

ELL783 Assignment1_hard

Ananya Sharma 2023EET2186

February 2024

1 Linux Installation: Virtual Machine Setup

I am using Oracle VM VirtualBox Manager to run Linux on a Windows host. I first download the Ubuntu Server ISO image from [here](#). I have used the Ubuntu 22.04.03 version. After that, I created a Virtual Machine that has 4 CPU cores, 8GB RAM, and a 50 GB disk and I Installed the Ubuntu server ISO on the virtual machine.

2 Building the Linux Kernel

I installed the necessary tools for installing the Linux kernel: using the following command

```
$ sudo apt-get install git fakeroot build-essential ncurses-dev \
xz-utils libssl-dev bc flex libelf-dev bison
```

'**sudo apt-get**' to install several packages commonly used in building and compiling software on a Linux system. We have installed **git**, which is a Version control system; **fakeroot**, which allows non-privileged users to create fake root environments; and **build-essential**, which is a meta-package that includes commonly used tools for building software like gcc,g++, and make; **ncurses-dev**, has development files for the ncurses library, often used in terminal-based applications; **xz-utils** is utilities for working with the XZ compression format; **libssl-dev** has development files for the OpenSSL library, commonly used for secure communication; **bc** is arbitrary precision numeric processing language; **flex** is fast lexical analyzer generator; **libelf-dev** has development files for the ELF (Executable and Linkable Format) library; **bison** is GNU Bison, a parser generator.

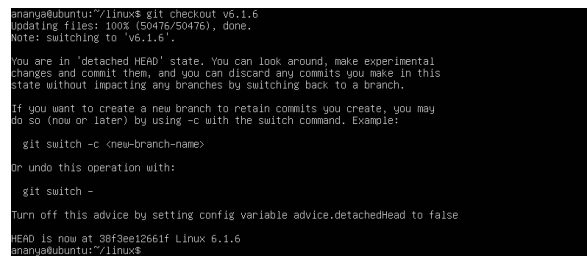
Then, I downloaded the source code of the latest stable Linux kernel using the following command, cloning the Linux kernel source code repository.

```
$ git clone \
git://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git
```

After that, using the following command, we clone the Linux kernel repository, move into the cloned directory, and switch to the specified version (v6.1.6), as shown in figure 1

```
$ cd linux
$ git checkout v6.1.6
```

To compile and run Linux on the virtual machine we used used the following commands:



```
ananya@ubuntu:~/linux$ git checkout v6.1.6
Updating files: 100% (50476/50476), done.
Note: switching to 'v6.1.6'.

You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by switching back to a branch.

If you want to create a new branch to retain commits you create, you may
do so (now or later) by using -c with the switch command. Example:

  git switch -c <new-branch-name>

Or undo this operation with:

  git switch -

Turn off this advice by setting config variable advice.detachedHead to false

HEAD is now at 38f3ee12661f Linux 6.1.6
ananya@ubuntu:~/linux$
```

Figure 1: HEAD now points at latest kernel version

```

CHK      kernel/kheaders_data.tar.xz
GEN      kernel/kheaders_data.tar.xz
AR       kernel/built-in.a
CC [M]   kernel/kheaders.o
make: *** [Makefile:1992: .] Error 2
ananya@ubuntu:~/linux$ _

```

Figure 2: Makefile ERROR 2

```

drivers/net/ethernet/qualcomm/emac/emac-mac.c:1480:1: fatal error: error writing to /tmp/ccFMoWuh.s:
No space left on device
1480 | }
      | ^
compilation terminated.
make[6]: *** [scripts/Makefile.build:250: drivers/net/ethernet/qualcomm/emac/emac-mac.o] Error 1
make[5]: *** [scripts/Makefile.build:500: drivers/net/ethernet/qualcomm/emac] Error 2
make[5]: *** Waiting for unfinished jobs....
CC [M]   drivers/net/ethernet/qualcomm/rmnet/rmnet_map_data.o
drivers/net/ethernet/qualcomm/rmnet/rmnet_vnd.c:359:1: fatal error: error writing to /tmp/ccQgxTvN.s:
: No space left on device
359 | }
     | ^
drivers/net/ethernet/qualcomm/rmnet/rmnet_config.c:518:1: fatal error: error writing to /tmp/ccAtTph
n.s: No space left on device
518 | MODULE_LICENSE("GPL v2");
     | ~~~~~
compilation terminated.
compilation terminated.
make[6]: *** [scripts/Makefile.build:250: drivers/net/ethernet/qualcomm/rmnet/rmnet_config.o] Error
1
make[6]: *** Waiting for unfinished jobs....
make[6]: *** [scripts/Makefile.build:250: drivers/net/ethernet/qualcomm/rmnet/rmnet_vnd.o] Error 1
drivers/net/ethernet/qualcomm/rmnet/rmnet_map_data.c:520:1: fatal error: error writing to /tmp/ccxKH
8ui.s: No space left on device
520 | }
     | ^
compilation terminated.
make[6]: *** [scripts/Makefile.build:250: drivers/net/ethernet/qualcomm/rmnet/rmnet_map_data.o] Erro
r 1
make[5]: *** [scripts/Makefile.build:500: drivers/net/ethernet/qualcomm/rmnet] Error 2
make[4]: *** [scripts/Makefile.build:500: drivers/net/ethernet/qualcomm] Error 2

