Operating Systems – 234123

Homework Exercise 2 – Wet

Due Date: 21/12/2025 23:59

Teaching assistant in charge: Alex Zhybirov

Important: the Q&A for the exercise will take place at a public forum Piazza only. Critical updates about the HW will be published in pinned notes in the piazza forum. These notes are mandatory and it is your responsibility to be updated. A number of guidelines to use the forum:

- Read previous Q&A carefully before asking the question; repeated questions will probably go without answers.
- Be polite, remember that course staff does this as a service for the students.
- You're not allowed to post any kind of solution and/or source code in the forum as a hint for other students; In case you feel that you have to discuss such a matter, please come to the reception hour.
- When posting questions regarding this assignment, put them in the "hw2-wet" folder.

Pay attention that in the dry part there are questions about the wet part, so it is recommended to review these questions before, and answer them gradually while working on the wet part.

Only the TA in charge of the assignment can authorize postponements. In case you need a postponement, please fill out the attached form:
https://forms.gle/22VoKLdEaDysNBkM9

1. Introduction

As we learned in class, serving user requests is an important role of the OS kernel. The kernel services are exposed to user applications through system calls. For example, C programs can create new processes and finish them via the `fork()` and `exit()` system calls, respectively.
The goal of this assignment is implementing a new set of services for managing a new feature called process syscall banning. To this end, you will implement four new system calls.

But before diving in, we start with a detailed walk-through of Linux kernel development. You should work on the Ubuntu virtual machine that was installed in HW0.

