LINUX KERNEL DEVELOPMENT

CSL3030
FALL 2022

Abhishek Rajora (B20CS002)
Abu Shahid (B20CS003)
Ayush Anand (B20CS082)

0 1 OBJECTIVE

- **To implement the learnings of CSL3030**

With this project, we wanted to implement what we studied during the semester, and if we cannot implement it, then at least observe and understand it's working from under the hood.

- **To understand the core workings of the Linux kernel.**

Instead of abstracting a separate topic based on our interests, we decided to toy around with the actual Linux kernels. This required access and a deeper understanding of the code base. This factor also determined the choice of distributions we, later on, chose to work on.

- **To understand the syscall interface in Linux**

We decided to play around with the syscall interface of the linux-6.0 kernel, To be precise we decided to add an additional syscall which prints something to the kernels log files.

- **To understand the source code**

The main reason for undertaking this project was to explore and understand the codebase of a commercial and highly in demand operating system.

Tasks Undertaken:

- **Building and Executing redox OS:**
  - Setting up Rust Architecture
  - Installing package managers
  - Cloning the OS repo
  - Compiling and building
- **Sandboxing Redox OS**
  - Since the setup was manual, we had complete access to the source code. This is where we did all the playing around; followed by subsequent compilation after each implementation.
- **Hacking Linux Kernel**
  - Building and compiling the linux kernel
  - Adding a syscall to the kernel.

02 METHODOLOGY

The first step in choosing the code base is picking out the most sophisticated OS with next generation file systems. We chose Redox OS to fulfill this purpose. Redox OS is a general purpose operating system written completely in Rust. It provides UNIX like micro-kernel for both security and independence.

Redox aims to synthesize years of research and hard won experience into a system that feels modern and familiar. With 16,000 lines of kernel code, most requests are serviced in user mode. The smaller amount of code in kernel mode results in nearly negligible bugs and security issues unlike LINUX with monolithic kernel.

Moreover Rust attempts to avoid the unexpected memory unsafe conditions which are a major source of security critical bugs.

Setting up Redox OS:

1. **Installing Rust tool-chain**

```
$ cd path/to/your/projects/folder/
$ curl https://sh.rustup.rs -sSf | sh
$ source $HOME/.cargo/env
```

2. **Clone the source**

```
$ git clone https://gitlab.redox-os.org/redox-os/redox.git --origin
  upstream --recursive
```

3. **Compile and Build the kernel**

```
$ cargo install xargo cargo-config
$ git submodule update --recursive --init
$ make all
$ make qemu
 // Launch using qemu without using kvm nor graphics
$ make qemu kvm=no vga=no
```

Setting up Linux kernel:

1. **Clone the source**

```
$ cd path/to/your/projects/folder/
$ git clone
git://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git
```

2. **Compile and Build the kernel image**

```
 $ make deconfig
 $ make oldconfig
 // 8 simultaneous jobs for compiling and building
 $ make -j8
```

3. **Mount a filesystem**

```
// To emulate the kernel image in qemu emulator we need
    a filesystem, to do that we use syzkaller script
$ ./create-image.sh
```

Adding syscall to the linux kernel:

1. **First add a valid entry to syscall table**

```
$ cd arch/x86/entry/syscalls/
$ vim syscall_64.tbl
```

add a valid entry like this "451 common hello_there sys_hello_there"

Now add the following snippet in kernel/sys.c file to define the syscall.

```
SYSCALL_DEFINE0(hello_there){
    printk(KERN_INFO "xyz");
    return 0;
 };
```

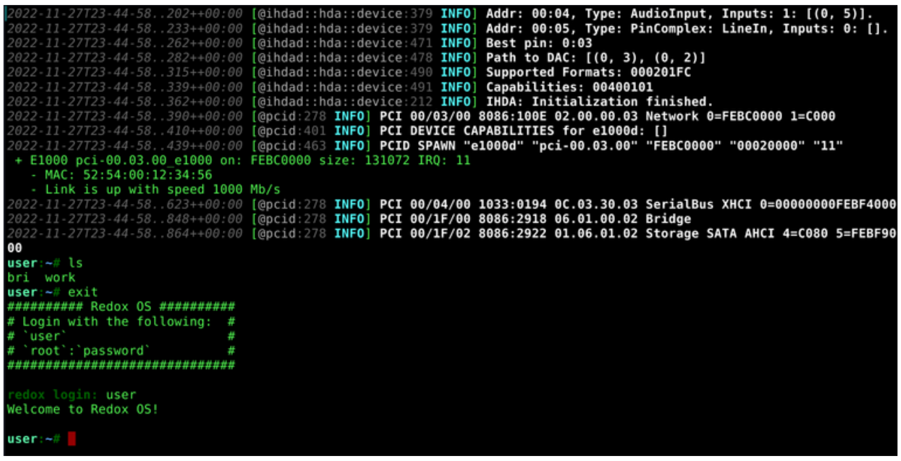
2. **Compile and Build the kernel image again**

```
 $ make deconfig
 $ make oldconfig
 // 8 simultaneous jobs for compiling and building
 $ make -j8
```