```

Figure 3: No space left on disk

```
$ cp -v /boot/config-$(uname -r) .config
```

I faced MakeFile error 2, which was related to system certificates, as shown in figure 2, So I opened the .config file using vim editor and set the

CONFIG_SYSTEM_TRUSTED_KEYS and SYSTEM_REVOCATION_KEYS option to "",

and configure the features to be compiled using .config file that can be generated using the make menuconfig command. A default configuration is generated by running the make menuconfig. I saved it without modifying any configuration. Then, I faced another error related the memory shown in figure 3. Initially, I configured 50GB, so this time I increased the memory and I built it again from the beginning.

Then, using the following commands shown in figure 4, I have set the num_cores to 4

```
$ sudo make -j 4
```

This command is used to compile the Linux kernel.

```
$ sudo make modules_install -j 4
```

After compiling the kernel, we installed the kernel modules.

```
$ sudo make install -j 4
```

```

ananya@ubuntu:~/linux$ sudo make install -j 4
INSTALL /boot
run-parts: executing /etc/kernel/postinst.d/initramfs-tools 6.1.6 /boot/vmlinuz-6.1.6
update-initramfs: Generating /boot/initrd.img-6.1.6
run-parts: executing /etc/kernel/postinst.d/unattended-upgrades 6.1.6 /boot/vmlinuz-6.1.6
run-parts: executing /etc/kernel/postinst.d/update-notifier 6.1.6 /boot/vmlinuz-6.1.6
run-parts: executing /etc/kernel/postinst.d/xx-update-initrd-links 6.1.6 /boot/vmlinuz-6.1.6
E: /boot/initrd.img is now a symlink to initrd.img-6.1.6
run-parts: executing /etc/kernel/postinst.d/zz-update-grub 6.1.6 /boot/vmlinuz-6.1.6
Sourcing file /etc/default/grub
Sourcing file /etc/default/grub.d/init-select.cfg
Generating grub configuration file ...
Found linux image: /boot/vmlinuz-6.1.6
Found initrd image: /boot/initrd.img-6.1.6
Found linux image: /boot/vmlinuz-5.15.0-91-generic
Found initrd image: /boot/initrd.img-5.15.0-91-generic
Warning: os-prober will not be executed to detect other bootable partitions.
Systems on them will not be added to the GRUB boot configuration.
Check GRUB_DISABLE_OS_PROBER documentation entry.
done
ananya@ubuntu:~/linux$ _

```

Figure 4: sudo make install -j 4

```

"/etc/default/grub" 33L, 1199B written
ananya@ubuntu:~/linux$ sudo update-grub
Sourcing file /etc/default/grub
Sourcing file /etc/default/grub.d/init-select.cfg
Generating grub configuration file ...
Found linux image: /boot/vmlinuz-6.1.6
Found initrd image: /boot/initrd.img-6.1.6
Found linux image: /boot/vmlinuz-5.15.0-91-generic
Found initrd image: /boot/initrd.img-5.15.0-91-generic
Warning: os-prober will not be executed to detect other bootable partitions.
Systems on them will not be added to the GRUB boot configuration.
Check GRUB_DISABLE_OS_PROBER documentation entry.
done
ananya@ubuntu:~/linux$ _

```

Figure 5: Linux Kernel build completed

and then, we install the newly compiled Linux kernel. Here **-j** flag specifies the number of parallel jobs that can run during the build process. This will speed up the compilation process by utilizing multiple cores. The build took 6-7 hours even though my system specifications were good. Then, I opened the grub configuration file using the vi editor

```
$ sudo vi /etc/default/grub
```

and set the GRUB_TIMEOUT to 60 and commented GRUB_TIMEOUT_STYLE option i.e., GRUB will wait for 60 seconds before automatically booting the default operating system. Then, using the below command, my Linux kernel build was complete, as shown in figure 5. Once the kernel is compiled, it will be available at **arch/x86_64/boot/bzImage**. I restarted my virtual machine and selected my latest kernel from the grub menu.

```
$ sudo update-grub
```

3 Context Switch Tracker

3.1 System Call Implementation

This part had 3 sections in which we had to create 3 system calls. Initially, I started with a dummy i.e. "Hello World" system call. Once my system calls will return 0 i.e. successful. I will be focusing on the Implementation part. For creating system calls, I first created a directory **mysystemcalls** and created 3 separate .c files. Figure 10 shows a dummy system call that just prints to kernel log files, which the command dmesg can access. Then, I created a Makefile using the command

```
$ vi Makefile
```

```

539 x32 process_vm_readv compat_sys_process_vm_readv
540 x32 process_vm_writev compat_sys_process_vm_writev
541 x32 setsockopt compat_sys_setsockopt
542 x32 getsockopt compat_sys_getsockopt
543 x32 io_setup compat_sys_io_setup
544 x32 io_submit compat_sys_io_submit
545 x32 execveat compat_sys_execveat
546 x32 preadv2 compat_sys_preadv64v2
547 x32 pwritev2 compat_sys_pwritev64v2
548 64 register sys_register
549 64 fetch sys_fetch
550 64 deregister sys_deregister
# This is the end of the legacy x32 range. Numbers 548 and above are
# not special and are not to be used for x32-specific syscalls.
"syscall_64.tbl" 422L, 14927B

