when creating a library (a “something.a” archive file of object files, which later will be linked with the rest of the

kernel), [M] means that this code is part of a dynamically loadable kernel module, not part of the monolithic Linux kernel proper.

- 9) Execute **du -s -h .** to see how much disk space the current directory containing the whole compiled Linux source code takes (it should be around 19GB).
- 10) When compiling the kernel, all the kernel modules are also compiled at the same time. We now need to install these new kernel modules: **sudo make INSTALL_MOD_STRIP=1 modules_install**
This must be done as root (the system administrator) hence the use of “sudo” here. The **INSTALL_MOD_STRIP=1** argument given to the **make** command tells **make** to strip extra debugging information from the modules when installing them, which saves around 2GB of disk space.
Once the modules are installed, you can execute **ls -l /usr/lib/modules** and you should see in the output a directory with a name that has your login name and “pa2” at the end, which is where the modules were installed. The command **du -s -h /usr/lib/modules/*** should show you that all the different directories there (the one for your own modules, as well as the one for the “known good” kernel) are about 400MB in size.
- 11) Now we can install the new “working” kernel itself: **sudo make install**
Do not worry about any “Please install the Linux kernel “header” files matching the current kernel” message in the command’s output; we would need this only if we were to try to use our Debian VM as the host for another VM on top of it!
Execute **ls -l /boot** and you will see that **make** installed several files there, all of which have your login name and “pa2” at the end of their name: “config-X.Y...-pmeunier-pa2” which is a copy of the “.config” file you used for compiling your kernel (you can check this using the **diff** command, for example; see above), “initrd.img-X.Y...-pmeunier-pa2” which is the kernel’s “initial RAM disk” which the kernel only uses at boot time (see [here](#) for more information if you are curious), “System.map-X.Y...-pmeunier-pa2” which contains a list of the kernel’s symbols (the function names and variable names inside the kernel) for your kernel (see slide 44 in “Introduction to Linux” lecture notes), and finally “vmlinuz-X.Y...-pmeunier-pa2” which is your very own “working” kernel!
As part of the kernel’s installation, the configuration file for the GRUB bootloader is also automatically updated so that the bootloader now knows about your new kernel.
- 12) Use Debian’s battery icon menu to restart the virtual machine. Again, **as soon as** you see the white-on-blue-green menu of the GRUB boot loader, press the down arrow key on the keyboard to select the second menu entry named “Advanced options for Debian GNU/Linux”. You have only 5 seconds to do this! Then press **Enter** to get a sub-menu. In the sub-menu, select now your own “working” kernel, which has your login name and “pa2” at the end of its name (but again not the “recovery mode” one) and press **Enter** to boot your own kernel. Happiness ensues.
- 13) After booting your own “working” kernel, log in, open a terminal window, use Ctrl++ to increase the font size of the terminal window, and execute **uname -a** (which should show a kernel name with your own login name and “pa2” at the end of it) and then **id**. Take a screenshot and save it as a picture somewhere on your host computer, you will need to submit it later on Canvas (more details about this at the end of this document). Make sure the kernel version and your login name (Stevens login name) are clearly visible inside the terminal window in your screenshot. For example:

A terminal window titled 'Terminal' with a search bar and window controls. The prompt is 'pmeunier@debian: ~'. The user has entered 'uname -a' and 'id'. The output of 'uname -a' is 'Linux debian 6.1.69-pmeunier-pa2 #1 SMP PREEMPT_DYNAMIC Wed Jan 24 21:06:04 EST 2024 x86_64 GNU/Linux'. The output of 'id' is 'uid=1000(pmeunier) gid=1000(pmeunier) groups=1000(pmeunier),27(sudo),100(users),995(vboxsf)'.

```
pmeunier@debian:~$ uname -a
Linux debian 6.1.69-pmeunier-pa2 #1 SMP PREEMPT_DYNAMIC Wed Jan 24 21:06:04 EST 2024 x86_64 GNU/Linux
pmeunier@debian:~$ id
uid=1000(pmeunier) gid=1000(pmeunier) groups=1000(pmeunier),27(sudo),100(users),995(vboxsf)
pmeunier@debian:~$
```

Congratulations on compiling and booting your first Linux kernel!

