COL331 OS Homework Assignment 1 - Report

Hard Track

Aryan Dua - 2020CS50475

 # Steps done from scratch:

VM setup:

- sudo apt-get install cpu-checker
- sudo kvm-ok
- sudo apt install -y qemu qemu-kvm libvirt-daemon libvirt-clients bridge-utils virt-manager
- sudo usermod -G libvirt -a $USER
- sudo systemctl status libvirtd
- virt-manager

Building linux kernel:

- sudo apt-get install git fakeroot build-essential ncurses-dev xz-utils libssl-dev bc flex libelf-dev bison
- git clone —depth 1 —branch v6.1.6 git://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git
- cd linux
- git checkout v6.1.6

Compiling and running linux:

- cp -v /boot/config-$(uname -r) .config
  # Open .config file and set the CONFIG_SYSTEM_TRUSTED_KEYS and the other option which looks quite similar in name and in the value to ""(empty string)
- make -j4
- sudo make modules_install -j4
- sudo make install -j4
  #Open the grub configuration file at /etc/default/grub set the GRUB_TIMEOUT to 60 and comment the GRUB_TIMEOUT_STYLE option.
- sudo update-grub

  Reboot the system

1. **Implementation of the Context switch Tracker**

I have created a few system calls that can track the total number of voluntary and involuntary context switches for a group of monitored processes being maintained in a doubly linked list.

1. int **sys_register**(pid_t pid) - This call adds a process to the list of monitored processes.

   Error-handling -
   - **Return -3** If the process does not exist
   - **Return -22** If the process PID is less than 1
   - **Return 0** If the system call was successful

2. int **sys_fetch**(struct pid_ctxt_switch *stats) - Finds the sum total of voluntary and involuntary context switches for all the processes in the doubly linked list of the processes monitored. It sends the result back in the *stats* struct.

   Error-Handling -
   Return -22 if:
   - The pointer given by the user is a null pointer
   - The list entry points to a null pointer
   - The corresponding task struct is a null pointer

3. int **sys_deregister**(pid_t pid) - Removes the process from the list of monitored processes.

   Error-Handling -
   - **Return -3**, If the process does not exist
   - **Return -22**, If the process PID is less than 1
   - **Return 0**, if the system call was successful

Here is how I added a system call:

Adding a new system call(Helloworld system call):

1. Open arch/x86/entry/syscalls/**syscall_32.tbl**. Append like: 451 i386 hello sys_hello
2. Open arch/x86/entry/syscalls/**syscall_64.tbl**. Append like: 451 common hello sys_hello
3. Open include/linux/**syscalls.h**. Append at EOF just before endif. asmlinkage long sys_hello(void);
4. Open kernel/**sys_ni.c**. Add COND_SYSCALL(hello) anywhere u feel like
5. Open kernel/**sys.c**. Add SYSCALL_DEFINE0(hello){} and write your function
6. Open include/uapi/asm-generic/**unistd.h**. Add
   #define __NR_hello 451
   __SYSCALL(__NR_hello, sys_hello)

In my kernel, here are the system call numbers for all these system calls.

sys_register- 452
sys_fetch - 453
sys_deregister - 454

```
aryan@aryan:~/mod$ cd ..
aryan@aryan:~$ gcc userspace.c
aryan@aryan:~$ ./a.out 452 8
System call sys_register returned 0, added pid 8 (only if return value = 0)
aryan@aryan:~$ ./a.out 453
Value of vol = 4
Value of inv = 0
System call sys_fetch returned 0
aryan@aryan:~$ ./a.out 454 8
System call sys_deregister returned 0, removed pid 8 (only if return value = 0)
aryan@aryan:~$ ./a.out 453
Value of vol = 0
Value of inv = 0
System call sys_fetch returned 0
aryan@aryan:~$
```

Explanation:

1. Adding a process with PID = 8 to the linked list of monitored processes through system call number 452.
2. Checking the total number of voluntary and involuntary context switches in the list through system call number 453. (Data will be shown for process pid = 8)
3. Removing the process with PID = 8 from the linked list of monitored processes through system call number 454.
4. Checking the total number of voluntary and involuntary context switches of the empty list.

```c
#include <stdlib.h>

int main(int argc, char *argv[])
{
        struct pid_ctxt_switch
        {
                unsigned long ninvctxt;
                unsigned long nvctxt;
        };

        int code = atoi(argv[1]);
        if(code==452)
        {
                int pid = atoi(argv[2]);
                long int call = syscall(code, pid);
                printf("System call sys_register returned %ld, added pid %d (only if return value = 0)\n", call, pid);
        }
        else if(code==453)
        {
                struct pid_ctxt_switch *stats;
                stats = malloc(16*sizeof(struct pid_ctxt_switch *));
                long int call = syscall(code, stats);
                printf("Value of vol = %lu\n", stats->nvctxt);
                printf("Value of inv = %lu\n", stats->ninvctxt);
                printf("System call sys_fetch returned %ld\n", call);
        }
        else if(code==454)
        {
                int pid = atoi(argv[2]);
                long int call = syscall(code, pid);
                printf("System call sys_deregister returned %ld, removed pid %d (only if return value = 0)\n", call, pid);
        }


        return 0;
}
```

Here is the userspace program that I made to test the system calls.

2. **How exactly does my Linux Module Work?**

The requirement was to first create a /proc/sig_target file when the module is initialized. Modules are different from normal programs due to the fact that the kernel does not need to be compiled again and again, the module can just be loaded and removed.

Anyways, the processes which need to send signals to other processes will then use this file and enter entries in the form pid, signal_id. The delimiter I have considered is a comma followed by a space.

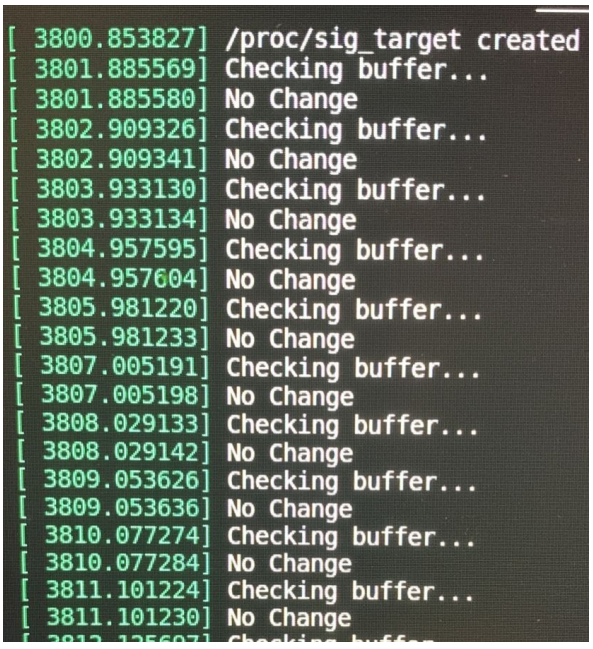
Then, my kernel module will regularly read the /proc/sig_target file at an interval of 1 second and send the required signals to the required processes.