```

Figure 6: syscall_64.tbl

```

# Ordinary directory descending
# -----
obj-y      += init/
obj-y      += usr/
obj-y      += arch/$(SRCARCH)/
obj-y      += $(ARCH_CORE)/
obj-y      += kernel/
obj-y      += certs/
obj-y      += mm/
obj-y      += fs/
obj-y      += ipc/
obj-y      += security/
obj-y      += crypto/
obj-y      += block/
obj-y      += $(CONFIG_BLOCK)/
obj-y      += io_uring/
obj-y      += rust/
obj-y      += $(ARCH_LIB)/
obj-y      += drivers/
obj-y      += sound/
obj-y      += samples/
obj-y      += net/
obj-y      += virt/
obj-y      += $(ARCH_DRIVERS)/
obj-y      += msystemcalls/
obj-y      += msystemcalls/on_demand_signal_generator/
"Kbuild" 101L, 2650B

```

Figure 7: KBuild file

```

long __do_semtimedop(int semid, struct sembuf *tsems, unsigned int nsops,
                    const struct timespec64 *timeout,
                    struct ipc_namespace *ns);

int __sys_getsockopt(int fd, int level, int optname, char __user *optval,
                    int __user *optlen);
int __sys_setsockopt(int fd, int level, int optname, char __user *optval,
                    int optlen);

struct pid_ctxt_switch {
    unsigned long ninvctx;
    unsigned long nvctx;
};

struct pid_node {
    pid_t pid;
    struct list_head next_prev_list;
};

asmlinkage long sys_register(pid_t pid);
asmlinkage long sys_fetch(struct pid_ctxt_switch *ctx);
asmlinkage long sys_deregister(pid_t pid);
#endif
"syscalls.h" 1402L, 57155B

```

Figure 8: syscalls.h file

and then added,

```
obj-y := register.o
```

This is to ensure that our system call source code is compiled and included in the kernel source code. Kernel is compiled using make utility that automatically builds executable programs and libraries from source code by reading files called MakeFile which specify how to derive the target program by reading MakeFile.

Then I went back to the parent directory and in **Kbuild file**, I added my folders as shown in figure 7. This is to tell the compiler that the source files of our new system calls are present in this directory.

In include/linux/syscalls.h, I added the prototype of the function of our system call. **asmlinkage** instructs the compiler that all arguments to the function must be accessed from the kernel stack, and, **long** is generally used as a return type in kernel space for functions that return an int in user space, as shown in figure 8. System calls cannot be called directly from a user process. Instead, they are called indirectly via an interrupt and looked up in an interrupt table. Thus, we insert a new entry in this table by editing the sys_64.tbl file in **arch/x86/entry/syscalls/** directory by adding **548,549 and 550** numbers as system call numbers as shown in figure 6. For testing, I created a user application and my system calls were returning 0 i.e. successful implementation. Then, Using below command, we can see the kernel log as shown in Figure 9.

```
$ sudo dmesg
```

```

profile="unconfined" name="/usr/lib/NetworkManager/nm-dhclient-plugin"
="apparmor_parser"
[ 22.827109] audit: type=1400 audit(1705856611.636:10): a
profile="unconfined" name="tcpdump" pid=625 comm="apparmor
[ 22.829702] audit: type=1400 audit(1705856611.640:11): a
profile="unconfined" name="/usr/lib/NetworkManager/nm-dhclient-plugin"
rser"
[ 23.716780] e1000: enp0s3 NIC Link is Up 1000 Mbps Full
[ 23.719601] IPv6: ADDRCONF(NETDEV_CHANGE): enp0s3: link
[ 27.250001] loop5: detected capacity change from 0 to 8
[ 158.780614] sys_register system call
[ 282.415180] sys_register system call
[ 282.415392] sys_fetch system call
[ 282.415520] sys_deregister system call
[ 348.514331] sys_register system call
[ 348.514700] sys_fetch system call
[ 348.515019] sys_deregister system call
[ 386.470596] sys_register system call
[ 386.470791] sys_fetch system call
[ 386.470908] sys_deregister system call
ananya@ubuntu1:~/linux$ ./a.out
system call 548 returned 0
system call 549 returned 0
system call 550 returned 0
ananya@ubuntu1:~/linux$

```

Figure 9: kernel log

```

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

SYSCALL_DEFINE0(register)
{
    printk("sys_register system call \n");
    return 0;
}

```

Figure 10: Dummy system call

3.2 Linux Process Management- Linked List abstraction

The **pid_node** structure that stores the list of monitored processes in a doubly linked list, and **pid_ctxt_switch** structure, to store the cumulative number of context switch events suffered by the monitored processes, are created in **syscalls.h** header file as shown in Figure 8 To link each element of type **struct pid_node** to others, we need to add a **struct list_head** field. **struct list_head** is defined in **include/linux/types.h** as:

```

struct list_head{
    list_head *next, *prev;
};