Now create in your home directory another new directory that you will use for submitting your assignment (again, use your own Stevens login name): **cd; mkdir pmeunier-pa2**

Then copy your “working” kernel configuration file into your submission directory:

cp pa2/.config pmeunier-pa2

To finish this part of the assignment, create a backup copy of the kernel source code you just compiled, just in case (**-r** means to do a recursive copy that will automatically copy the directory, its sub-directories, its sub-sub-directories, etc.; **-p** preserves the file time stamps when copying which the **make** command uses to decide whether to compile a file again or not, so this is important when copying compiled code that has been created using **make**; this command takes a couple of minutes to complete because you are copying about 19GB of data): **cp -p -r pa2 pa2-backup**

Later, when doing the rest of this assignment, if you make a big mistake inside your “pa2” directory, such as accidentally deleting some files, you can always find the original files inside this backup directory. In the worst case, if you do not know how to fix the mistake, you can always completely delete your “pa2” directory (**cd; rm -rf pa2**) and re-create it using the backup copy (**cp -p -r pa2-backup pa2**).

4 Creating a Kernel Module

In your submission directory (**cd pmeunier-pa2**), create a new directory (**mkdir module**). In that new directory (**cd module**), write a Linux kernel module in a file named “pmeunier.c” (use your own login name). This module must have a **printk()** statement that outputs “Hello World from NAME (LOGIN)” in the kernel log when the module is loaded into the kernel, where NAME is your full legal name and LOGIN is your Stevens login name. When the kernel module is unloaded it must print “PID is XYZ and program name is NAME” where XYZ is the PID number and NAME the program name of the current process. (See your “Introduction to Linux” and “Processes in Linux” lecture notes.)

Compile your kernel module using the appropriate “Makefile”. (Make sure that you are still running your own “working” kernel when doing this!)

You must provide a single screenshot that shows two Debian terminal windows side by side: one window must show the kernel log (with your module's output visible in it), and the other window must show the output of the **uname -a** command followed by commands to load-unload your module at least twice (so we can see in the kernel log that the PID printed when unloading the module changes during each unload). Save the screenshot somewhere on your host computer, you will need to submit it later on Canvas (more details about this at the end of this document). For example (partly censored):

```

pmeunier@debian: ~/pmeunier-pa2/module
pmeunier@debian:~/pmeunier-pa2/module$ uname -a
Linux debian 6.1.69-pmeunier-pa2 #1 SMP PREEMPT_DYNAMIC Wed Jan 24 21:06:04 EST 2024 x86_64 GNU/Linux
pmeunier@debian:~/pmeunier-pa2/module$ sudo [REDACTED]
[sudo] password for pmeunier:
pmeunier@debian:~/pmeunier-pa2/module$ sudo [REDACTED]
pmeunier@debian:~/pmeunier-pa2/module$ sudo [REDACTED]
pmeunier@debian:~/pmeunier-pa2/module$ sudo [REDACTED]
pmeunier@debian:~/pmeunier-pa2/module$

EMPT_DYNAMIC Wed Jan 24 21:06:04 EST 2024
4,589,6648334,-;23:21:05.332676 main Executable: /opt/VBoxG
uestAdditions-7.0.14/sbin/VBoxService\x0a23:21:05.332677 main
Process ID: 999\x0a23:21:05.332677 main Package type: LI
NIX_64BITS_GENERIC
4,590,6649916,-;23:21:05.334240 main 7.0.14 r161095 started
Verbose level = 0
4,591,6651685,-;23:21:05.335997 main vbglR3GuestCtrlDetectP
eekGetCancelSupport: Supported (#1)
4,592,6664402,-;vboxsf: g_fHostFeatures=0x8000000f g_fSfFeature
s=0x1 g_uSfLastFunction=29
5,593,6664452,-;vboxsf: Successfully loaded version 7.0.14 r161
095
4,594,6664485,-;vboxsf: Successfully loaded version 7.0.14 r161
095 on 6.1.69-pmeunier-pa2 (LINUX_VERSION_CODE=0x60145)
4,595,6665463,-;23:21:05.349791 automount vbsvcAutomounterMount
It: Successfully mounted 'shared' on '/mnt/shared'
7,596,9151583,-;rfkill: input handler disabled

1,597,109334160,-;Hello World from Philippe Meunier (pmeunier)
1,598,117435048,-;PID is 2655 and program name is [REDACTED]
1,599,120603090,-;Hello World from Philippe Meunier (pmeunier)
1,600,123059297,-;PID is 2667 and program name is [REDACTED]