3. **After QEMU lauch, test the syscall using syscall(451) function**

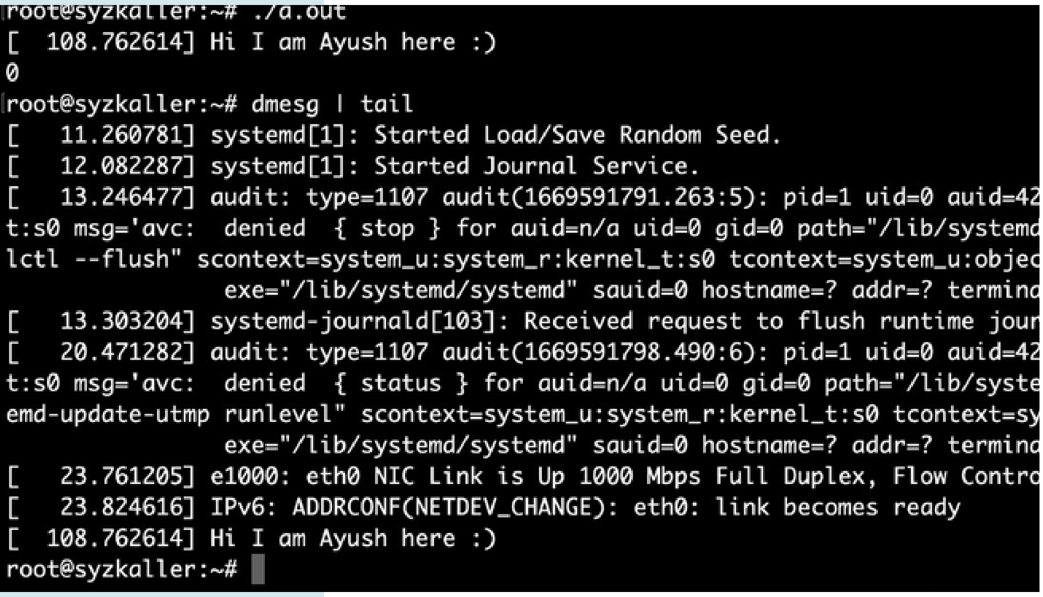
03 IMPLEMENTATION & OUTCOMES

After the successful build, we call qemu emulator to lauch the redox os on terminal

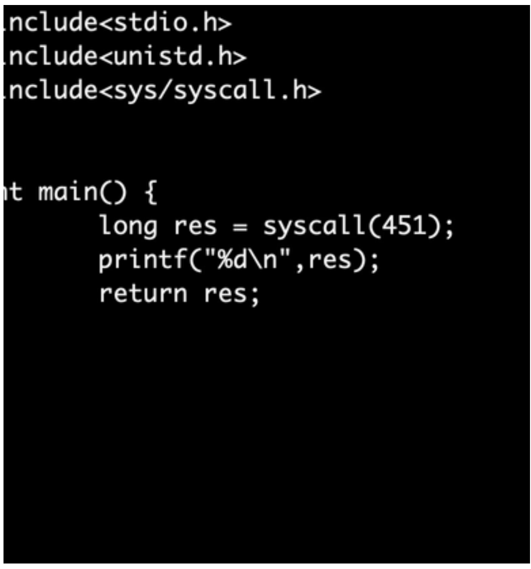


```
2022-11-27T23-44-58..202++00:00 [@ihdad::hda::device:379 INFO] Addr: 00:04, Type: AudioInput, Inputs: 1: [(0, 5)].
2022-11-27T23-44-58..233++00:00 [@ihdad::hda::device:379 INFO] Addr: 00:05, Type: PinComplex: LineIn, Inputs: 0: [].
2022-11-27T23-44-58..262++00:00 [@ihdad::hda::device:471 INFO] Best pin: 0:03
2022-11-27T23-44-58..282++00:00 [@ihdad::hda::device:478 INFO] Path to DAC: [(0, 3), (0, 2)]
2022-11-27T23-44-58..315++00:00 [@ihdad::hda::device:490 INFO] Supported Formats: 000201FC
2022-11-27T23-44-58..339++00:00 [@ihdad::hda::device:491 INFO] Capabilities: 00400101
2022-11-27T23-44-58..362++00:00 [@ihdad::hda::device:212 INFO] IHDA: Initialization finished.
2022-11-27T23-44-58..390++00:00 [@pcid:278 INFO] PCI 00/03/00 8086:100E 02.00.00.03 Network 0=FEBC0000 1=C000
2022-11-27T23-44-58..410++00:00 [@pcid:401 INFO] PCI DEVICE CAPABILITIES for e1000d: []
2022-11-27T23-44-58..439++00:00 [@pcid:463 INFO] PCID SPAWN "e1000d" "pci-00.03.00" "FEBC0000" "00020000" "11"
 + E1000 pci-00.03.00_e1000 on: FEBC0000 size: 131072 IRQ: 11
   - MAC: 52:54:00:12:34:56
   - Link is up with speed 1000 Mb/s
2022-11-27T23-44-58..623++00:00 [@pcid:278 INFO] PCI 00/04/00 1033:0194 0C.03.30.03 SerialBus XHCI 0=00000000FEBF4000
2022-11-27T23-44-58..848++00:00 [@pcid:278 INFO] PCI 00/1F/00 8086:2918 06.01.00.02 Bridge
2022-11-27T23-44-58..864++00:00 [@pcid:278 INFO] PCI 00/1F/02 8086:2922 01.06.01.02 Storage SATA AHCI 4=C080 5=FEBF90
00
user:~# ls
bri  work
user:~# exit
########## Redox OS ##########
# Login with the following:  #
# `user`                     #
# `root`:`password`          #
##############################

redox login: user
Welcome to Redox OS!

user:~# 
```