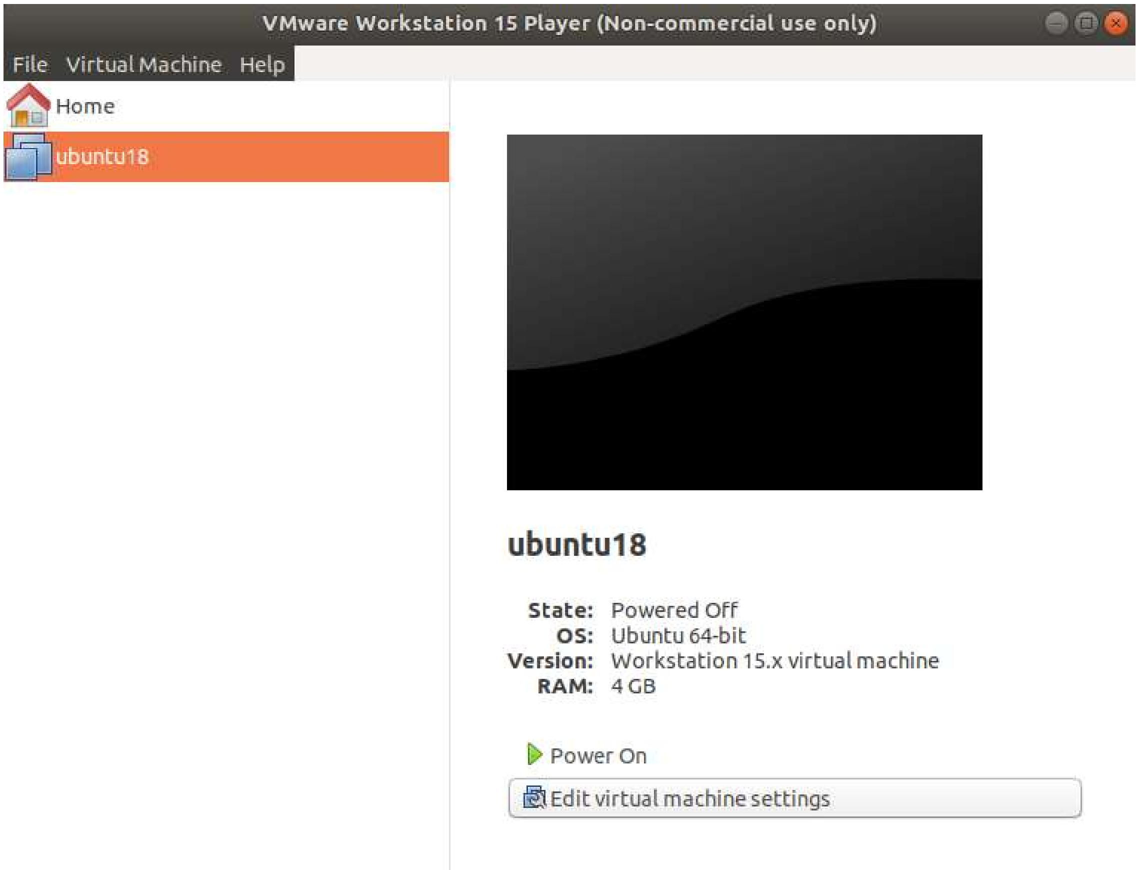
2. Minimum System Requirements

For this assignment, you will need at least:

- Processing: Intel/AMD 64-bit CPU.
- Memory: 4GB RAM.
- Storage: 50GB HDD or SSD.

If your PC has multiple cores, we recommend increasing the number of cores allocated to the VM to reduce the kernel build time (as explained in the next section):

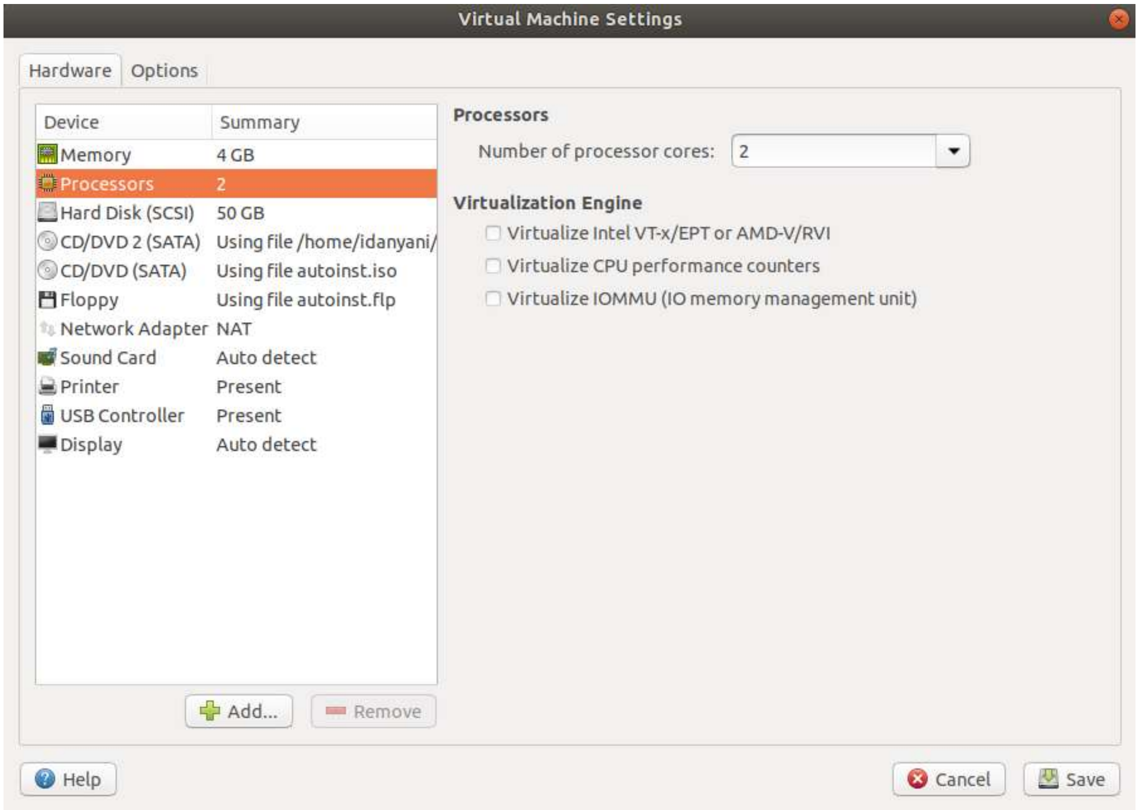
1) Click “Edit virtual machine settings” on VMware main panel.


