In the project we were asked to implement a new system call (*set_myFlag*) to set the value of a flag (*myFlag)* which is added into task descriptor of a process. To be able to add a new flag into the processes we had to change the task descriptor of the processes. In Linux, the kernel stores the list of processes in a circular doubly link list called the *task list*. Each element in the task list is a *process descriptor* which contains all of the information about a process and these process descriptors are the type of *struct task_struct*. Therefore, firstly, we added a new field into that *task_struct* structure which is defined in *<linux/sched.h>* and that header file was located in */include/linux/shed.h* path :

```
1053  struct task_struct {
1054    volatile long state;  /* -1 unrunnable, 0 runnable, >0 stopped */
```

...

```
1464  #if defined(CONFIG_BCACHE) || defined(CONFIG_BCACHE_MODULE)
1465    unsigned int   sequential_io;
1466    unsigned int   sequential_io_avg;
1467  #endif
1468
1469    int myFlag;
1470  };
```

After adding our flag to the task descriptor, to initialize our new field during system initialization (creation of process 0) we needed to update INIT_TASK macro located in */include/linux/init_task.h* :

```
162   #define INIT_TASK(tsk)    \
163   {                         \
164     .state    = 0,          \
165     .stack    = &init_thread_info,    \
```

...

```
217     thread_node   = LIST_HEAD_INIT(init_signals.thread_head), \
218     .myFlag    = 0,                \
219     INIT_IDS                       \
220     INIT_PERF_EVENTS(tsk)          \
```

After those initial operations, we wrote our system call function and added it to the kernel. For this purpose, we created a new folder (*set_myFlag*) under root of the source and in that folder we created our new system call file (*set_myFlag.c*) :

```c
#include <linux/syscalls.h>
#include <linux/kernel.h>
#include <asm/errno.h>

asmlinkage long sys_set_myFlag(pid_t pid, int flag_val)
{
  if (capable(CAP_SYS_ADMIN))
  {
    struct task_struct *tsk = find_task_by_vpid(pid);

    if (tsk != NULL)
    {
      if (flag_val == 1 || flag_val == 0)
      {
        tsk->myFlag = flag_val;
        return 0;
      }
      return -EINVAL;
    }
    return -ESRCH;
  }
  return -EPERM;
}
```

First of this system call can only be used by processes with root privilages so first *if* statement ensures that restriction. If the calling process does not have the root privilages we returned an error (*-EPERM*) to indicate that it is not permitted to do that operation. Then to be able to obtain a pointer to the task descriptor of a process with its pid we examined *sched.h* file and found the appropriate function *find_task_by_vpid()* which takes pid as the parameter. Then we simply checked whether we obtained the pointer or not. If not, we returned an error (*-ESRCH*) to indicate a process with that pid could not be found. And finally, we checked the given flag value to the function. If it is not zero (0) or one (1), we returned another error (*-EINVAL*) to indicate value is invalid. If all constraints are satisfied, we assigned flag value to the current process.

After writing our system call, we also created a *Makefile* under that folder and added it to the Makefile of the source to make this call available at the compilation.

/set_myFlag/Makefile:

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

/Makefile:

```
drivers-y := drivers/ sound/ firmware/ ubuntu/
net-y     := net/
libs-y    := lib/
core-y    := usr/ set_myFlag/
```

Then we modified system call table under */arch/x86/syscalls/syscall_32.tbl* and system call header file under */include/linux/syscall.h*.

/arch/x86/syscalls/syscall_32.tbl:

```
362   353  i386 renameat2      sys_ni_syscall
363   354  i386 seccomp        sys_seccomp
364   355 i386  set_myFlag       sys_set_myFlag
365
```

/include/linux/syscall.h:

```
850     asmlinkage long sys_seccomp(unsigned int op, unsigned int flags,
851             const char __user *uargs);
852     asmlinkage long sys_set_myFlag(pid_t pid, int flag);
853     #endif
```

Now, since all of the operations regarding system call was finished, we could start to second requirement of the project: modifying fork and exit system calls. In Linux, new processes are created by *fork* system call and that call uses *do_fork* function located in */kernel/fork.c*. So, we were asked to change that function so that, if the value of our new flag (myFlag) is zero (0), default actions will be performed but it is one (1) and its nice value should is greater than 10 no processes will be created.

```
1642    long do_fork(unsigned long clone_flags,
1643          unsigned long stack_start,
1644          unsigned long stack_size,
1645          int __user *parent_tidptr,
1646          int __user *child_tidptr)
1647    {
1648        struct task_struct *tsk = current;
1649        // Do not create a child process under this situations
1650        if (tsk->myFlag == 1 && task_nice(tsk) > 10)
1651            return -ECHILD;
```

Here, inside do_fork function, first of all we read the current process and checked its flag value and nice value. If its flag is 1 and nice value is greater then 10, we returned from the function with an error (*-ECHILD*) to indicate no child process will be created. If these conditions are satisfied, do_fork function calls another function *copy_process* to create new child process. Therefore, we simply added a line to initialize flag of child processes.

```
1191    static struct task_struct *copy_process(unsigned long clone_flags,
1192              unsigned long stack_start,
1193              unsigned long stack_size,
1194              int __user *child_tidptr,
1195              struct pid *pid,
1196              int trace)
1197    {
1198        int retval;
1199        struct task_struct *p;
```

•••

```
1289
1290        p->myFlag = 0;
1291
```

Finally, we are also asked to modify exit system call so that, if the myFlag value of process is zero (0), default actions will be performed, but if its value is one (1) and its nice value is greater than 10, not only that specific process, but all of its siblings will also exit from the system. In linux processes terminated via exit system call and for this purpose *do_exit* function is used located in */kernel/exit.c* file. So, we made some changes in that function.

```
709     if (tsk->myFlag == 1 && task_nice(tsk) > 10)
710     {
711       struct list_head *sibling_list;
712       struct task_struct *_sibling;
713       struct task_struct *_parent = current->parent;
714       list_for_each(sibling_list, &_parent->children)
715       {
716         _sibling = list_entry(sibling_list, struct task_struct, sibling);
717         siginfo_t siginfo;
718         kill_proc_info(SIGKILL, &siginfo, _sibling->pid);
719       }
720     }
```

Here, first of we checked whether the flag value of the current process is 1 and its nice value is greater than 10, or not. If these conditions were satisfied, we needed to kill all of the siblings of that process, too. To be able to do that, we used *list_for_each* and *list_entry* macros. Firstly we created a linked list called *sibling_list,* then we created two task_struct pointers to point siblings (*_sibling*) and the parent of the current process (*_parent*). After that we used *list_for_each* macro to iterate over all of the childrens of that parent and by using *list_entry* macro we obtained those child processes and terminated them by sending *SIGKILL* signal.