Project 1

Report

Course Name: BLG413E – System Programming
Date: 12.10.2017

Group Information		
Student ID	**Name**	**CRN**
150140119	Muhammed Kadir Yücel	13331
150140123	*Mahmut Lutfullah Özbilen*	*13332*

*This report was prepared by Muhammed Kadir Yücel (*150140119*).

Table of Contents

Introduction

This report was prepared for Project 1 of System Programming course by Muhammed Kadir Yücel (*150140119*). In this project, we aimed to work on Linux kernel source code to add a new system call that will affect general working structure of *exit()* system call. A flag value should be added to process definition structure, if flag is set and nice value of process is bigger than 10 when process is exited all children of this process will be killed.

We have worked on Linux Kernel 3.13.0 version on this project with Lubuntu 14.04.5 system. Updated source files and tests programs can be found inside *.zip file provided through Ninova system. Sources provided in folder structure as in the kernel sources structure.

Compiling

1. Download *.zip file provided through Ninova and Linux 3.13.0 kernel sources.
2. Extract *.zip file inside kernel source code folder.
3. Open terminal in Linux kernel source folder.
4. `make defconfig`
5. `make-kpkg clean`
6. `fakeroot --initrd --append-to-version=-custom kernel_image kernel_headers`
7. Install *.deb packages and reboot system with new kernel.

Source Fix

Unmodified kernel source had some problem at compilation time if the system is not configured as EFI. We solved this problem with the help of our teaching assistants and updated *head_32.S* file which caused the problem. *.zip file that was provided through Ninova includes this file as updated.

Project Work

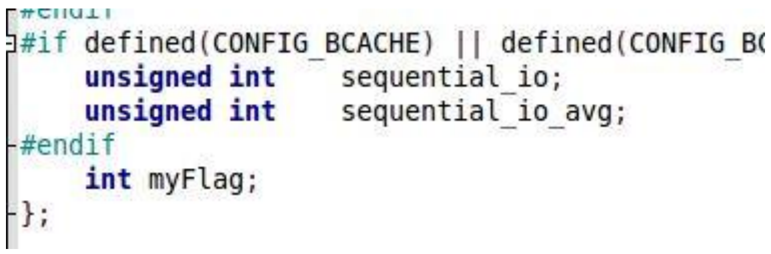
We have changed some kernel codes and added our code to implement new system call. Also we have implemented and conducted some test programs to test if added system call works as expected.

System Call Implementation

We have worked under *linux-source-3.13.0* folder which is the root folder for kernel source structure. In this report it will be called as *main folder* for further paragraphs.

Modifying Process Structure

We needed to process structure defined as *task_struct* in *sched.h* file in kernel code to add flag value to be checked by *exit()* system call and updated by system call that is implemented. Flag is an *int* type of variable that was added to end of the *task_struct* structure. We have added it to the end of the structure because some kernel code may implemented to work with exact mapped structure of *task_structure* with direct memory addresses. By adding our new value to end of the structure, we avoided that kind of problems that may happen.



```
#if defined(CONFIG_BCACHE) || defined(CONFIG_BC
    unsigned int    sequential_io;
    unsigned int    sequential_io_avg;
#endif
    int myFlag;
};
```

Figure 1: myFlag value at the end of task_struct

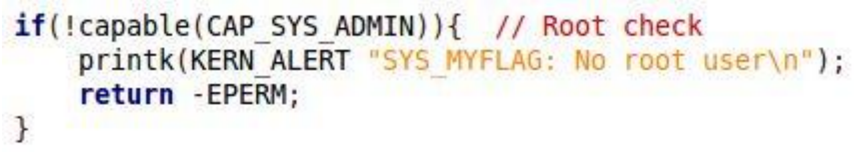
Adding System Call

We have created *myflag* folder inside *main folder* to keep our implementation of system call. Under *myflag* folder we have created two files;

- myflag.c
  Holds implementation of system call.
- Makefile
  This file is required to add and define new system call to compilation process.