VMware Workstation 15 Player (Non-commercial use only)
File
Virtual Machine
Help
Home
ubuntu18
Power On
Edit virtual machine settings

ubuntu18

State: Powered Off
OS: Ubuntu 64-bit
Version: Workstation 15.x virtual machine
RAM: 4 GB

2) Navigate to the “Processors” tab and set the “number of cores” to $N-1$, where N is the number of cores on your PC (we keep a free processor core for the host).



Device	Summary
Memory	4 GB
Processors	2
Hard Disk (SCSI)	50 GB
CD/DVD 2 (SATA)	Using file /home/idanyani/
CD/DVD (SATA)	Using file autoinst.iso
Floppy	Using file autoinst.flp
Network Adapter	NAT
Sound Card	Auto detect
Printer	Present
USB Controller	Present
Display	Auto detect

Processors

Number of processor cores: 2

Virtualization Engine

☐ Virtualize Intel VT-x/EPT or AMD-V/RVI
☐ Virtualize CPU performance counters
☐ Virtualize IOMMU (IO memory management unit)

3) Save the settings and power on the VM.

3. Build the Linux Kernel

Estimated time: around 30 minutes on a VM with 2 cores.
Please follow the following instructions on your terminal:

1) Install prerequisites (software required for the next steps):

```
>> sudo sed -i "s/# deb-src/deb-src/g" /etc/apt/sources.list
>> sudo apt update -y
>> sudo apt install -y build-essential libncurses-dev bison flex
>> sudo apt install -y libssl-dev libelf-dev
```

APT (Advanced Package Tool) is a tool that simplifies software management on Ubuntu systems by automating the retrieval, configuration, installation and removal of software packages.

2) Download the Linux source code:

```
>> cd ~
>> apt source linux
```

3) Change permissions and rename the directory:

```
>> sudo chown -R student:student ~/linux-4.15.0/
>> mv ~/linux-4.15.0 ~/linux-4.15.18-custom
```

4) Configure the kernel build process (starting from an existing configuration file):

```
>> cd ~/linux-4.15.18-custom
>> cp -f /boot/config-$(uname -r) .config
>> geany .config # you may use any text editor you like
>> # search the CONFIG_LOCALVERSION parameter and set it to "-custom"
>> yes '' | make localmodconfig
>> yes '' | make oldconfig
```

The “make localmodconfig” command enables only the modules loaded in the currently running kernel. This step reduces the build time significantly (by ~80%). Note that we use the “yes” command with single quotes (two ‘ characters) that encloses nothing. Look at the man page of “yes” to see what it is doing.

5) Compile the kernel and its modules:

```
>> make -j $(nproc)
```

The “-j” flag specifies the number of “make” jobs to run simultaneously. In our case, we use all available cores.

6) Install the kernel modules:

```
>> sudo make modules_install
```

This command will copy the module objects under `/lib/modules/4.15.18-custom`.

7) Install the kernel image:

```
>> sudo make install
```

This command will copy the kernel bzImage from `arch/x86/boot/bzImage` to `/boot/vmlinuz-4.15.18-custom` and create a matching initramfs under `/boot`.