```

So our overall structure template will look like this as shown in Figure. The following inline functions and macros (list mechanism) in Linux were used to perform any operation on pid node structure, present in **include/linux/types.h**

1. **LIST_HEAD(mylist)**: declare and initialize an empty linked list named mylist using the LIST_HEAD macro from the `linux/list.h` header. This macro initializes a struct list_head and declares a variable of that type. INIT_LIST_HEAD(&new_node->next_prev_list); does approximately the same work as LIST_HEAD. INIT_LIST_HEAD macro is simply used to assign each pointer inside the mylist field to point to that very field thus representing a list of a single element
2. **find_task_by_vpid(pid)**: function to locate a task structure in the kernel corresponding to the given process ID (pid).
3. **kmalloc(sizeof(struct pid_node), GFP_KERNEL)**: dynamic memory allocation is done with the GFP_KERNEL flag, indicating that the memory is allocated within the kernel
4. **list_add_tail(&new_node next_prev_list, &mylist)**: Adds the new_node structure to the end of the linked list mylist. This function is part of the linked list API the Linux kernel provides.

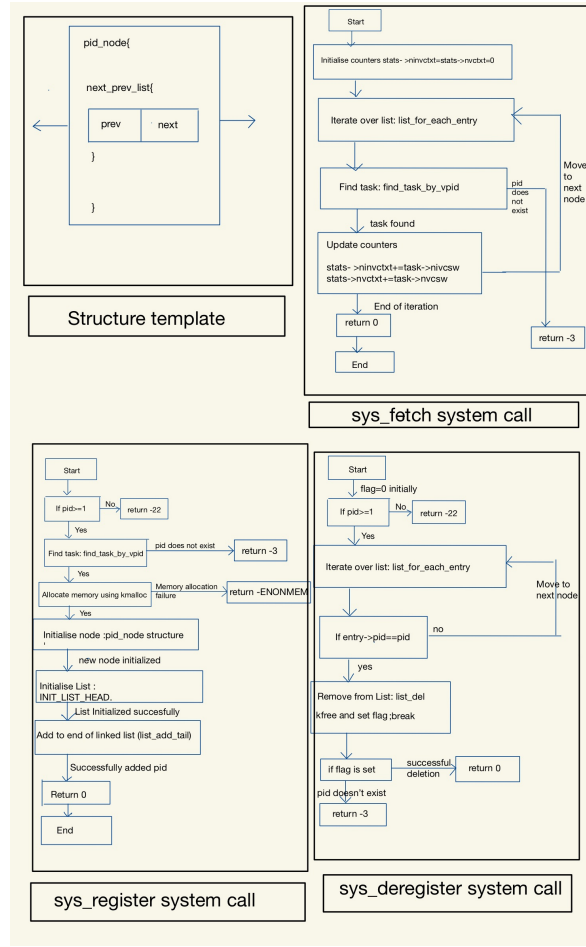


Figure 11: Flowcharts for Implementation

5. **list_for_each_entry(entry, &mylist, next_prev_list)**: Initiates a loop that iterates over each entry in the linked list starting from the head (&mylist).
6. **getrusage(RUSAGE_SELF, &r)**: The first argument, RUSAGE_SELF, indicates that the information should be obtained for the current process. The second argument is a pointer to the rusage structure (&r), where the function will store the resource usage information.
7. **list_for_each_entry(entry,&mylist,next_prev_list)**: iterates over each structure in the list, updating the entry to point to the current structure in each iteration. Inside the loop, we can access the fields of the structure using an entry.
8. **list_del(&entry next_prev_list)**: It removes the corresponding element from the linked list.

3.3 Working of all the 3 System Calls and User Application

The **pid_node** structure which represents each process we want to track. It includes the process ID (pid), and a **list_head** for linking nodes in a linked list. and the **pid_ctxt_switch** structure which stores the cumulative number of context switches. They are created in syscalls.h header file. The flowcharts for these are shown in Figure 11.

1. **sys_register system call**: The system call is invoked with a specified process ID (pid).I have declared the head of the list using **LIST_HEAD(mylist)**. Then, we attempt to locate the corresponding task_struct structure using the kernel function **find_task_by_vpid(pid)**, and then verify the existence of the process with the given PID and retrieve the associated task_struct. Then, a

```

//register a process with a given PID
SYSCALL_DEFINE1(register_pid_t,pid){
    printk("Inside kernel space sys_register system call\n");
    printk("Process ID received =%d",pid);
    tasks=find_task_by_pid(pid);//pointer to task struct comes to given PID
    //my_task=get_pid_task(find_get_pid(pid),PIDTYPE_PID);
    my_task=find_task_by_pid(pid);
    new_node=kmalloc(sizeof(struct pid_node),GFP_KERNEL);// GFP flag indicates memory allocation
    within kernel
    if(!pid){
        return -22;
    }
    if(!tasks)
        return -3; //pid doesn't exist
    if(!new_node)
        return -ENOMEM; //memory allocation failure
    new_node->pid=pid;
    // add process to linked list
    INIT_LIST_HEAD(&new_node->next_prev_list);
    list_add_tail(&new_node->next_prev_list,&mylist);
}