System call functions must have *sys_* prefix. So our system call's function name is *sys_set_myFlag* which takes two arguments which are PID of process and flag value. We needed to check root permissions and flag value as a project requirement;

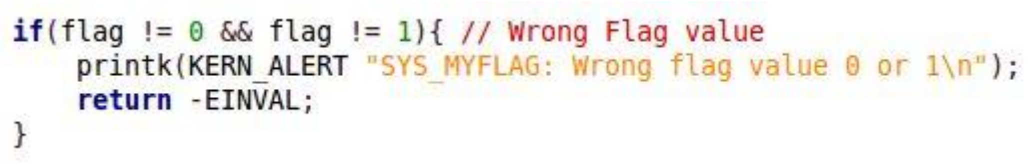
We checked root permissions with *capable* function provided by kernel. If user does not have root privileges, system call will print a kernel alert and return permission error.



```
if(!capable(CAP_SYS_ADMIN)){  // Root check
    printk(KERN_ALERT "SYS_MYFLAG: No root user\n");
    return -EPERM;
}
```

Figure 2: Root check

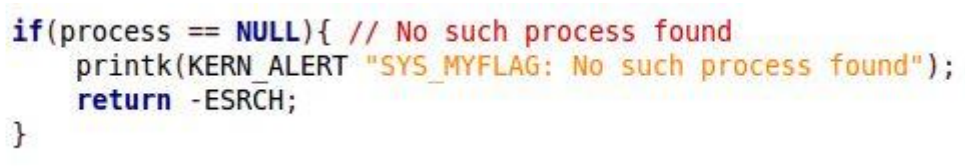
We checked flag value by integer comparison to 0 and 1. Flag value will be checked after root privilege check. If user entered a flag value different than 0 or 1, system call will print a kernel alert than return invalid value error.



```
if(flag != 0 && flag != 1){ // Wrong Flag value
    printk(KERN_ALERT "SYS_MYFLAG: Wrong flag value 0 or 1\n");
    return -EINVAL;
}
```

Figure 3: Flag value check

System call will search for process with given PID by using a function provided by kernel. This function returns a pointer to *task_struct* of process by given PID. If no such process is found, this function returns NULL pointer. So we have to check if returned value is NULL or not to update the *myFlag* value inside structure. If there is no process found inside currently running processes of system, search error will be returned. NULL check implemented after both root and flag value checks.

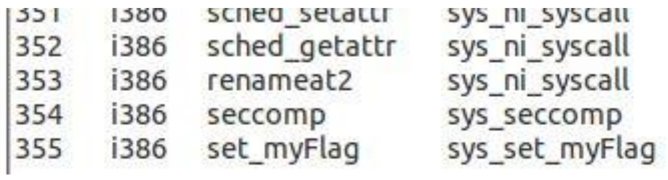


```
if(process == NULL){ // No such process found
    printk(KERN_ALERT "SYS_MYFLAG: No such process found");
    return -ESRCH;
}
```

Figure 4: Check if there are any process

After implementing system call function we created a *Makefile* inside *myflag* folder and updated *Makefile* inside main folder to add our new system call function's folder (*myflag*) to compilation process.

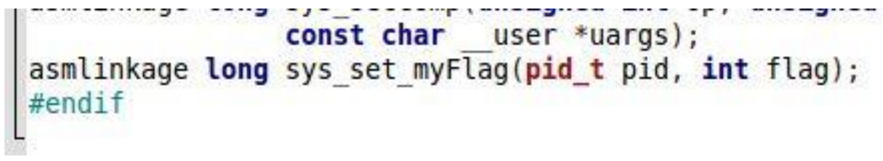
We have to define our system call inside kernel headers and system call table to execute system call from user space. We have added our system call to the end of the system call table file (*syscall_32.tbl*) with ID **355**.



```
351   i386   sched_setattr   sys_ni_syscall
352   i386   sched_getattr   sys_ni_syscall
353   i386   renameat2       sys_ni_syscall
354   i386   seccomp         sys_seccomp
355   i386   set_myFlag      sys_set_myFlag
```

Figure 5: System Call table updated

We have added prototype of the system call that we have implemented to the *syscalls.h* file which includes other implemented system calls' prototypes.

