System calls

In this assignment, you are to implement a custom system call handler in the Linux kernel. For system call handling, the OS reserves a vector in the interrupt descriptor table (IDT). Linux 32-bit OS reserves vector 128 in the IDT for system call handling. You have to use vector 15 (currently unused and reserved for exception) for the custom system call handler.

1 Interrupt descriptor table

An interrupt descriptor table (IDT) is a sequential array of descriptors (max 256). The size of each descriptor is 64-bits. The first 32 descriptors are reserved for exception handling, and the others are used for interrupts. A descriptor corresponding to a vector `x` contains the virtual address of the target handler (routine), which is called on every occurrence of interrupt vector `x`. Apart from containing the virtual address of the target interrupt handler (32-bits), the rest 32-bits of the descriptor contain the `cs` segment selector and other flags. You can refer to page-197, fig 6-2 of Intel manual - 3. For this assignment, we are only interested in the address of the target interrupt handler. The other fields can be copied from the entry corresponding to the existing system call handler (vector 128).

2 Kernel module

The IDT can only be modified in the kernel. In this assignment, you are provided a working kernel module. A kernel module is a piece of software that is the part of the kernel. User program can call routines in the kernel module via `ioctl` system call. The `kernel` folder in the assignment repository contains the code corresponding to kernel module. When the user-program does the `ioctl` call for kernel module services, `device_ioctl` (in main.c) is called. `ioctl` takes an identifier and a 32-bit argument (user-mode addresses can also be passed using typecasting). `device_ioctl` invokes the routine corresponding to the input identifier. You need to implement `register_syscall` and `unregister_syscall` ioctls.

3 klib.c

`klib.c` in kernel folder implements the following functionalities.

1. `print_kernel` is the system call handler corresponding to the `PRINT_KERNEL_SYS` system call, which can be invoked using the custom system call handler.
2. `syscall_handler_k` is the custom system call handler that is invoked by the `system_handler` in `ksys.S`.
3. `imp_copy_idt` allocates space for new IDT, copies the contents of the current IDT (loaded in IDTR) to the new IDT, and returns the base and size of the new IDT in the input `struct idt_desc`. `struct idt_desc` contains the base address and size of an IDT.
4. `imp_load_idt` takes two pointers `new` and `old` of type ``` ``struct idt_desc *'' ```. `new` contains the base and size of the target IDT. `imp_load_idt` loads the target IDT in `IDTR` and returns the base and size of the previous IDT in `old`.
5. `imp_free_desc` takes `struct idt_desc` corresponding to an IDT allocated in `imp_copy_idt` and releases the corresponding memory.

4 Implementation

You need to implement the following routines.

1. `system_call`
2. `register_syscall`
3. `unregister_syscall`
4. `syscall_u`

4.1 system_call

`system_call` in "kernel/ksys.S" is the custom system call handler. You need to map the `system_call` routine at index 15 in the IDT. The user passes an identifier and a buffer via registers. You need to modify `system_call` to pass these arguments directly to `syscall_handler_k` in `klib.c`. the kernel uses a fixed value (216) in the `%fs` register. You need to set the kernel's `%fs` register before calling `syscall_handler_k`, but make sure to restore the user's `%fs` before returning to user-mode.

4.2 register_syscall

`register_syscall` in "kernel/main.c" is the handler corresponding to the `REGISTER_SYSCALL` ioctl. This routine allocates a new IDT, modifies the index 15 in the new IDT to point to the `system_call` routine, and loads the new IDT in IDTR. You can use the routines in `klib.c` to implement this functionality. `struct idt_entry` in `klib.h` represents an IDT entry. Here, lower16 and higher16 correspond to lower 16-bits and higher 16-bits address of target interrupt handler. You need to change theses values corresponding to entry at index 15 in the new IDT. The rest of the fields of the IDT entry can be copied from the default system call handler (i.e., index 128).

4.3 unregister_syscall

`unregister_syscall` in "kernel/main.c" is the handler corresponding to the `UNREGISTER_SYSCALL` ioctl. This routine would load the original IDT in IDTR, if the IDTR was changed earlier by register_syscall.

4.4 syscall_u

`syscall_u` in "user/usys.S" is the user-mode routine that invokes the `system_call` using `int $15` instruction. `syscall_u` takes an identifier and a buffer and pass it to the `system_call` via registers.

You are only allowed to modify "kernel/ksys.S", "kernel/main.c", and "user/usys.S" files. You are not allowed to add a new file.

5 User-program

The `user` folder contains the user-program. The "user/syscall.c" is the main program that enables the custom system call handler by doing `REGISTER_SYSCALL` ioctl, invokes the `PRINT_KERNEL_SYS` system call through `syscall_u`, and finally restores the original IDT using `UNREGISTER_SYSCALL` ioctl. If the system call is successful, the system call handler copies the string "syscall done." to the user's buffer.

6 Environment

For this assignment, you need to clone the assignment repo from `https://github.com/Systems-IIITD/syscall`.
Download Linux 32-bit iso from `http://ba.releases.ubuntu.com/16.04/`.

You need to create a virtual machine using this `iso` image. For this assignment, we need a 32-bit Linux OS. We have tested the assignment for the provided Linux distribution (iso image). You have to use the above `iso` image for this assignment.

6.1 Compilation and running

To compile:
"cd kernel && make" (compiles the kernel module)
"cd user && make" (compiles the user program)

To run:
"cd kernel && sudo ./load" (loads the kernel module).
"cd user && ./syscall" (executes the user program).

The kernel equivalent of `printf` is `printk`. You can run `dmesg` to print the kernel log (i.e., log generated using printk, etc.) on the terminal.

Before running the user-program, you have to load the kernel module. `start_module` in "main.c" is called automatically when the module is loaded. If you run `dmesg` at this point, you should see "module loaded successfully". `make clean` removes the executables and temporary files.

7 Design documentation

You also have to submit design documentation along with your implementation; otherwise, the assignment will not be graded. Answer the following questions in your design documentation.

- Paste your code corresponding to `system_call`.
- Paste your code corresponding to `u_syscall`.
- Paste your code corresponding to `register_syscall`.
- Paste your code corresponding to `unregister_syscall`.
- The output of user-program.
- How do you know the location of the original IDT in `unregister_syscall`?
- How do you know the location of the current IDT in `unregister_syscall`?
- If somebody calls `unregister_syscall` twice, without calling `register_syscall` in between.
  - Do you load the original IDT twice?
  - Do you free the current IDT twice?

7.1 How to submit.

To be done individually. Submit a zip folder that contains four files: "user/usys.S", "kernel/ksys.S", "kernel/main.c", and design documentation (in pdf format). Please make sure that your implementation is not printing any debug messages before submitting the final code. The submission link is on backpack.