Context Switch Tracker and Signal Generator

Operating Systems Assignment 1

Dakshit Babbar | 2020EE30163

February 2024

1 Objective

Given the Linux source code, we have to implement some functionalities that will be added to the Linux Kernel. Following are the objectives:

- To modify the source code to make a Context Switch Tracker.
- To write an external module that will function as an On-Demand Signal Generator.

2 System Setup

The system setup is completed using the *Guide to Hacking the Linux Kernel* by Abhishek Safui. Following is the overview of the steps involved in the setup:

1. Install the source code of the *Yocto Project*(*poky*). This is an embedded Linux platform tool which helps us create a minimal configuration for our kernel to reduce the compilation and testing time.
2. Build *poky* to get the following:
   - Virtual Disk Image
   - Minimal .config file
3. Install the Linux source code from *kernel.org*
4. Build the Linux source code using the minimal .config file obtained from *poky*
5. Boot the compiled kernel on a VM with the Virtual Disk Image obtained from *poky*.

3 Context Switch Tracker

Here we were supposed to modify the Linux Source Code to find the total number of voluntary and involuntary context switch events suffered by a subset of processes currently running on the system (referred to as monitored processes).

3.1 Directory Structure

For implementing the Context Switch Tracker we have created a separate directory *cstracker* in the Linux root directory. This contains the file *cstracker.c* which implements the Context Switch Tracker in detail.

In order to allow the user to make use of this functionality we have also defined a header file *cstracker.h* at *include/uapi/linux* which exposes the **`struct pid_ctxt_switch`** to the user which she can use to fetch the number of voluntary and involuntary context switch events.

3.2 Implementation Details

Here we specify the details of the *cstracker.c* file which implements the Context Switch Tracker.

- **Global Scope:**
  We do some basic initialisations in the global scope of the file as shown below.

  ```
  //define the struct pid_node used as a wrapper over a node in the list

  //initialise the doubly linked list
  //initialise a spinlock to secure the list
  ```

- **System Call `sys_register`:**
  The user makes this syscall to register a particular process as a monitored process by providing its *pid*. This syscall will perform the checks on the input as shown in the pseudo code and then add the *pid* to the list in a new node.

  ```
  SYSCALL_DEFINE1(sys_register, pid_t, pid){

      //if pid<1 return -22;
      //if pid does not exist return -3;
      //if pid is already in the list return -22;

      //make a new node

      //aquire lock
      //add node to list
      //release lock

      return 0;
  }
  ```

- **System Call `sys_fetch`:**
  The user makes this syscall to fetch the number of voluntary and involuntary context switch events. The user passes a pointer to an object of the **`struct pid_ctxt_switch`** and this syscall will iterate through the list to obtain the number of voluntary and involuntary context switch events from the **`task_struct`** of the processes, add them up and put the final numbers in the user space struct. An important feature that this syscall also implements is that whenever this will be called it will delete all the nodes from the list which correspond to processes that are no longer running.

```
SYSCALL_DEFINE1(sys_fetch, struct pid_ctxt_switch*, stats){

    struct pid_ctxt_switch kstats={0,0};

    //aquire lock
    //for each node in list:
        //if node->pid still exists:
            //let task be the task_struct of the pid
            kstats.ninvctxt += task->nivcsw;
            kstats.nvctxt += task->nvcsw;
        //else delete the node
    //release lock

    //copy from kernel space kstats to user space stats
    return 0;
}
```

- **System Call `sys_deregister`:**
  The user makes this syscall to deregister a particular process as a monitored process by providing its *pid*. This syscall will perform the checks on the input as shown in the pseudo code and then delete the *pid* from the list if it exists.

```
SYSCALL_DEFINE1(sys_deregister, pid_t, pid){

    //if pid<1 return -22

    //aquire lock
    //for each node in the list:
        //if (node->pid==pid):
            //delete node
            //release lock
            return 0;
    //release lock

    return -3;
}
```

- **System Call `sys_cst_num_nodes`:**
  The user provides an integer pointer & the syscall copies the total number of nodes into it.

```
SYSCALL_DEFINE1(sys_cst_num_nodes, int*, x){

    int count=0;

    //aquire lock
    //for each node in list:
        count++;
    //release lock

    //copy from kernel space &count to user space x
    return 0;
}
```

3.3 Extra Work

Following are some of the implementations done in addition to the ones that were specified.

- Used locks to secure the list to avoid racing conditions
- If a process already exists in the list then the **`sys_register`** syscall will not add it again and will return -22.
- Whenever a **`sys_fetch`** syscall is made, it will delete all the invalid (the processes that are no longer running) nodes from the list while counting the number of voluntary and involuntary context switch events.
- Implemented another syscall **`sys_cst_num_nodes`** which counts the number of registered processes in the list. This helps us know the present state of the list.