You can verify the installation by listing the installed files with:

```
>> ls /boot/*-custom
/boot/config-4.15.18-custom      /boot/System.map-4.15.18-custom
/boot/initrd.img-4.15.18-custom  /boot/vmlinuz-4.15.18-custom
```

8) Configure GRUB to boot from our custom kernel:

```
>> sudo geany /etc/default/grub
>> # search GRUB_DEFAULT and set it to: "Ubuntu, with Linux 4.15.18-
custom"
>> # search GRUB_TIMEOUT_STYLE and set it to menu
>> # search GRUB_TIMEOUT and set it to 5
>> # add the line GRUB_DISABLE_SUBMENU=y at the end of the file
```

As we learned in class, GRUB is the default boot loader for our Ubuntu VM (and for most Linux operating systems). The GRUB settings are stored in "`/etc/default/grub`".

9) Generate the GRUB configuration file and reboot:

```
>> sudo update-grub
>> sudo reboot
```

10) Make sure that you booted into the custom kernel:

```
>> uname -r
4.15.18-custom
```

4. Modify and Rebuild Your Kernel

As you probably noticed, downloading and building a Linux kernel from scratch requires some time and effort. Rebuilding a modified kernel, however, is easier and faster because the “make” utility rebuilds only the necessary object files and skips most of the work.

Enter the kernel source directory `~/linux-4.15.18-custom/` and edit `init/main.c` by inserting the following print statement in the `run_init_process()` function:

```
static int run_init_process(const char *init_filename)
{
      printk("Hello, Kernel!\n");
      argv_init[0] = init_filename;
      return do_execve(getname_kernel(init_filename),
            (const char __user *const __user *)argv_init,
            (const char __user *const __user *)envp_init);
}
```

Now rebuild and reinstall the modified kernel:

1) Recompile the kernel image (only `bzImage`, without the modules):

```
>> make -j $(nproc) bzImage
```

2) Copy the new `bzImage` to `/boot` (there’s no need to modify the `initramfs`):

```
>> sudo cp -f arch/x86/boot/bzImage /boot/vmlinuz-4.15.18-custom
```

Note that “make install” will also work in this stage, but it will take a few more seconds and create another copy (which we don’t need) of the old `bzImage` under `/boot/vmlinuz-4.15.18-custom-old`.

3) Reboot to load the modified kernel:

```
>> sudo reboot
```

4) Validate that your print appears in the kernel log:

```
>> dmesg | grep Hello
[    2.144190] Hello, Kernel!
```

If your modified kernel didn’t boot properly, you can fix it from a working vanilla kernel. Remember: GRUB lets you choose the kernel image to be loaded from the disk at boot stage.

The GRUB menu stays for only a few seconds on your screen, so be quick to catch it ;)

5. Background: System Calls in Linux/x86-64

Let us now recall what system calls are and how they are implemented in Linux on x86-64 machines. You should be familiar with this content because you already learned it in the “Computer Organization and Programming” course (234118), a.k.a. ATAM.

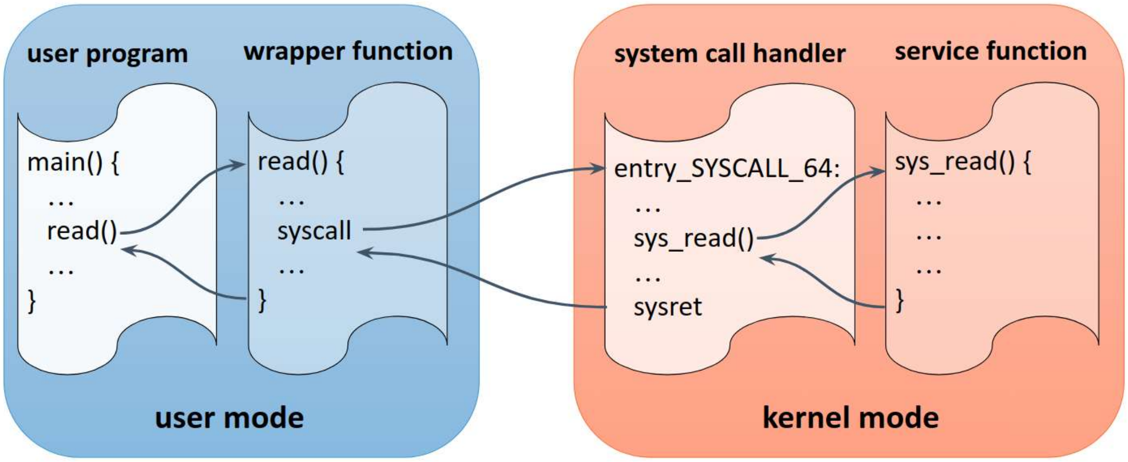
System calls are the way user code requests a service from the OS kernel. At first sight, a system call looks exactly like a typical function call in C. For example, the `read()` system call,

```
#include <unistd.h>
ssize_t read(int fd, void *buf, size_t count);
```