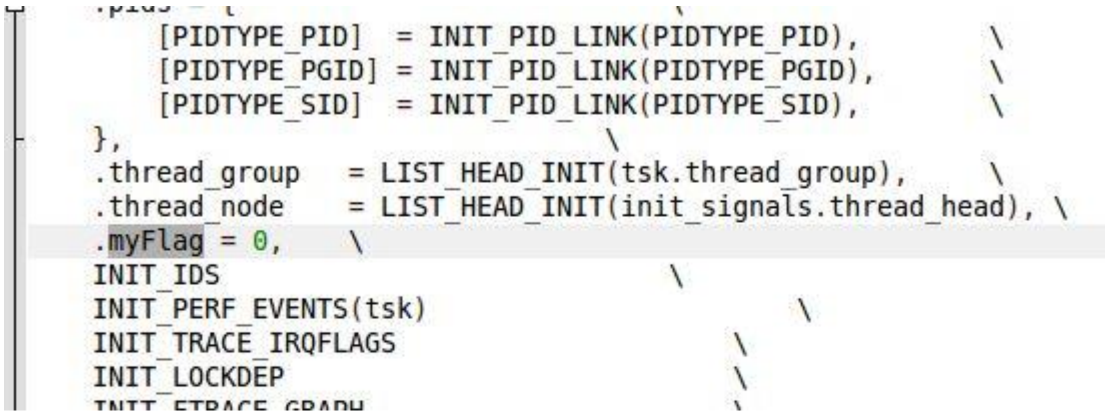


```
asmlinkage long sys_seccomp(unsigned int op, unsigned
                const char __user *uargs);
asmlinkage long sys_set_myFlag(pid_t pid, int flag);
#endif
```

Figure 6: System Call Prototype added

Updating *fork()* System Call

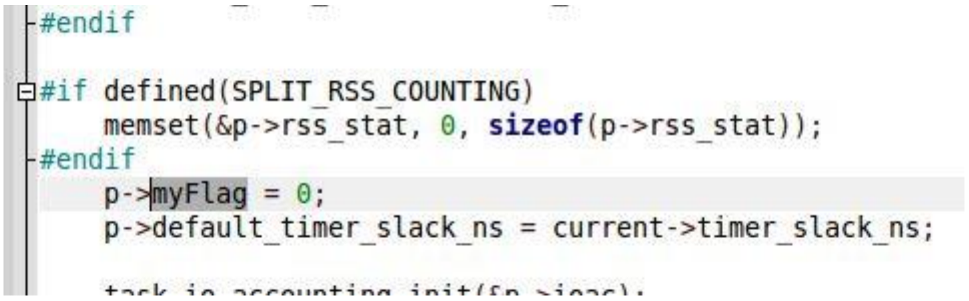
We have added flag value to the *task_struct* but we have to initialize that value for further operations. As a project requirement we needed to initialize it with value 0. Since processes are created with *fork()* system call we have to update it. But this flag value will also be inside the *init* process which is not created by *fork()* system call. So we have to initialize the same flag value for *init* process as well. *init* process is initialized by a macro which has name *INIT_TASK*. This macro can be found inside *init_task.h* file.



```
        [PIDTYPE_PID]  = INIT_PID_LINK(PIDTYPE_PID),        \
        [PIDTYPE_PGID] = INIT_PID_LINK(PIDTYPE_PGID),       \
        [PIDTYPE_SID]  = INIT_PID_LINK(PIDTYPE_SID),        \
    },                          \
    .thread_group   = LIST_HEAD_INIT(tsk.thread_group),     \
    .thread_node    = LIST_HEAD_INIT(init_signals.thread_head), \
    .myFlag = 0,    \
    INIT_IDS                        \
    INIT_PERF_EVENTS(tsk)                   \
    INIT_TRACE_IRQFLAGS                 \
    INIT_LOCKDEP                        \
```