The implemented syscall in the linux-6.0 kernel runs as expected and prints a message to the kernel log.

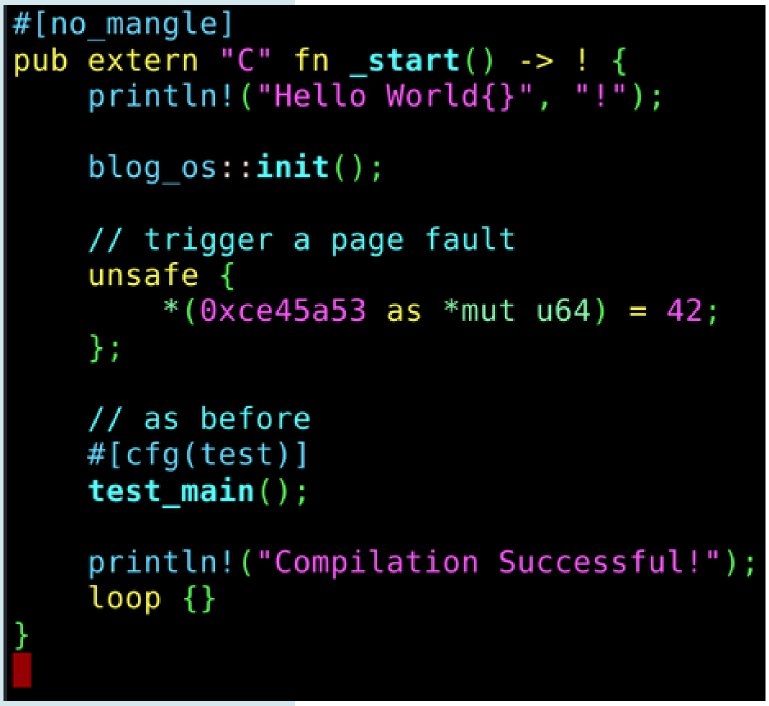


```
root@syzkaller:~# ./a.out
[  108.762614] Hi I am Ayush here :)
0
root@syzkaller:~# dmesg | tail
[   11.260781] systemd[1]: Started Load/Save Random Seed.
[   12.082287] systemd[1]: Started Journal Service.
[   13.246477] audit: type=1107 audit(1669591791.263:5): pid=1 uid=0 auid=42
t:s0 msg='avc:  denied  { stop } for auid=n/a uid=0 gid=0 path="/lib/systemd
lctl --flush" scontext=system_u:system_r:kernel_t:s0 tcontext=system_u:objec
                exe="/lib/systemd/systemd" sauid=0 hostname=? addr=? termina
[   13.303204] systemd-journald[103]: Received request to flush runtime jour
[   20.471282] audit: type=1107 audit(1669591798.490:6): pid=1 uid=0 auid=42
t:s0 msg='avc:  denied  { status } for auid=n/a uid=0 gid=0 path="/lib/syste
emd-update-utmp runlevel" scontext=system_u:system_r:kernel_t:s0 tcontext=sy
                exe="/lib/systemd/systemd" sauid=0 hostname=? addr=? termina
[   23.761205] e1000: eth0 NIC Link is Up 1000 Mbps Full Duplex, Flow Contro
[   23.824616] IPv6: ADDRCONF(NETDEV_CHANGE): eth0: link becomes ready
[  108.762614] Hi I am Ayush here :)
root@syzkaller:~# 
```



```
nclude<stdio.h>
nclude<unistd.h>
nclude<sys/syscall.h>


nt main() {
        long res = syscall(451);
        printf("%d\n",res);
        return res;
```