```

5 Adding a System Call to Your “Working” Kernel and Testing It

5.1 Adding The System Call

In the directory containing the compiled source code of your “working” kernel (**cd ~/pa2**; “~” is a shell special character that always represents your home directory) create a new directory (**mkdir my_syscall**). In that new directory (**cd my_syscall**), in a file named “my_syscall.c”, write the C code for a new system call named “LOGIN_syscall” (replace LOGIN with your Stevens login name) that takes as single parameter a pointer to a character array containing a string. Make sure you use the proper C macro to define your system call (see the “System Calls” lecture notes). The code of your system call must always return a signed long integer (long) as result.

Your system call must do the following:

- 1) If the string pointer given as argument is NULL then your system call must immediately return -1.
- 2) If the string length is larger than 32 (where the string length is its total number of characters, including the ‘\0’ string terminator character), then your system call must immediately return -1.
- 3) Copy the string from user space to kernel space.
- 4) Use **printk** to print “before: “ followed by the string. Note: whenever you use **printk**, always make sure that the string you want to print is terminated with a ‘\n’ newline character, otherwise the string will not immediately appear in the kernel logs when your code is executed.

- 5) Replace all occurrences of a lowercase vowel letter (a, e, i, o, u, y) in the string with the first letter of your login name, in uppercase (for example, given my “pmeunier” login name, the letters a, e, i, o, u, and y are all replaced with the uppercase letter P).
- 6) Use **printk** to print “after: ” followed by the modified string.
- 7) Copy the modified string from kernel space to user space.
- 8) The system call then returns as result the number of character replacements performed in step 5.

In your code, use the functions listed on slide 25 of the “System Calls” lecture notes as much as possible. Make sure your code **properly checks for errors when using each of these functions** (see the example system call code on slide 21 in those same lecture notes). Do not use a module to implement your system call. Also create a “Makefile” in the same directory (see slide 27).

Modify other files in the Linux kernel source code as necessary (see slides 26 and 27; use **gettimeofday** from slide 21 as an example of how to add your own system call to the various Linux files). Give your system call the number 451. Keep a list somewhere of all the files you modify, because later you will need to copy them to your submission directory.

Recompile your “working” kernel, the same way you did it in section 3 of this assignment. Note: some of the files you need to modify are referenced throughout the kernel’s code, so, when recompiling, the **make** command then needs to recompile most of the kernel code again, which again takes a very long time. So make sure you modify the files **correctly on the first try**, so you do not have to recompile the whole kernel over and over. If you only need to change your `my_syscall/my_syscall.c` file then recompiling the kernel should take only about 5 minutes since the rest of the kernel’s code will not need to be recompiled again in this case.

If the compilation succeeds then reinstall your “working” kernel and its associated modules (do NOT try to install anything if the compilation failed!) Note that, when installing your modified “working” kernel, the previous version of your “working” kernel will be renamed with an extra “.old” extension at the end of its name. You can just ignore this one.

Reboot the virtual machine and make sure that you use the GRUB bootloader to select the correct kernel when rebooting (your “working” kernel with the “-pmeunier-pa2” name, that now contains the code for your own system call).

If your “working” kernel panics (crashes) on boot for some reason, reboot with your “known good” kernel and fix the problem in the code of your “working” kernel.

After you have booted the correct kernel, execute the following command to double-check that the kernel you are now running actually knows about your system call (see slide 44 of the “Introduction to Linux” lecture notes; replace “pmeunier” with your own login name): **grep pmeunier /proc/kallsyms**
If the output of this command is empty then the current kernel does not know about your system call and so you did something wrong somewhere. Re-read this whole section and try again.