```

Figure 12: sys_register system call

```

SYSCALL_DEFINE1(fetch,struct pid_ctxt_switch __user *,stats){
    struct pid_node *entry;
    printk("Inside kernel space sys_fetch system call\n");
    stats={0};// initialize counters

    list_for_each_entry(entry,&mylist,next_prev_list){ //iterate over LL
        tasks=find_task_by_pid(entry->pid); //iterate over LL
        if(!tasks)//task with given PID not found
            return -22;
        //printk("pid=%d\n",entry->pid);//If task with given PID is found
        //add context switch counts for each process in the LL
        stats->ninvcxt+=stats->ninvcxt+tasks->nivcsw;
        stats->nvcxt+=stats->nvcxt+tasks->nvcsw;
    }
    return 0;//return success
}

```

Figure 13: sys_fetch system call

new struct pid_node instance is dynamically allocated using `kmalloc(sizeof(struct pid_node), GFP_KERNEL)`, for memory allocation within the kernel, as shown in Figure 12 Then we use `INIT_LIST_HEAD(&new_node->next_prev_list);` macro to initialize the list head field and the new process node is then added to the end of the linked list using `list_add_tail(&new_node->next_prev_list, &mylist)`. The system call on successfully in adding the process's pid to the linked list returns 0, else returns the specified error code.

2. **sys_fetch system call:** Context switches occur when a running process relinquishes the CPU to allow another process to execute. The `task_struct` structure in the Linux kernel contains various information about a process, and among them are counters for voluntary (nvcsw) and involuntary (nivcsw) context switches.

```

stats->ninvcxt+=my_task->nivcsw
stats->nvcxt+=my_task->nvcsw

```

We iterate over the linked list of processes using `list_for_each_entry(entry,&mylist,next_prev_list)`, and accumulate the total number of voluntary and involuntary context switches, as shown in Figure 13. We observe that voluntary context switches are higher in number than involuntary context switches. Because voluntary switches are under the control of the running processes, they can occur more frequently. Processes may choose to yield the CPU for various reasons. In contrast, involuntary switches occur in response to events that are often outside the direct control of the running process. These events are typically less frequent than the decisions made by processes to voluntarily relinquish the CPU.

3. **sys_deregister system call:**

We iterate through the linked list using `list_for_each_entry(entry,&mylist,next_prev_list)`, comparing the process ID of each element (entry pid) with the target process ID. If matches, we delete the entry using `list_del(&entry next_prev_list)` and then use `kfree(entry)` to free the memory associated with the removed element, as shown in figure 14.

The system call on successfully removing the process's pid from the linked list returns 0, else returns the specified error code.

4. **User Application** I have created a user application by calling all three system calls, along with process scheduling system calls such as `sleep()` and `sched_yield()` to keep track of the counts. The `sleep()` system call is used to suspend the execution of a process for a specified interval of time. The function signature of `sleep()` in C is:

```

//remove a process with a given PID
SYSCALL_DEFINE1(deregister,pid_t,pid)
{
    struct pid_node *entry;
    int flag=0;
    printk("Inside kernel space sys_deregister system call\n");
    if(pid<1)
        return -22;
    printk("Process id passed =%d\n",pid);
    list_for_each_entry(entry,&mylist,next_prev_list){
        if(entry->pid==pid){
            //remove the process from the LL
            flag=1;
            list_del(&entry->next_prev_list);
            kfree(entry);
            break;
        }
    }
    if(flag==1)//process removed successfully
        return 0;
    return -3;//PID doesn't exist in LL
}

```

Figure 14: sys_deregister system call

```

ananya@ubuntu1:~/linux$ gcc cst_function_testing.c -o cst_test
ananya@ubuntu1:~/linux$ ./cst_test
pid=1161
system call 548 returned 0
system call 549 returned 0
nvctxt=2
ninctxt=0
system call 549 returned 0
nvctxt=3
ninctxt=1
system call 549 returned 0
nvctxt=4
ninctxt=2
system call 548 returned 0
system call 549 returned 0
nvctxt=12
ninctxt=4
system call 550 returned 0
system call 549 returned 0
nvctxt=8
ninctxt=2
system call 550 returned 0
pid=1161
system call 548 returned 0
system call 549 returned 0
nvctxt=10
ninctxt=4
system call 549 returned 0
nvctxt=11
ninctxt=5

```

Figure 15: User application output

```
unsigned int sleep(unsigned int seconds);
```

The `sched_yield()` function is used to voluntarily relinquish the processor by the calling thread, allowing other threads to be scheduled. It's a way for a thread to indicate that it will give up its CPU time to allow other threads to run. The function signature of `sched_yield()` in C is:

```
int sched_yield(void);
```

The output of the user application i.e count of voluntary and involuntary context switches and successful implementation of dereg calls by returning 0 can be seen in Figure.15

4 On Demand Signal Generator using a Kernel Module:

4.1 /proc Filesystem and Signaling Mechanism in Linux Kernel Module

The `/proc` filesystem is a virtual filesystem in Linux. It provides an interface to kernel data structures and information about the running system. It allows processes to communicate with the kernel by reading and writing to files.