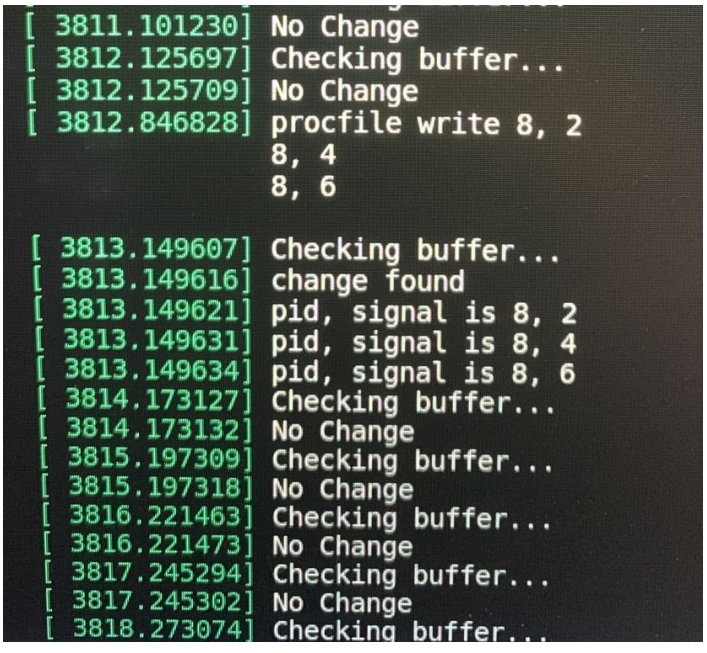
This is done by creating a second thread during the init of the module, when the /proc/sig_target file is created. This new thread runs a loop till this thread is killed, which has a sleep(1) command in every iteration, to ensure that the loop commands are run at an interval of 1 second. The contents of the loop are securely maintained using a spinlock, which I have described in the extras section.

In this loop, I first do string matching to extract the pid number and the signal number from the character buffer. It can contain multiple lines, so the loop has to send multiple signals in that case.

After I've extracted the pid_number and the sig_number from the buffer, I have to convert it to an integer data type. This is done by a custom atoi() function that I have defined in my module. After converting to the required data types, I find the corresponding pid structure and then the corresponding task struct. Finally, I send the required signal using the send_sig() function, in the privileged mode.

Here is a screenshot of the working of my Linux Module:





- **Explanation of the working:**

    1. The kernel module initializes and the /proc/sig_target file is created.
    2. The new thread starts and the while loop starts running.
    3. Since there is no new write to the file, there is no need to send any signal to any process.
    4. When the module finds that a write has taken place, I change the value of a variable to mark that a new write has been made.
    5. Every second when the buffer is checked, if this flag variable is ‘set’, then I extract the pid and signal number for each entry from the buffer and send the required signals through the process mentioned above. The ‘change’ mechanism has been made to avoid repetition of signal sending.
    6. Afterwards, since there is no new write to the buffer, no new signals are being sent.

- **Instructions to run:**
    1. make
    2. make(running make the first time gives a frame size warning, however, running it again, removes the warning, purely an observation)
    3. sudo insmod final.c (loading the module)
    4. sudo echo -e “8, 2\n8, 4\n8, 6” > /proc/sig_target (writes to the buffer)
    5. sudo dmesg (shows the log)
    6. sudo rmmod final.c (unloads the module)

- **How I made my patch file:**
    1. Changed the directory names of my changed folder to linux-change and the original folder to linux-base.
    2. Ran the command: sudo diff -rupN linux-change/ linux-base/ > ctxttrack.patch

- **Contents of the submission:**
    1. **report.pdf -** this report
    2. **final.c** - Module c code
    3. **Makefile** - Module Makefile
    4. **ctxttrack.patch** - the patch file for the context switch tracker

3. **Implementation Details**

   There was a choice in the way I handled the buffer in the linux module. The way I did it is as follows. I am setting a trigger variable which tells me whether there is a new write to the /proc/sig_target file. I check the file every second. If the trigger is on, then I copy the data from the buffer and process the data to send the required signals. Other ways to handle this is to just read and act according to the data present in the buffer whenever it is read every second. I did not opt for this method because it can cause repetition in the sending of signals to processes.

4. **Extras/Learnings**

   In the process of creating the system calls and the linux modules, I came across many interesting facts that I did not know earlier. They include:

   1. The function **copy_to_user()** and **copy_from_user()**. The kernel and user address spaces are different. If something corresponds to address number 0 in the kernel address space, then it is something else in the user address space. To provide a smooth bridge between the two, we have these 2 functions.

   2. I learnt about how pointers can be passed between different functions

   3. When finding certain functions, instead of going through the kernel source code manually, we can go through https://elixir.bootlin.com/linux/latest/source.