can be used by any C program to read `count` bytes from the file specified by `fd` to the buffer starting at address `buf`.

There is, however, a significant difference between system calls and function calls: system calls should run in a higher privilege level (kernel mode), whereas “regular” functions can be executed in a lower privilege level (user mode). This distinction is important to provide privacy and security: if we allowed “regular” functions to run in kernel mode, then user programs would be able to run privileged machine instructions and read any file on the disk.

System calls are implemented both in user space and in kernel space, as shown in the figure:



5.1 User-Side Implementation

User applications run user mode, which means the hardware restricts what applications can do. For example, an application running in user mode can’t access I/O devices (e.g., the disk) directly via the `in`/`out` machine instructions. User applications that want to access the disk, for example to read a file, should invoke the `read()` system call. When a program invokes the `read()` system call, it actually calls the wrapper function implemented in the standard C library, or `libc` in short.

The wrapper function is named so because it “wraps” the special machine instruction `syscall`. On x86-64 processors, this instruction transfers control (i.e., jumps) into the OS kernel while simultaneously raising the CPU privilege level. The wrapper function is also responsible to put the system call arguments in well-known locations (in specific registers) and to put the system-call number into a well-known location as well (in the RAX register). When the system call returns, the wrapper code unpacks the value returned from the kernel (in the RAX register) and returns control to the program that issued the system call. `libc` wrapper functions are coded in assembly, as they need to carefully follow convention in order to process arguments and return values correctly, as well as execute the hardware-specific `syscall` instruction. You can explore the libc code here, but beware: navigating through `libc` is very challenging as large parts of its code are auto-generated.

5.2 Kernel-Side Implementation

The `syscall` instruction makes a controlled transition into the OS. It points the RIP register to a pre-specified system call handler (which the OS sets up at boot time) and simultaneously raises the privilege level to kernel mode. The Linux system call handler is `entry_SYSCALL_64` is defined in the kernel source file `arch/x86/entry/entry_64.S` . The system call handler reads the system call number from the RAX register and dispatches the corresponding service function. All service functions are listed in the system call table (not shown in the above figure), which is defined in the kernel source file `arch/x86/entry/syscalls/syscall_64.tbl` . The `sys_read()` service function, for example, is stored in the first entry of this table because the number reserved for the `read()` system call is 0.

Service functions run in kernel mode, and as such have full access to the hardware of the system. The service functions are responsible for doing the “real” work, for example, initiating an I/O request or allocating more memory to a program. When the OS is done servicing the request, it passes control back to the user via the special `sysret` instruction, which reverts to user mode while simultaneously passing control back to where the application left off.

Note the clear borderline between user space and kernel space. User programs cannot include kernel headers in their code and cannot call kernel functions directly. In other words, your program can’t simply call the `sys_read()` service function to read a file from the disk. Similarly, kernel code does not call user-space functions like `printf()`, does not include user-space header like `<stdio.h>` or `<iostream>`, and does not link against user-space libraries like `libc`. The only gate to kernel mode (and OS services) that’s the user can use is the `syscall` instruction as described above.

6. Add Your First System Call

This section will take you through the steps required for implementing a new system call. Following the standard practice, you will implement an `hello()` system call, which simply dumps an “Hello, World!” message to the kernel log buffer.