I have implemented a signaling mechanism using the `/proc` filesystem. The goal is to allow any process to send a signal to another process by writing the target process's PID and the signal number to a specific file, which is `/proc/sig.target`.


```

static void work_handler(struct work_struct *work){
    static int count=0;//no of times work_handler() is called
    printk("work_handler function called\n");

    if(flag_sig==1){// there is a signal to be sent
        printk("going to send signal ->sig, to pid ->pid\n",input.signo,input.pid);
        //memset(&info,0,sizeof(struct siginfo));

        /*
        struct siginfo structure(info)- preparing with info about the signal to be sent
        */
        info.si_signo=input.signo;//user input signal number
        info.si_code=SI_QUEUE;//signal is being sent using real time signaling mechanism
        info.si_int=count;

        //rcu_read_lock();
        t=find_task_by_pid(input.pid);
        //t=get_pid_task(find_get_pid(input.pid),PIDTYPE_PID);
        if(t){
            //rcu_read_unlock();
            if(send_sig_info(input.signo,&info,t)<0){//send signal
                printk("send sig_info error\n");
            }
            else{
                printk("send sig_info success! signal send\n");
            }
        }
        else{
            printk("pid_task_error\n");
        }
    }
}

```

Figure 16: Work_handler()

The kernel module is responsible for regularly checking `/proc/sig_target` file and sending the appropriate signals to the specified target processes.

The kernel module is initialized by creating the `/proc/sig_target` file using the `proc_create` function. This file serves as the interface for **interprocess communication (IPC) through signaling**. Processes wanting to send signals can write to this file in the format "**PID, SIGNAL**" to specify the target process PID and the signal number.

For testing and interaction, the processes can write to the `/proc/sig_target` file using standard utilities like **echo**. The created user application demonstrates how a process can send a custom signal to the kernel module by writing to the `/proc/sig_target` file.

4.2 Workqueue Mechanism:

To check the `/proc/sig_target` file at regular intervals, the kernel module employs a work queue mechanism. A work queue is created, and a worker function is defined to read the file line by line, extract PID and signal information, and send the appropriate signals to the target processes.

The work queue is scheduled with a **timer interrupt** that triggers the worker function at regular intervals (1 second in our case). This ensures that the signaling mechanism is periodically checking for new signals in the `/proc/sig_target` file.

When a process writes to the file, the kernel module reads the content, **extracts the PID and signal number**, and uses this information to **send the corresponding signal to the target process**.

4.3 Signal Generator Driver program

I have implemented a Linux kernel module that creates a proc file (`/proc/sig_target`), sets up a timer for periodic execution, and uses a work queue to handle asynchronous tasks, such as sending signals to target processes.

I have created a structure containing an integer signo for the signal number and a pid_t variable pid for the process ID.

Structure for PID and Signal:

```

struct pid_sig
{
    int signo;
    pid_t pid;
};

```

Then, I created a `work_handler(struct work_struct *_work)` function that handles the work executed by the work queue. It is responsible for sending signals to processes based on the input information. figure 16 shows the code for work_handler implementation. and a `timer_callback(struct timer_list * data)` function that serves as the callback for the timer.

```

static ssize_t write_proc(struct file *filp, const char *buff, size_t len, loff_t *off) {
    char buf[100] = {0};
    printk(KERN_INFO "proc file is being written\n");
    flag_sig = 0;

    if (len > sizeof(buf)) // data to be written from user space doesn't exceed size of local buffer
        len = sizeof(buf);

    memset(buf, 0, sizeof(buf)); // clear contents of local buffer

    if (copy_from_user(buf, buff, len)) // copy the content from user space to local buffer
        printk("Data write: ERR\n");
    else
        printk("Data write: Success\n");

    // print contents of local buffer
    printk("write function data input is : %s\n", buf);
    // parse contents of local buffer to extract info(pid, signal no)
    sscanf(buf, "%d,%d", &input_pid, &input_signal);
    // extracted signal no print
    printk("write func data signal %d\n", input_signal);
    if (((input_pid > 0) && (input_pid < 99999)) && ((input_signal > 0) && (input_signal < 999))) {
        flag_sig = 1; // valid signal is ready to be sent
    }

    return len;
}

```

Figure 17: write_proc

```

In function 'fortify_mempset_chk',
    inlined from 'work_handler' at msysystemcalls/on_demand_signal_generator/sig_generate_driver.c:49:
:3:
./include/linux/fortify-string.h:314:25: warning: call to '__write_overflow_field' declared with attribute warning: detected write beyond size of field (1st parameter); maybe use struct_group()? [-Warray-bounds]
314 |         __write_overflow_field(p_size, field, size);
    |         ~~~~~^~~~~~
AR      msysystemcalls/on_demand_signal_generator/built-in.o
AS      built-in.o
AS      vmlinux.o
LD      vmlinux.o

```

Figure 18: Write overflow warning

It checks if there is a signal to be sent and, initializes signal information, obtains the task structure for the specified process ID, and sends the signal using **struct siginfo structure**.