   4. The kernel has a different set of header files. When some common header file does not work, it will probably have its own library, or function. Like **kthread** instead of pthread, **kmalloc** instead of malloc

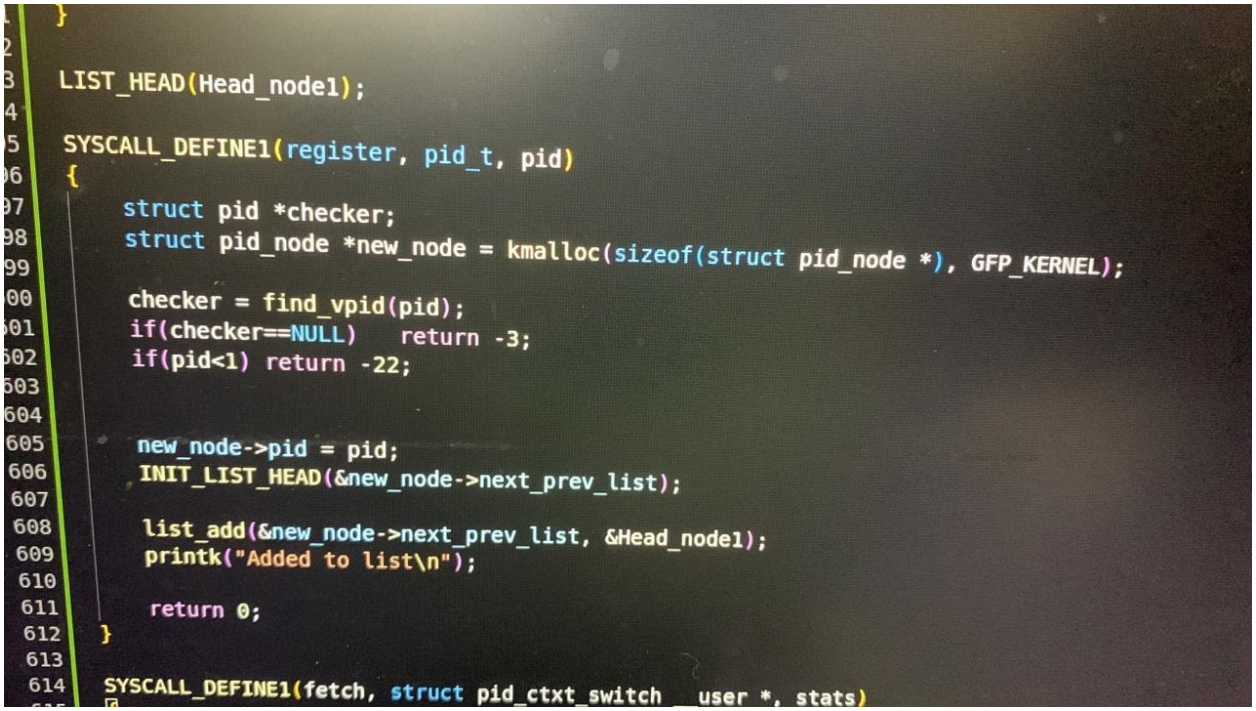
   5. In the linux module, I have used **spin locks**. Spin locks are a low-level synchronization mechanism suitable primarily for use on shared memory multiprocessors. When the calling thread requests a spin lock that is already held by another thread, the second thread spins in a loop to test if the lock has become available. When the lock is obtained, it should be held only for a short time, as the spinning wastes processor cycles.