Figure 7: Macro updated for init process

fork() system call's function can be found inside *fork.c* source code file. We have updated *copy_process* function inside this source file to initialize the flag value. Since *do_fork()* function, which is the function for *fork()* system call, calls *copy_process* for new *task_struct* creation. We thought that it is better to initialize flag value inside that function which makes us to be sure about flag values will be 0 for each child process no matter what flag value has the parent process.



```
#endif

#if defined(SPLIT_RSS_COUNTING)
    memset(&p->rss_stat, 0, sizeof(p->rss_stat));
#endif
    p->myFlag = 0;
    p->default_timer_slack_ns = current->timer_slack_ns;
```

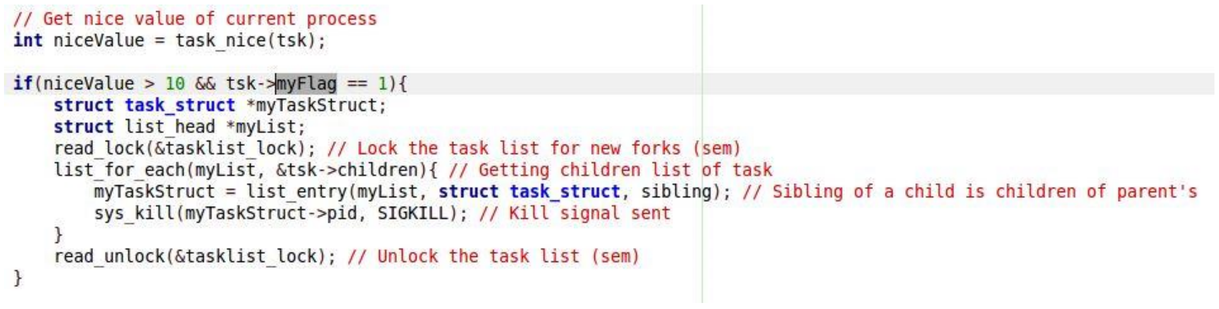
Figure 8: Flag value initialized for fork

Updatig *exit()* system call

We added this flag value to process structure and system call to update flag value to change the *exit()* system call's behavior. In Linux systems, each process has to have a parent, except for *init* process. When a parent process dies, invokes *exit()* system call, it will be removed from the system but if it has children processes, all children process will be assigned to new parent process, which is in general *init* process. Flag value that we've added is there for the change this behavior. If the flag value of a process is 1 and it has a nice value greater than 10, all children processes of this process will be killed also instead of assigned to another process.

exit() system call invokes *do_exit()* function which can be found inside *exit.c* source code file. We've implemented flag check and children kill functions here inside *do_exit()* function. Inside this function there is a pointer named as *tsk* which holds currently calling process' *task_struct* structure. Firs we needed to get nice value of current process, we called *task_nice* function which needs *task_struct* pointer as argument and returns nice value of process. This function is provided by kernel. After we've checked if nice value is greater than 10 and flag value is 1.

When requirements were met, we started to iterate over children processes of current process and signaling them with kill signal one by one.



```
// Get nice value of current process
int niceValue = task_nice(tsk);

if(niceValue > 10 && tsk->myFlag == 1){
    struct task_struct *myTaskStruct;
    struct list_head *myList;
    read_lock(&tasklist_lock); // Lock the task list for new forks (sem)
    list_for_each(myList, &tsk->children){ // Getting children list of task
        myTaskStruct = list_entry(myList, struct task_struct, sibling); // Sibling of a child is children of parent's
        sys_kill(myTaskStruct->pid, SIGKILL); // Kill signal sent
    }
    read_unlock(&tasklist_lock); // Unlock the task list (sem)
}
```

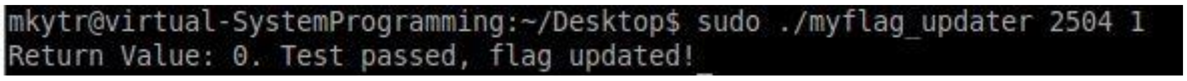
Figure 9: exit system call updated

We've gained a lock for task list of system, since we are entering a critical section. If one of the children of the current process is killed after we got its PID but before we signaled it to be killed, it may become a fatal error. Since we would signal a non-existing process. We read locked the task list and started to iterate over list. List head will be the *children* list head of current process that can be found inside *task_struct*. Since task list is a doubly linked list in Linux's kernel, *sibling* of the head will be the other child of the current process, *sibling* is also another list head that can be found inside *task_struct*. After killing processes we released the lock of task list.