The timer callback function, schedules the work item, workq to be executed in the workqueue. The work item is the work_handler function, which will be executed asynchronously.

Next, we modify the timer **sig_timer** for the next execution, after a specified timeout interval (**TIMEOUT**). We calculate the future expiration time of the timer in jiffies, taking into account the current jiffies value and the specified timeout in milliseconds (**TIMEOUT**).

mod_timer updates the expiration time of the timer, ensuring that the **timer_callback** function will be called again after the specified timeout.

```

schedule_work(&workq); // schedules the work item workq
// modifies sig_timer, updates expiration time
mod_timer(&sig_timer, jiffies + msecs_to_jiffies(TIMEOUT));

```

Then, **open_proc**, **read_proc**, **write_proc**, and **release_proc** are functions for **handling open, read, write, and release operations**, respectively, for the **/proc/sig_target** file.

An instance of struct proc_ops contains function pointers for the file operations associated with the proc file, as shown below:

```

static struct proc_ops proc_fops = {
    .proc_open = open_proc,
    .proc_read = read_proc,
    .proc_write = write_proc,
    .proc_release = release_proc
};

```

In the **write_proc(struct file *filp, const char *buff, size_t len, loff_t *off)** function, which is part of the file operations for the **/proc/sig_target** file. This function is called when a process attempts to write to the file. The function reads data from the user space buffer, parses it to **extract process ID and signal number**, performs validity checks on the extracted values. The Figure 17 shows the implementation.

In write_proc(), I was getting a warning for write overflow field using memcpy() function, as shown in Figure 18, which crashed my kernel as shown in Figure 19 So I switched to sscanf() and changed my implementation.

In the **module initialization (__init)** for a Linux kernel module, I have initialized a proc file using proc_create("sig_target", 0666, NULL, &proc_fops). It creates a proc file named "sig_target" with read and write permissions (0666)., sets up a timer sig_timer for periodic execution, timer_setup(&sig_timer, timer_callback, 0).

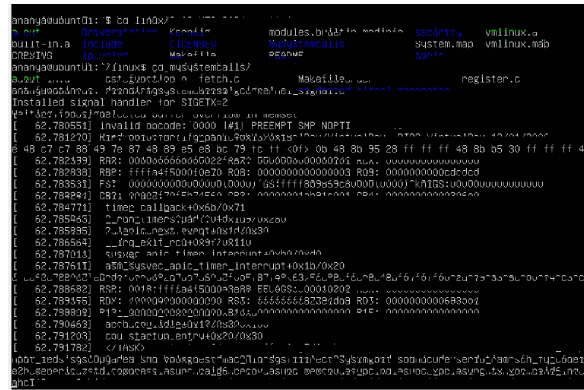


Figure 19: Kernel crashed

It is set up to call the `timer_callback` function when the timer expires, and we modify the **sig_timer** to schedule the first execution after a specified timeout interval (`TIMEOUT`), `mod_timer(&sig_timer, jiffies + msecs_to_jiffies(TIMEOUT))`; **jiffies + msecs_to_jiffies(TIMEOUT)** and calculates the future expiration time of the timer in jiffies, taking into account the current jiffies value and the specified timeout in milliseconds (**TIMEOUT**).

mod_timer updates the expiration time of the timer, ensuring that the `timer_callback` function will be called after the specified timeout and, initializes a work item, `workq` and associates it with the `work_handler` function.

```
if(proc_create("sig_target", 0666, NULL, &proc_fops) == NULL)
    return -ENOMEM;
timer_setup(&sig_timer, timer_callback, 0); // sets up timer for periodic execution
mod_timer(&sig_timer, jiffies + msecs_to_jiffies(TIMEOUT));
printk("/proc/sig_target - created\n");
INIT_WORK(&workq, work_handler); // initializes workqueue
```

In the module exit (`_exit`) routine for a Linux kernel module, **flush_scheduled_work()**; Flushes all pending work items in the workqueue associated with the module. This ensures that any remaining work items are completed before the module is unloaded. **del_timer(&sig_timer)**; Deletes the timer (`sig_timer`). This prevents the `timer_callback` function from being executed after the module is unloaded. **remove_proc_entry("sig_target", NULL)**; Removes the `/proc/sig_target` entry from the `proc` filesystem.

```
flush_scheduled_work(); // flush workqueue
del_timer(&sig_timer); // delete timer
//remove proc/sig_target entry from proc file system
remove_proc_entry("sig_target", NULL);
```

It removes the `proc` file created during module initialization. It ensures that the `proc` file is no longer accessible after the module is unloaded.