5.2 Testing The System Call

In your submission directory (**cd ~/pmeunier-pa2**), write the C code for a user-space test program called “syscall.c” that invokes your system call. Since your system call does not have a corresponding C wrapper function available in the C standard library, you must directly use the “syscall” function of the C standard library to call your system call (see slide 18 of the “OS Concepts and Structure” lecture notes for

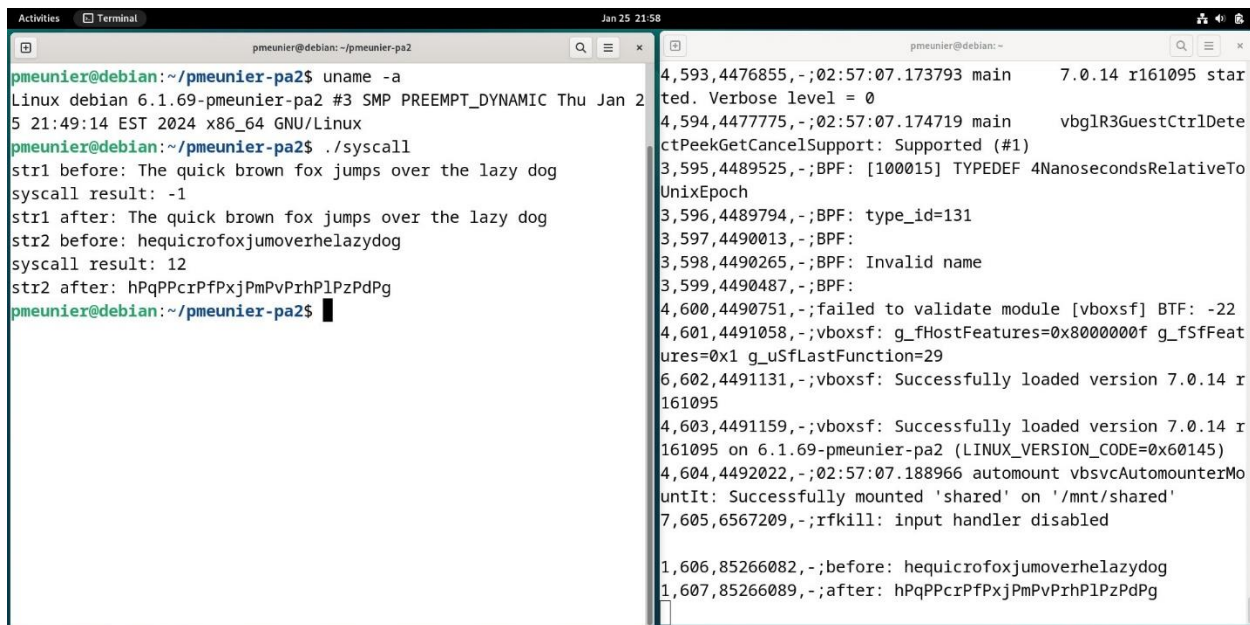
an example). Your code must invoke your system call twice: once for the case where the string size is larger than 32, and once for the case where the string size is less than 32. In both cases, your C program must print on the screen:

- 1) The string given to the syscall, before the system call happens.
- 2) The return value of the syscall.
- 3) The string after the system call, even if it has not been modified.

Note: you can directly use the system call number 451 in your C code.

Important note: the strings that you use in your code must be defined as local variables (the same way it is done for example on slide 18 of the “OS Concepts and Structure” lecture notes), not as string constants that you directly give to the system call as argument. This is because, when your code executes inside a process, all the string constants from your code are stored in a part of the “data” segment of the process’s address space which is read-only (since string constants are... constant), which in turn means that your kernel system call would fail when trying to modify those strings. Local variables are stored in the process’s stack, which is read-write and your system call will then be able to modify those strings.

