# Custom system calls and wrappers at kernel level

Working on ubuntu 22.04.5 virtual machine

Downloaded kernel from

<https://cdn.kernel.org/pub/linux/kernel/v6.x/linux-6.0.7.tar.xz>

(Other versions with respective URL)

How to compile and install

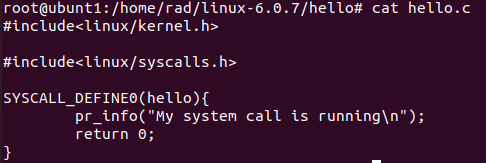
1. Set the configurations using make menuconfig or copy previous kernel configs to .config file. (cp -v /boot/config-$(uname -r) .config) in linux-version folder
2. Use make command to compile. Wait several hours. (Used make -jn where n is number of processors to do faster)
3. Use make modules\_install and make install to add the kernel to grub.
4. Can boot with new kernel using grub. It will be visible in advanced options in grub.

Once booted check kernel version with uname -r command in terminal.

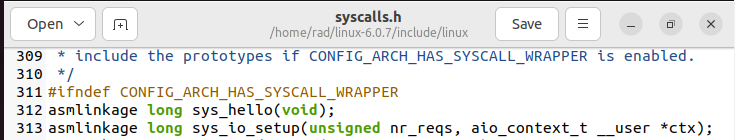
Notes:

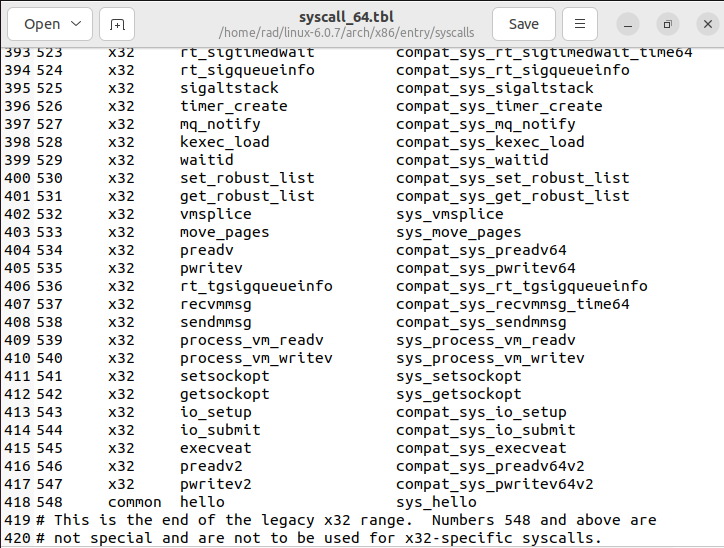
Compilation of 4.17.4 failed due to some error. Possibly due to reverse an incompatibility error in gcc.

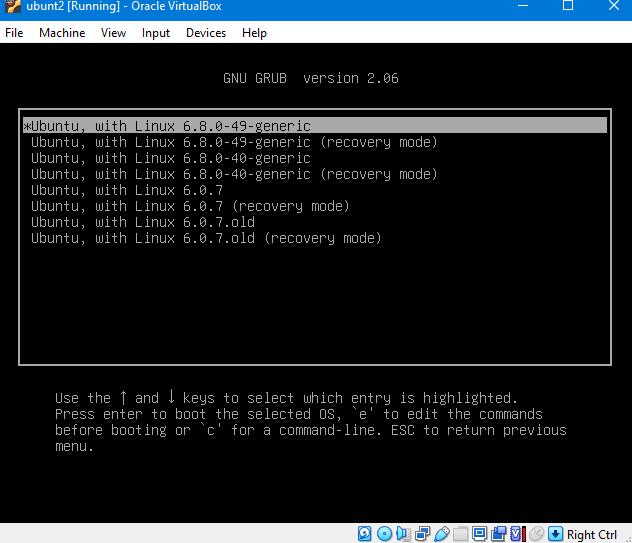
## Custom System call created as demonstration

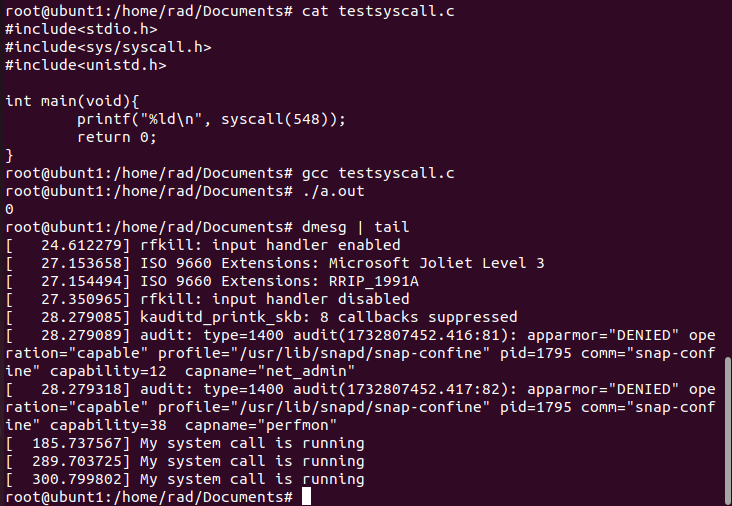
* System call code:
* System call Makefile:



* Updated syscalls.h with system call declaration:
* Added system call at end of system call table with new system call number

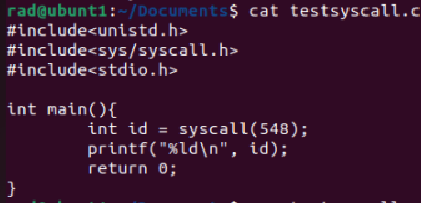


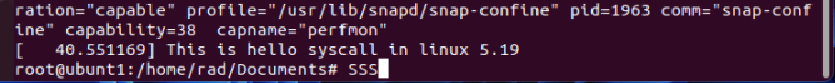
* Compile kernel as done previously and update grub. Reboot system and use new kernel. (Check advanced options) Use 6.0.7 which is the latest version.
* Do note that you must press shift before original kernel boots to get grub menu



* Test the system call by writing and compiling a C code to call the syscall by number.
* Custom System call running linux 6.0.7 kernel ubuntu distribution: (check system logs to see if it works). Also returns 0 which means the system call is running.

**Hello syscall implemented in linux 5.19**

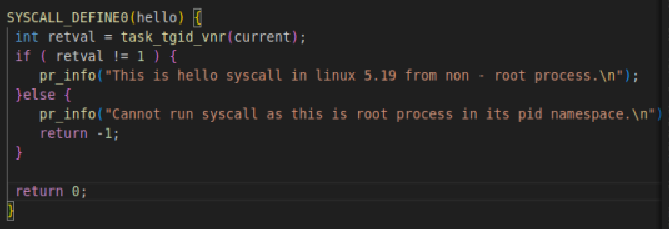




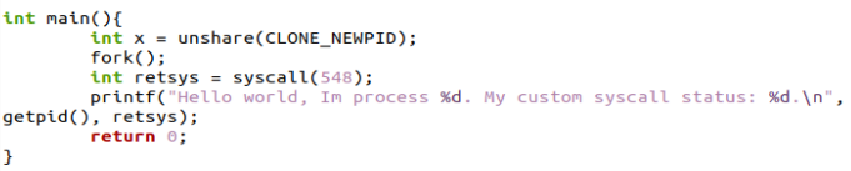
## Disable custom system call in a particular process

* In this extension we look to disable the hello system call in the root process of a pid namespace. (i.e the process who pid is 1). The goal is that this should be done purely in the kernel context WITHOUT any userspace code.
* This involved finding the implementation of getpid to figure out the kernel function to call to get the process pid. It should be noted that system calls are not usually called directly in the kernel.

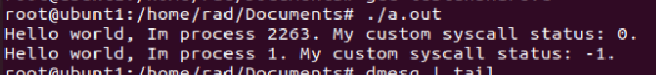
Modified system call code:



Test program:



Stdout:

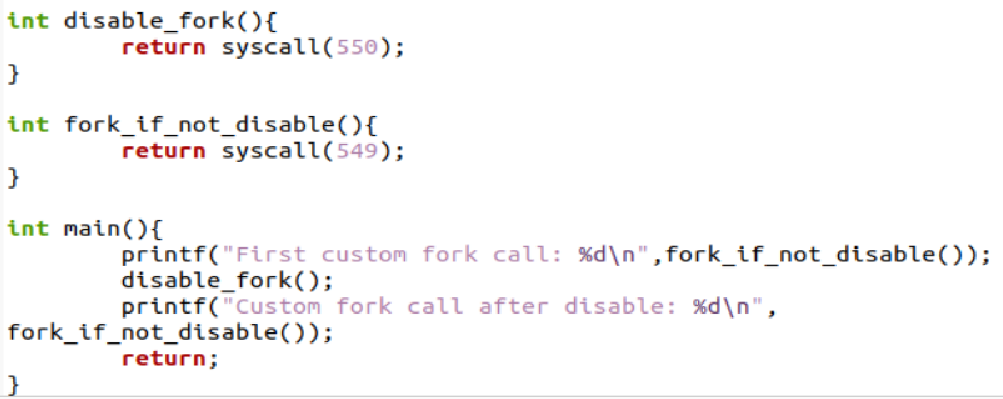
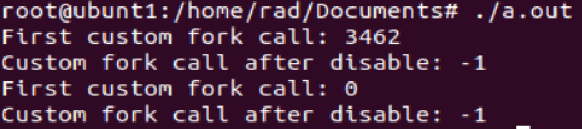
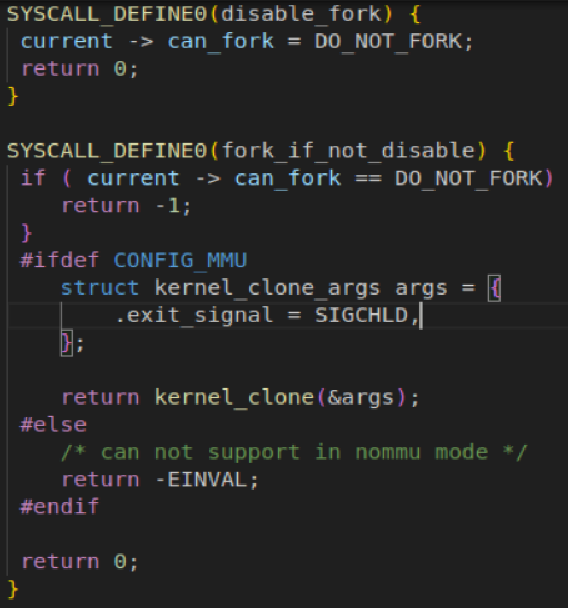