Tests

We provided some test programs inside *.zip file provided through Ninova.

- myflag_updater
  With this program we can test the system call, if it updates the flag value. We can also check root privilege checking and flag value checking with this program.
- exit_tester
  This program forks 2 children processes, which makes 3 processes total with parent, and each process will wait for user input. Flag value should be given while starting this test program. It will set flag value as given value. Because of that it needed to be run as root. After you give flag value as 1 and update renice value of parent process, if you terminate parent process from task manager, all children will be terminated also and removed from task manager.

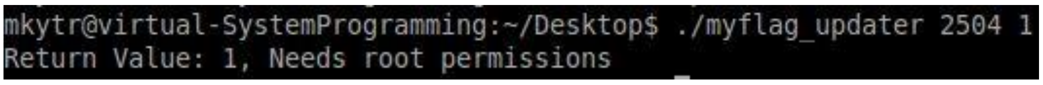


```
mkytr@virtual-SystemProgramming:~/Desktop$ sudo ./myflag_updater 2504 1
Return Value: 0. Test passed, flag updated!
```

Figure 10: When everything works

Checking Root Privileges

Run *myflag_updater* without root privileges.

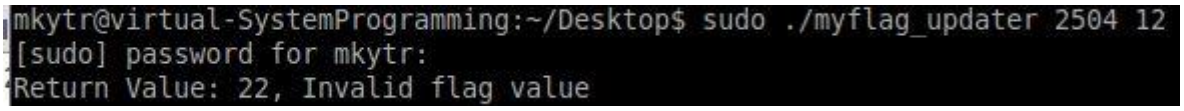


```
mkytr@virtual-SystemProgramming:~/Desktop$ ./myflag_updater 2504 1
Return Value: 1, Needs root permissions
```

Figure 11: User without root privileges try to update flag value

Checking Invalid Flag Value

Run *myflag_updater* with flag values different than 0 or 1.

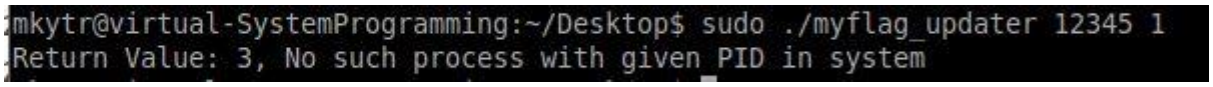


```
mkytr@virtual-SystemProgramming:~/Desktop$ sudo ./myflag_updater 2504 12
[sudo] password for mkytr:
Return Value: 22, Invalid flag value
```

Figure 12: Invalid flag value (12) sent to system cal

Checking Invalid PID Value

Run *myflag_updater* with flag value that is not assigned to any process inside system.



```
mkytr@virtual-SystemProgramming:~/Desktop$ sudo ./myflag_updater 12345 1
Return Value: 3, No such process with given PID in system
```

Figure 13: PID value that is not assigned to any process given (12345)

Conclusion

In this project we have learned;

- How system calls work
- Implementing a system call
- Switching from user space to kernel space
- Executing system call from program in user space
- How a process defined in Linux kernel, *task_struct*
- How *fork()* and *exit()* system calls work
- How to check root permissions inside a system call

We have gained some skills like;

- Understanding code structure of kernel
- Finding default system calls from system call table
- Compiling Linux kernel and booting computer from new kernel
- Applying changes to kernel code

With this project we have learned and gained important key points that will help us through our future work life about computers. Linux can be found inside many embedded devices which may need special abilities from kernel that are not implemented yet. With abilities we gained with this project, we will be able to implement these requirements to accomplish needs of users and environments.