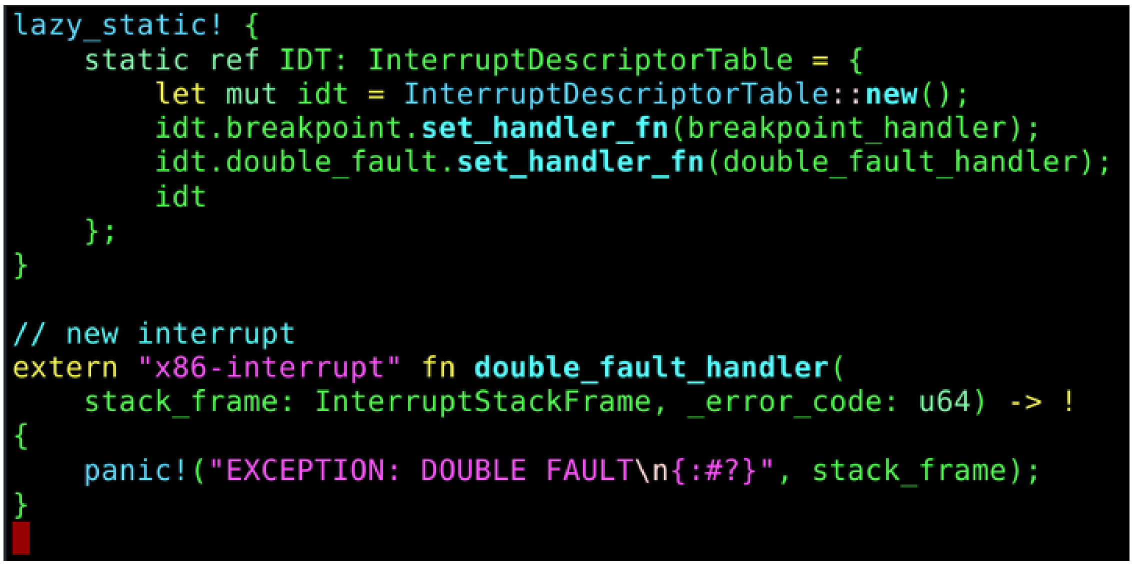
In our exploration of Redox OS, we came across the Interrupt Descriptor table and attempted to seed a Double Fault. Double faults occur when CPU fails to invoke exception handler. We prevent fatal triple faults that require a system reset by addressing this exception. We also set up an Interrupt Stack Table to detect double faults on a different kernel stack in order to always prevent triple faults.
To trigger we double fault, the following snippet can be pasted in for which handler does not exist.



```
#[no_mangle]
pub extern "C" fn _start() -> ! {
    println!("Hello World{}", "!");

    blog_os::init();

    // trigger a page fault
    unsafe {
        *(0xce45a53 as *mut u64) = 42;
    };

    // as before
    #[cfg(test)]
    test_main();

    println!("Compilation Successful!");
    loop {}
}
```

We here are using `unsafe` to write to the invalid address **`0xce45a53`** (**read:ceyxasm :)**) Since this virtual address is not mapped. page fault spirals into a double fault.

On compiling and starting our kernel, it enters an endless boot cycle. This happens because since no handler function is specified, page handler cannot be called and causes a double fault which then causes a triple fault. QEMU crashes.
This can be taken care of by simply specifying the handler function:



```
lazy_static! {
    static ref IDT: InterruptDescriptorTable = {
        let mut idt = InterruptDescriptorTable::new();
        idt.breakpoint.set_handler_fn(breakpoint_handler);
        idt.double_fault.set_handler_fn(double_fault_handler);
        idt
    };
}

// new interrupt
extern "x86-interrupt" fn double_fault_handler(
    stack_frame: InterruptStackFrame, _error_code: u64) -> !
{
    panic!("EXCEPTION: DOUBLE FAULT\n{:#?}", stack_frame);
}
```

04 LESSONS LEARNED

We learned several useful things about developing on a linux kernel. It was difficult to get the installation done right on some systems such as the mac os. This was due to the lack of support of elf.c header files by the mac system. Although it is possible to build linux system on a mac, it is far from convenient and is extremely hacky. Therefore we had to resort to using a google cloud instance which had ubuntu on it. This helped in building the linux kernel in a very hasslefree manner. We also got to see the raw code of Round-Robin in RedoxOS and could feel the gap of learning and implementation bridging. (PS: an attempt to play around with the scheduler code did not turn out well; 2 iterations of OS crashes after 3 hour compilation time :/)