Kernel logs:



## Resource Management (Disable fork in process and further children)

* *Two* system calls created to enable this feature
* Main points:
  + No user space data needs to be stored in any file, as all meta-data regarding disabling is stored in kernel data structures (Specifically the task\_struct of the processes)
* syscall(549): fork\_if\_not\_disable()
  + The fork\_if\_not\_disable is a system call wrapper for the original fork implemented as a system call within the kernel.
* syscall(550): disable\_fork()
  + Updates kernel data structure for the process to disable further calls to fork\_if\_not\_disable() in the same process or any of its descendants
* Future Work:
  + We can simply copy the system call code for fork\_if\_not\_disable to the original fork so that the user experience does not change. Currently we have not implemented it as such as it is only a prototype.
* Use: This system call can be used to safely disable forks to prevent memory / resource overflow in use cases when users predicts fork to be called in incorrect places.



## Close All Files System Call

* Created a close all files system call
* It should be noted that linux has a system call that closes files in a certain range
* This implementation was used as reference to create the close all files system call

Main Issues Encountered:

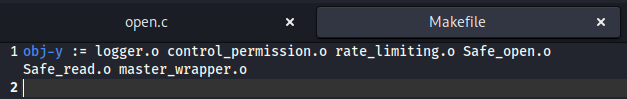
1. Compilation Issues which vary based on kernel version
2. Difference in system call implementation based on kernel version.
   1. In higher versions SYSCALL\_DEFINEn(some fields) can be used, whereas in lower versions it is different.
   2. Changes to Makefile differs based on kernel version. In higher versions may have to change Kbuild file whereas in other versions Makefile change is sufficient.
3. Compilation time is extremely long, generally exceeding 2 hours (Highly dependent on system specifications). Any small change in a file may lead to cascading dependencies that may cause the whole kernel to recompile.

References

* Common errors and required libraries specified in below references to compile smoothly

1. <https://phoenixnap.com/kb/build-linux-kernel> (Compiling kernel 6)
2. <https://medium.com/anubhav-shrimal/adding-a-hello-world-system-call-to-linux-kernel-dad32875872> (Kernel 4.17.4)
3. <https://medium.com/@aryan20/create-custom-system-call-on-linux-6-8-126edef6caaf#:~:text=Create%20the%20new%20syscall&text=%22SYSCALL_DEFINEn()%20macro%20rather%20than,for%20the%20parameters%20as%20arguments.%E2%80%9D&text=This%20is%20a%20prototype%20for%20the%20syscall%20function%20we%20created%20earlier>. (kernel 6)

Attempt at implementing a wrapper library

* Tried to implement the previously made file i/o wrappers inside kernel code.
* Firstly we can’t call some system calls directly as we had done in user space due to security reasons.
* So we used kernel APIs like flip\_open/flip\_close(for opening/closing a file), vfs\_read/write, set\_fs , printk, kmalloc etc. for the same functionality as the user space syscalls.
* Follow the same steps as above to create a new folder, put all the .c files here and add a Makefile like shown below.hi tk
* Add the syscalls into the syscall table and the syscalls.h file.
* Repeat all the steps and recompile the kernel.
* There were some issues with compilation due to some calls not being defined properly and unexpected behaviour.
* Some .c files did compile but I was not able to test it, as my kernel compiled but wasn’t able to run OS onto it.
* Got the error: driver frame buffer was unable to register with bus\_type coreboot ; bus was not initialized
* Searched google but wasn’t able to fix the error.