6.1 Kernel-Side Implementation

Please take the following instructions in your terminal:

5) Enter the kernel source directory:

```
>> cd ~/linux-4.15.18-custom/
```

6) Declare the system call in `include/linux/syscalls.h` :

```
...
asmlinkage long sys_statx(int dfd, const char __user *path, unsigned
flags, unsigned mask, struct statx __user *buffer);
asmlinkage long sys_hello(void); // only add this

#endif
```

Reminder: “long” is a shorthand for “long int”, which is a 64-bit signed integer in Linux systems (according to the LP64 data model).

7) Reserve a service number in `arch/x86/entry/syscalls/syscall_64.tbl` :

```
...
332   common      statx             sys_statx
333   common      hello             sys_hello
...
```

The above line reserves the first available number in the “64-bit system call” list.

8) Create a new file called `kernel/hw2.c` and implement the system call there:

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

asmlinkage long sys_hello(void) {
    printk("Hello, World!\n");
    return 0;
}
```

9) Add the new file (`kernel/hw2.c`) to the build process by listing it in `kernel/Makefile` :
(Add only what’s in bold)

```
# SPDX-License-Identifier: GPL-2.0
#
# Makefile for the linux kernel.
#

obj-y     = fork.o exec_domain.o panic.o \
          cpu.o exit.o softirq.o resource.o \
          sysctl.o sysctl_binary.o capability.o ptrace.o user.o \
          signal.o sys.o umh.o workqueue.o pid.o task_work.o \
          extable.o params.o \
          kthread.o sys_ni.o nsproxy.o \
          notifier.o ksysfs.o cred.o reboot.o \
          async.o range.o smpboot.o ucount.o hw2.o
...
```

10) Rebuild the modified kernel and reboot as explained in the previous section:

```
>> make -j $(nproc)
>> sudo cp -f arch/x86/boot/bzImage /boot/vmlinuz-4.15.18-custom
>> sudo reboot
```

6.2 User-Side Implementation

After the OS is loaded properly, we would like to test the system call from user-level code.

1) Write a simple C program under `~/test_hello.cxx` :

```
#include <iostream>
#include <unistd.h>

int main() {
    long r = syscall(333);
    std::cout << "sys_hello returned " << r << std::endl;
    return 0;
}
```

   We stress that this user code is not part of the kernel source code; rather, this code will run in user privileges (CPL=3) and call the kernel `hello()` system call.

2) Build and run the user program:

```
>> g++ ~/test_hello.cxx -o ~/test_hello
>> ./test_hello
sys_hello returned 0
```

3) Validate that your “Hello, World!” was printed to the kernel log:

```
>> dmesg | grep Hello
[    2.144190] Hello, Kernel!
[   80.465532] Hello, World!
```

Note how we used the `syscall()` function in our `test_hello` program. This function provides an easy way for invoking a system call that has no `libc` wrapper function, like our new `hello()` system call. Make sure that you understand what `syscall()` does and how it works. How many parameters can `syscall()` get? Why do we pass `333` as an argument? Reading the “`man 2 syscall`” page is a good place to start.

7. Add Support for Process Syscall Banning

Now we have what we need in order to do some real work.
Your mission is to add a new functionality to the linux kernel called **process syscall bans**.
The modified kernel will have the option to ban processes from using certain system calls – **getpid()**, **pipe()**, and **kill()**.
Each process can have any combination of these bans, and we would be able to set and check them using system calls. When a process is forked, the child process should inherit all the bans the parent had. Init will have no bans at startup.
To keep the modified kernel as lightweight as possible, **you are forbidden from increasing the size of the PCB struct by more than one byte** (enough for one char sized variable).
Before you start writing code, ask yourselves: how should we store each process bans?
To this end, you will implement four new system calls, as described below.
You are required to implement these four system calls in `kernel/hw2.c` and submit this file, along with any other files you changed in the linux kernel.
We remind you that kernel functions that implement system calls should start with `sys_` , as was demonstrated in the implementation of the `hello()` system call from the previous section.

7.1 `long set_ban(int ban_getpid, int ban_pipe, int ban_kill)`

System call number: 334.

Action: Sets bans for the current process. Requires root privileges.

Arguments: `ban_getpid` – 1 bans the process from using getpid(), 0 lifts the ban
`ban_pipe` – 1 bans the process from using pipe(), 0 lifts the ban
`ban_kill` – 1 bans the process from using kill(), 0 lifts the ban

Note: arguments greater than 1 are to be considered as 1.

Return value:

- On success, return 0.
- On failure,
  - `-EINVAL` at least one invalid argument was specified (e.g. negative value).
  - `-EPERM` the calling process does not have root privileges.

Note: the priority of errors corresponds to the order in which they are listed. you should return the error with the highest priority when multiple failures occur.
example - if an invalid argument was specified and the calling process doesn't have root privileges, you should return `–EINVAL`.

We remind you that system calls report failures to user space by returning negative values. In such cases, the `libc` wrapper function will store the absolute return value in the `errno` global variable and return `-1` to the user-level code that invoked the system call.

7.2 `long get_ban(char ban)`

System call number: 335.

Action: Returns true if the current process is banned from using the syscall, or false if it isn't.

Arguments: `ban` – the ban to check, as the syscall's first char (e.g. 'p' for "pipe", lowercase).

Return value:

- On success, returns true (1) if the process is banned from using the syscall represented by 'ban', false (0) if it isn't.
- On failure,
  - `-EINVAL` an invalid ban was specified (any char other than 'g', 'p', 'k').

7.3 `long check_ban(pid_t pid, char ban)`

System call number: 336.

Action: Can be used to check if a process has a specific ban. The calling process must not be banned from using the syscall represented by 'ban' in order to perform the check.

Arguments: `pid` – the pid of the process we want to check.
`ban` – the ban to check, as the syscall's first char (e.g. 'k' for "kill", lowercase).

Return value:

- On success, returns true (1) if the process is banned from using the syscall represented by 'ban', false (0) if it isn't.
- On failure,
  - `-EINVAL` an invalid ban was specified (any char other than 'g', 'p', 'k').
  - `-ESRCH` the target process does not exist.
  - `-EPERM` the calling process is banned from using the syscall 'ban'.

7.4 `long flip_ban_branch(int height, char ban)`

System call number: 337.

Action: Inverts a specific ban of the direct parents of the calling process (father, great father, great great father etc). lifts the ban if it exists or imposes it otherwise.

Arguments: `height –` limits the number of direct parents we'll try to update. In other words, the maximum height we will travel up the family tree of the process to update parents. non-positive values of height should be considered invalid.
`ban` – the ban to flip, as the syscall's first char (e.g. 'g' for "getpid", lowercase).

Return value:

- On success, return the number of direct parents that were banned as a result of this syscall.
- On failure,
  - `-EINVAL` at least one invalid argument was specified.
  - `-EPERM` the calling process is banned from using the syscall 'ban'.

8. Ban Processes from Using Syscalls

8.1 Altering Syscall Implementations

Now that we can track which syscalls are banned, we are ready to implement blocking itself. You now need to find the implementation of the relevant syscalls (getpid, pipe, kill) and alter them so they return –EPERM when the process is banned from calling them. –EPERM must be returned BEFORE any work is done (for example, we don't want to return –EPERM after we have already sent a signal with kill(), or allocated a pipe with pipe()).

The syscalls are found in these files:

```
kernel/sys.c    (sys_getpid())
kernel/signal.c (sys_kill()  )
fs/pipe.c       (sys_pipe()  )
```

Note: If you have trouble finding the syscall implementations, try to use the following site:
https://elixir.bootlin.com/linux/v4.15.12
It allows you to easily search symbols like function names in the linux source code.

Note: You may notice the syscalls are defined using a macro – SYSCALL_DEFINEN.
This macro takes the name of the syscall and it's parameters and defines a matching sys_ syscall implementation. because of this, sys_kill will appear as SYSCALL_DEFINE2(kill, …).
You should not use this macro to define your syscalls in hw2.c, as this will create problems.

8.2 Blocking Syscalls Effectively (Bonus)

In this assignment you were tasked with blocking only 3 syscalls. What if we wanted to block more, or maybe whole groups of syscalls? The method of finding each syscall implementation and altering it would prove difficult since we need to find and understand each implementation. We might even break some of the more complex syscalls while trying to alter them, leading to unforeseen consequences for user processes.
In this section, your mission is to find a better way of implementing syscall banning, which doesn't involve modifying system call implementations.
Implementing this feature successfully without modifying syscall implementations permits 10% bonus reward.
Since this is a bonus section, you will have to research ways to do this on your own, without support from the course staff.

Hint: To achieve this, you only have to change one file – arch/x86/entry/common.c.

Important Notes and Tips

- It is important to remember that we do not require anything regarding the runtime complexity of your system call implementations, so there is no need to try and optimize your implementation if it runs correctly. Also bear in mind that you do not control the process tree, meaning that other kernel functions may change it without changing the fields you added.
- The default behaviour of system calls such as `fork` and `execv` already works well with the new features you added, so you don't have to change them.
- In the implementation of `check_ban()`, you will have to find the task_struct (PCB) of a specific process, using it's PID. since you have access to the kernel source code, you can read the implementation of sys_kill() (or any other syscall that find PCB by PID) for ways to do this effectively.
- If you chose not to implement the bonus, you should modify `kernel/sys.c, kernel/signal.c and fs/pipe.c`. If you decide to go for the bonus, you should modify `arch/x86/entry/common.c` instead.
- The grade calculation formula (with bonus): `grade + 0.1*grade`. 70 becomes 77.
- You are not expected to handle kernel synchronization in HW2.
- We provided you with a simple testing framework for this HW assignment, which comprises the following files:
  - `makefile` - searches the current directory for source files that matches `test*.cxx`, compiles them into `test*.exe` files. `make handin` will create a submission zip from your files (experimental).
  - `hw2_test.h , hw2_test.cxx` – user-level wrapper functions for our system calls.
  - `test1.cxx` - a simple test.

Submission Instructions

You should create a folder containing the following files:

a. The kernel source file `kernel/hw2.c`.
   This file should implement the four system calls described in this assignment: `hello()`, `set_ban()`, `get_ban()`, `check_ban()` and `flip_ban_branch()`.
b. Any other kernel files you might have changed, including but not limited to: `kernel/Makefile, arch/x86/entry/syscalls/syscall_64.tbl` and `include/linux/syscalls.h.`
   Please keep the directory structure as it is in your kernel (for example, put `hw2.c` in the `kernel` directory).
c. If you have chosen to implement the bonus, you should submit `arch/x86/entry/common.c`, and you should not submit any files in `kernel/` or `fs/` other than `Makefile & hw2.c.`
d. A file named `submitters.txt` which includes the ID, name and email of the participating students. The following format should be used:

```
Linus Torvalds linus@gmail.com 234567890
Ken Thompson ken@belllabs.com 345678901
```

Copy the makefile we provided with the assignment into the directory and use `make handin`

This should create a zip that should look like this:

```
final.zip -+
           +- kernel  -+
                       +- hw2.c
                       +- ...
           +- include -+
                       +- linux -+
                                 +- syscalls.h
                                 +- ...
           +- ...
           +- submitters.txt
```

Important Note: when you submit, keep your confirmation code and a copy of the file(s), in case of technical failure. Your confirmation code is the only valid proof that you submitted your assignment when you did.

Don’t forget to smile,
Operating Systems staff