   6. Learnt about how the 4-number coded file permissions work

   7. Even though VS code highlights “register” as a keyword, there is no issue in using it as a name for a system call.

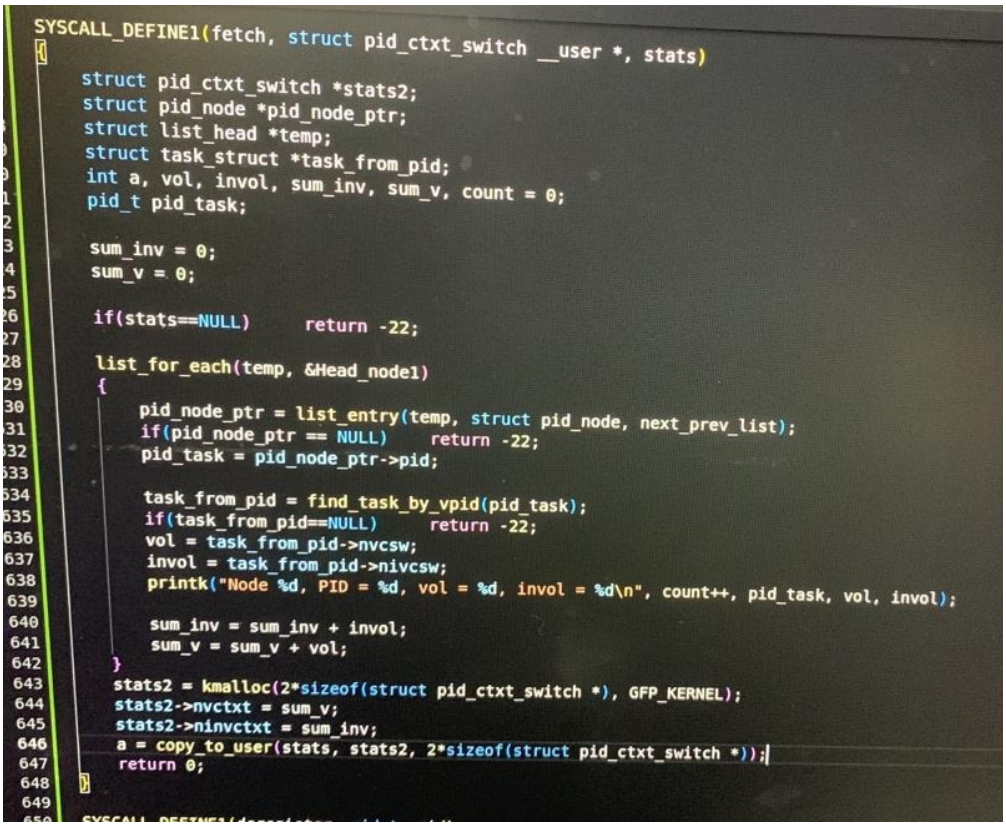
5. Code Snippets for the system calls in the context switch tracker