4 Signal Generator

Here we were supposed to write an external kernel module that will function as an On-Demand Signal Generator. Any process that wants to fire a signal to another process (target) uses the proc filesystem to insert the PID of the target process and the signal number in the */proc/sig_target* file, which is created when the module is initialised. The kernel module must check this file at a regular interval of 1 second and fire the appropriate signal to the target process.

We implement this in the *siggen.c* file which when compiled generates the *siggen.ko* file which can be installed in the system and loaded as and when required.

4.1 Implementation Details

Here we specify the details of the *siggen.c* file which implements the kernel module.

- **Global Scope:**
  We do some initialisations in the global scope.

```
struct pending_signal{
    pid_t pid;
    int signal;
    struct list_head list_node;
};

//initialise the doubly linked list
//initialise a spinlock to secure the list

//initialise a proc_ops struct, specifies the write callback function
//declare a proc_dir_entry struct, stores the details of out proc file
//declare a timer, when it expires it will run a timer callback function
```

- **Function `init_module()`:**
  This function will be called when the module is loaded in memory and performs the basic initialisations.

```
int init_module(){
    //create the proc file system entry with the name sig_target
    //set the timer to expire every 1s and execute the timer callback function
}
```

- **Function `cleanup_module()`:**
  This function will be called when the module is loaded out of memory and frees up all the resources that the module was using.

```
   void cleanup_module(){
        //remove the entry from the procfs
        //delete the timer
        //send any of the remaining signals
        //delete the list head
}
```

- **Function `proc_write_callback()`:**
  The proc file system is a virtual file system, this means that there is no specified location in the hard-disk that corresponds to a particular proc file, instead whenever a user process will try to write to our proc file *sig_target* then that data will be directed to this write callback function, which can then process it. In our case we have maintained a linked list of nodes of the type **`struct pending_signal`** which stores the pid number of the target process and the signal number that needs to be sent to it. Whenever a process writes to the proc file this function will be called which will insert a new node in the list with the target pid and the signal to be sent.

```
   ssize_t proc_write_callback(... const char __user *buffer, size_t data_size ...){

            char kbuffer[data_size];
            //copy from user space buffer to kernel space kbuffer
            //make a new node
            //parse the pid and signal number and store it in the new node

            //aquire lock
            //add the new node
            //release lock

            return data_size;
   }
```

- **Function `pending_signal_timer_callback()`:**
  The timer is set to expire after every 1 second by the **`init_module`** function and whenever it expires it will execute this timer callback function. This function will iterate through the list of pending signals and will send the signals to the target processes one by one. As soon as one signal is sent, that node will also be deleted by this function.

```
    void pending_signal_timer_callback(...){

        //aquire lock
        //for each node in the list:
            pid_t pid=node->pid;
            int signum=node->signal;
            //send the signal to pid
            //delete the node
        //release lock

        //restart timer
    }
```

4.2 Extra Work

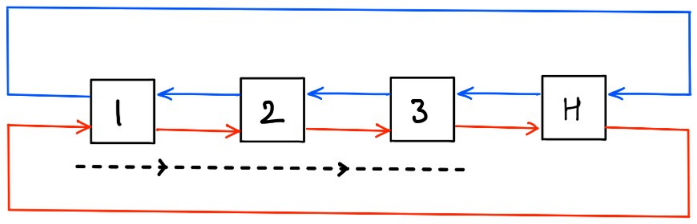
Following are some of the implementations done in addition to the ones that were specified.

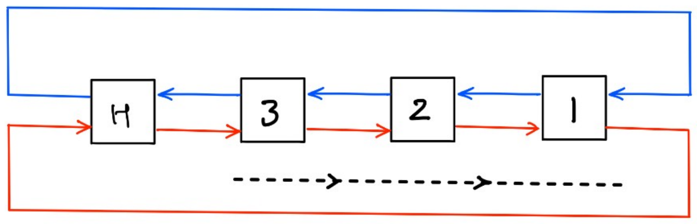
- Used locks to keep the list secure and prevent any racing condition
- The module will check if the target process is still alive before sending the pending signal.
- While cleaning up the module, it will send any signals that remain in the list

5 Miscellaneous Implementation Details

Following are some of the implementation choices made which seem to fit well with the applications.

- **Implementing the lists as queues:**
  The list Kernel API provides two functions to add a new node to the list, `list_add_tail()` and `list_add()`.





The first figure shows the addition order of the nodes for `list_add_tail()` that adds the new node behind the head node and the second figure shows the addition order of nodes for `list_add()` which adds the new node ahead of the head node.

The dotted line in both of them shows the direction of traversal using the list API that follows the next pointers. As we can see, the `list_add_tail()` function is the one which best resembles the queue.

In our use case, where pid's are registered and added to the list (Context Switch Tracker) and where the processes write to the list of pending signals in the proc file (Signal Generator) it makes sense to process the request of the user that was added first and then the later, which is done with a queue and hence we use the `list_add_tail()` function.