You must provide a single screenshot that shows two Debian terminal windows side by side: one window must show the kernel log (with your system call’s output visible in it, when the string given as argument to the system call is short enough), and the other window must show the output of the **uname -a** command followed by the execution of your user-space test program. Save the screenshot somewhere on your host computer, you will need to submit it later on Canvas (more details about this below). For example:



```
pmeunier@debian: ~/pmeunier-pa2
pmeunier@debian:~/pmeunier-pa2$ uname -a
Linux debian 6.1.69-pmeunier-pa2 #3 SMP PREEMPT_DYNAMIC Thu Jan 2
5 21:49:14 EST 2024 x86_64 GNU/Linux
pmeunier@debian:~/pmeunier-pa2$ ./syscall
str1 before: The quick brown fox jumps over the lazy dog
syscall result: -1
str1 after: The quick brown fox jumps over the lazy dog
str2 before: hequicrofoxjumoverhelazydog
syscall result: 12
str2 after: hPqPPcrPfPxjPmPvPrhPlPzPdPg
pmeunier@debian:~/pmeunier-pa2$
```

```
pmeunier@debian: ~
4,593,4476855,-;02:57:07.173793 main 7.0.14 r161095 star
ted. Verbose level = 0
4,594,4477775,-;02:57:07.174719 main vbglR3GuestCtrlDete
ctPeekGetCancelSupport: Supported (#1)
3,595,4489525,-;BPF: [100015] TYPEDEF 4NanosecondsRelativeTo
UnixEpoch
3,596,4489794,-;BPF: type_id=131
3,597,4490013,-;BPF:
3,598,4490265,-;BPF: Invalid name
3,599,4490487,-;BPF:
4,600,4490751,-;failed to validate module [vboxsf] BTF: -22
4,601,4491058,-;vboxsf: g_fHostFeatures=0x8000000f g_fSfFeat
ures=0x1 g_uSfLastFunction=29
6,602,4491131,-;vboxsf: Successfully loaded version 7.0.14 r
161095
4,603,4491159,-;vboxsf: Successfully loaded version 7.0.14 r
161095 on 6.1.69-pmeunier-pa2 (LINUX_VERSION_CODE=0x60145)
4,604,4492022,-;02:57:07.188966 automount vbsvcAutomounterMo
untIt: Successfully mounted 'shared' on '/mnt/shared'
7,605,6567209,-;rfkill: input handler disabled

1,606,85266082,-;before: hequicrofoxjumoverhelazydog
1,607,85266089,-;after: hPqPPcrPfPxjPmPvPrhPlPzPdPg
```


6 What to Submit For This Assignment

Once both your system call and your user-space test program work, copy the code of the system call, plus any other kernel file you modified, into your submission directory (your user-space test program must already be there): `cd ~/pa2; cp -p -r my_syscall ... any other file you modified ... ~/pmeunier-pa2`

In the same submission directory, create a PDF file named “screenshots.pdf” that contains:

- 1) Your full name.
- 2) The Stevens Honor pledge.
- 3) The three screenshots you created above: one showing that you compiled your own “working” kernel (section 3), one showing how your kernel module works (section 4), and one showing how your system call and your user-space test program work (section 5.2). Make sure the screenshots are clearly readable.
- 4) Also add a short explanation before each screenshot so the Course Assistants know what you are trying to show on those screenshots.

At this point the submission directory (“pmeunier-pa2”) must contain all the files you have created or modified during this assignment (including the kernel .config configuration file, which you can only see in the submission directory by using the `ls -a` command). Install the zip program on your virtual machine (`sudo apt install zip`) and then create a ZIP file of your submission directory (the `-r` option means to zip all the subdirectories recursively: do not forget it; also use your own student login name, twice, obviously): `cd; zip -r pmeunier-pa2.zip pmeunier-pa2`

Once you have correctly created the file “pmeunier-pa2.zip”, copy it to the host OS using your shared folder, **double-check its content to make sure it contains everything**, then submit it on Canvas.

After the deadline for this assignment has passed, you can delete the backup copy of your “working” kernel: `rm -rf ~/pa2-backup`

It is up to you whether you delete the compiled source code of your “working” kernel or not (`rm -rf ~/pa2`), you will not need it anymore. You can delete it after the deadline for this assignment has passed, if you want to save VM disk space, or delete it only at the end of the semester, or keep it for ever as a souvenir!

Done!

7 Rubric

1) Correct .config file:	5%
2) Screenshot showing the corresponding “working” kernel version:	5%
3) Code of module/pmeunier.c:	15%
4) Code of module/Makefile:	5%
5) Screenshot of kernel log with module loading-unloading (twice):	10%
6) Code of my_syscall/my_syscall.c:	25%
7) Code of my_syscall/Makefile:	5%
8) Code of syscall.c:	15%
9) Other modified kernel files:	5%
10) Screenshot of kernel log and output of syscall.c:	10%

Important note: you do not get points for screenshots unless the corresponding files are submitted too. So for example you will not get points for the first screenshot listed above if you do not also provide the corresponding .config file. Screenshots alone will not get you any points at all. So make sure you double check everything before you submit on Canvas!