1. Defining the list head node as “Head_node1”, and the sys_register() system call.



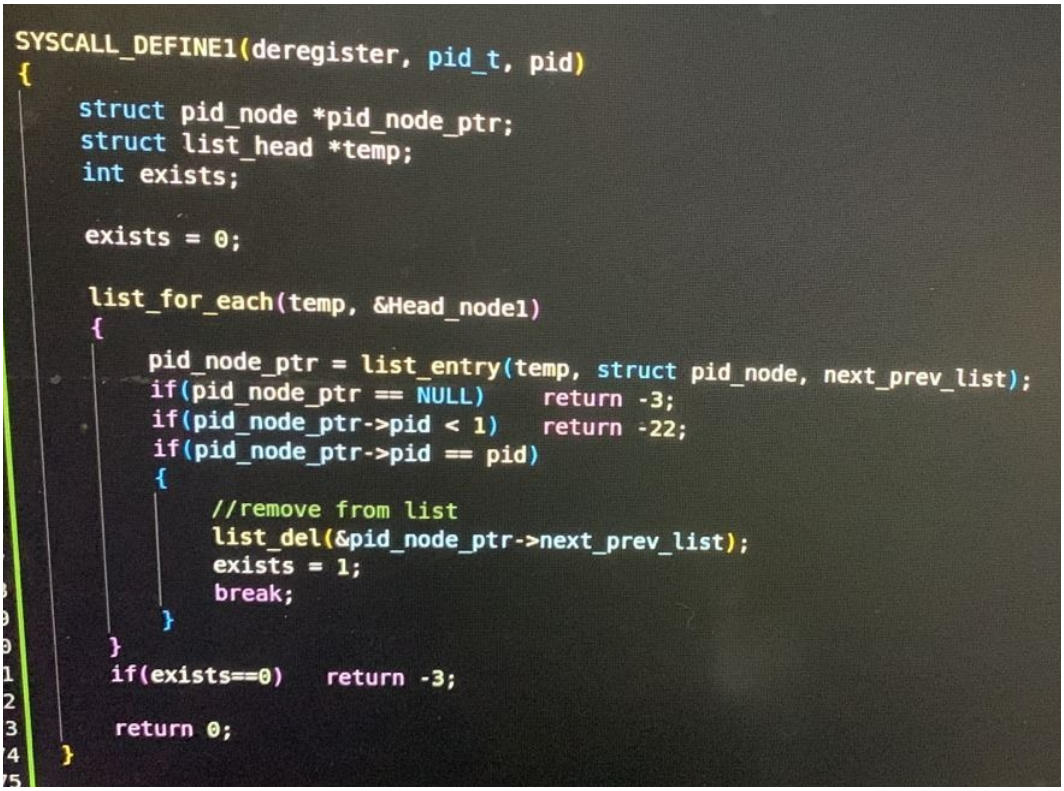
```
}

LIST_HEAD(Head_node1);

SYSCALL_DEFINE1(register, pid_t, pid)
{
    struct pid *checker;
    struct pid_node *new_node = kmalloc(sizeof(struct pid_node *), GFP_KERNEL);

    checker = find_vpid(pid);
    if(checker==NULL)   return -3;
    if(pid<1) return -22;


    new_node->pid = pid;
    INIT_LIST_HEAD(&new_node->next_prev_list);

    list_add(&new_node->next_prev_list, &Head_node1);
    printk("Added to list\n");

    return 0;
}

SYSCALL_DEFINE1(fetch, struct pid_ctxt_switch __user *, stats)
```

2. Defining the sys_fetch() system call.



```
SYSCALL_DEFINE1(fetch, struct pid_ctxt_switch __user *, stats)
{
    struct pid_ctxt_switch *stats2;
    struct pid_node *pid_node_ptr;
    struct list_head *temp;
    struct task_struct *task_from_pid;
    int a, vol, invol, sum_inv, sum_v, count = 0;
    pid_t pid_task;

    sum_inv = 0;
    sum_v = 0;

    if(stats==NULL)     return -22;

    list_for_each(temp, &Head_node1)
    {
        pid_node_ptr = list_entry(temp, struct pid_node, next_prev_list);
        if(pid_node_ptr == NULL)    return -22;
        pid_task = pid_node_ptr->pid;

        task_from_pid = find_task_by_vpid(pid_task);
        if(task_from_pid==NULL)     return -22;
        vol = task_from_pid->nvcsw;
        invol = task_from_pid->nivcsw;
        printk("Node %d, PID = %d, vol = %d, invol = %d\n", count++, pid_task, vol, invol);

        sum_inv = sum_inv + invol;
        sum_v = sum_v + vol;
    }
    stats2 = kmalloc(2*sizeof(struct pid_ctxt_switch *), GFP_KERNEL);
    stats2->nvctxt = sum_v;
    stats2->ninvctxt = sum_inv;
    a = copy_to_user(stats, stats2, 2*sizeof(struct pid_ctxt_switch *));
    return 0;
}
```

3. Defining the sys_deregister() system call



```
SYSCALL_DEFINE1(deregister, pid_t, pid)
{
    struct pid_node *pid_node_ptr;
    struct list_head *temp;
    int exists;

    exists = 0;

    list_for_each(temp, &Head_node1)
    {
        pid_node_ptr = list_entry(temp, struct pid_node, next_prev_list);
        if(pid_node_ptr == NULL)    return -3;
        if(pid_node_ptr->pid < 1)   return -22;
        if(pid_node_ptr->pid == pid)
        {
            //remove from list
            list_del(&pid_node_ptr->next_prev_list);
            exists = 1;
            break;
        }
    }
    if(exists==0)   return -3;

    return 0;
}
```