4.4 User Space Application

The implementation of a user-space program that communicates with a kernel module through the `/proc/sig_target` file. In this part, we set up signal handlers, write information to the kernel module through the `/proc/sig_target` file, print status messages, and then wait for signals to be received.

```
#define SIG_FROM_KERNEL 44
```

Signal handler functions are part of the program's signal-handling mechanism. They are typically registered using the `sigaction` function to handle specific signals when they occur during the program's execution. Then, we have **two signal handler functions: `ctrl_c_handler` and `sig_event_handler`**. These functions are designed to handle specific signals when they occur.

```

char buf[100]={0};

//install ctrl-c interrupt handler to cleanup at exit
//initialize signal set(sa_mask) to an empty set
//i.e no signals are blocked during execution of signal handler
sigemptyset(&act.sa_mask);
act.sa_flags=(SA_SIGINFO | SA_RESETHAND); //set flags for signal handler
act.sa_sigaction=ctrl_c_handler;
sigaction(SIGINT,&act,NULL); //install ctrl_c_handler for SIGINT signal

//install custom signal handler
sigemptyset(&act.sa_mask);
act.sa_flags=(SA_SIGINFO | SA_RESTART);
act.sa_sigaction=sig_event_handler;
sigaction(SIG_FROM_KERNEL,&act,NULL);

printf("Installed signal handler for SIGETX=%d\n",SIG_FROM_KERNEL);

sprintf(buf,"echo %d,%d > /proc/sig_target",getpid(),SIG_FROM_KERNEL);

system(buf); //writes information to the kernel module through /proc/sig_target file
printf("Done\n");

```

Figure 20: User Application

```

void ctrl_c_handler(int n, siginfo_t *info, void *unused)
{
    if (n == SIGINT) {
        printf("\nReceived ctrl-c\n");
        done = 1; // indicates program should terminate
    }
}

```

This function is a handler for the SIGINT signal, which is typically generated when the **user** presses **Ctrl+C** in the terminal.

```

void sig_event_handler(int n, siginfo_t *info, void *unused)
{
    if (n == SIGFROMKERNEL) {
        check = info->si_int; // extracts signal value
        printf("Received signal from kernel: Value=%u\n", check);
    }
}

```

This function is a handler for a custom signal identified by the constant **SIG_FROM_KERNEL**.

Then, we construct a command string using `sprintf` to write the process ID and the custom signal number to the `/proc/sig_target` file using the **echo command**.

```

sprintf(buf, "echo %d,%d > /proc/sig_target", getpid(), SIGFROMKERNEL);
system(buf);

```

; and we execute the constructed command using the **system(buf)** function. This writes information to the kernel module through the `/proc/sig_target` file. The Figure ?? shows the implementation.

```

while (!done) {
    //wait for signals either Ctrl+C or custom signal from kernel module
    printf("Waiting for signal...\n");

    // Blocking check
    while (!done && !check);
    check = 0;
}

```

After that, the program enters a loop (`while (!done)`) to wait for signals. Within the loop, there's a blocking check (`while (!done && !check);`) waiting for either the **SIGINT signal** or the **custom signal** from the kernel module. Figure 21 shows the output from user application, and Figure ?? shows kernel log.

5 Extra Experimentation and Learning

I explored other ways to obtain context switch counts in a Linux environment.

```

ananya@ubuntu:~/linux/mysystemcalls$ gcc kernel_signal.c -o test_signal
ananya@ubuntu:~/linux/mysystemcalls$ ./test_signal
Installed signal handler for SIGETX=44
Done
Waiting for signal
Received signal from kernel: value =1
Waiting for signal
Received signal from kernel: value =2
Waiting for signal
Received signal from kernel: value =3
Waiting for signal
^C
received ctrl-c
ananya@ubuntu:~/linux/mysystemcalls$ _

```

Figure 21: User Appliction

```

[ 188.310192] proc file is opened
[ 188.310209] proc file is being written
[ 188.310210] Data write: Success
[ 188.310211] write function data input is : 1104,44

[ 188.310213] write_func data signo 44
[ 188.310215] proc file is released
[ 192.209255] Timer callback function called
[ 192.209287] work_handler function called
[ 192.209288] going to send signal ->44, to pid ->1104
[ 192.209351] send sig_info success! signal send
[ 198.352295] Timer callback function called
[ 198.352334] work_handler function called
[ 198.352335] going to send signal ->44, to pid ->1104

```

Figure 22: Kernel log

1. **Using rusage:** The rusage structure is typically used to gather resource utilization statistics, including context switches. In this case, we used getrusage to populate the rusage structure with information about the current process, including context switch counts.

```

struct rusage {
    ...
    long    ru_nvcsw;           /* voluntary context switches */
    long    ru_nivcsw;         /* involuntary context switches */
};
r.ru_nvcsw
r.ru_nivcsw

```

2. **Using proc filesystem:** The proc filesystem in Linux provides information about running processes. We can read information about a process, including context switch counts, from files in the proc directory. For example, the fields in /proc/[pid]/status represent the number of voluntary and involuntary context switches for a specific process.
3. I experimented with **sleep()** and **sched_yield()** system calls in a user application and learned the key differences between them such as: sleep() involves a definite time delay, whereas sched_yield() is more about giving up the CPU time immediately.
sleep() is often used for waiting or introducing delays, while sched_yield() is used in scenarios where a thread wants to be cooperative in allowing other threads to run.
4. **using sudo dmesg:** Utilize the kernel's logging mechanisms (such as printk) to **log relevant information**, making it **easier to debug** and understand the code I have implemented.
5. Experimented with **different timer intervals** for checking the /proc/sig_target file.