# BLG413E-System Programming
# Project 1

Merve Elif Demirtaş - 150160706
Gülnur Kaya - 150160154
Buse Dilan Uslan - 150150063

30 October 2018

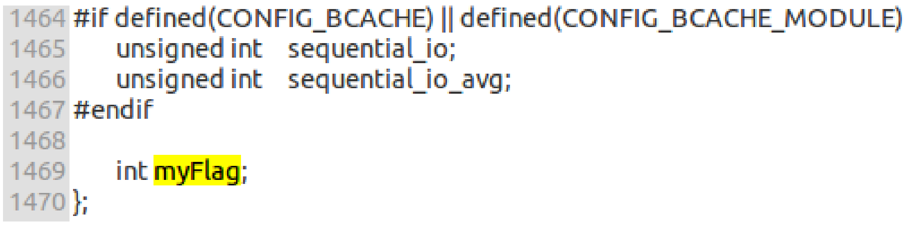
## 1 Introduction

In this project, firstly, a new variable -flag- added to the task descriptor of a process and then the behaviour of fork and exit system calls were changed based on the flag and nice value.

## 2 Changing the Flag Value

Each process has a descriptor related with it. This descriptor includes the information for tracking the process in memory. These information are PID, state, parent process, children, siblings, list of open files, etc. For using the flag value in kernel and also in user space, it has to declared in task descriptor.

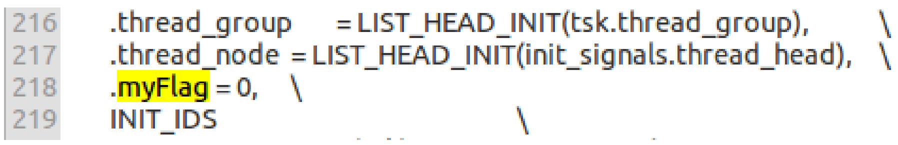
- The flag variable declared in task_descriptor. In the task descriptor, the variable added at the end of the file.



```
#if defined(CONFIG_BCACHE) || defined(CONFIG_BCACHE_MODULE)
    unsigned int   sequential_io;
    unsigned int   sequential_io_avg;
#endif

    int myFlag;
};
```

Figure 1: Flag variable declaration in task_struct

- The flag variable added to the task descriptor of mother process at the file where it is initialized in \include\linux \init_task.h . The reason of adding the variable to the mother process is fork function. When a fork is called, the child copies all the data of parent.



```
.thread_group     = LIST_HEAD_INIT(tsk.thread_group),       \
.thread_node = LIST_HEAD_INIT(init_signals.thread_head),   \
.myFlag = 0,   \
INIT_IDS                         \
```

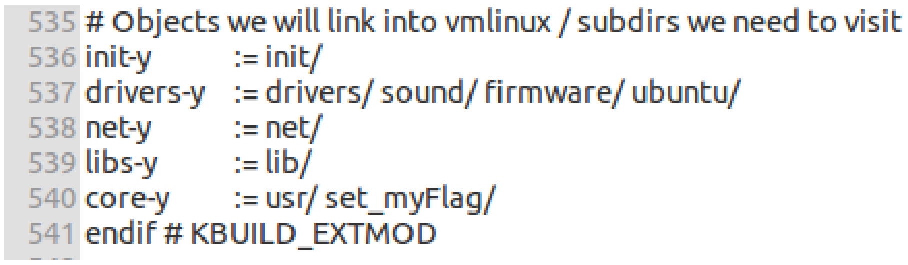
Figure 2: Flag variable initialization in init_task

Under linux-source-3.13.0, set_myFlag/set_myFlag.c function created as below.

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

asmlinkage long sys_set_myFlag(pid_t pid, int flag){

   if((current->cred)->uid == 0 && task_nice > 10){
      struct task_struct *process_ptr = find_task_by_vpid(pid); //return
          the pointer of pid process

      if(process_ptr != NULL){// there is process
         if(flag == 0 || flag == 1){ //valid flag values
            process_ptr-> myFlag = flag;
            return 0;
         }else{
            return EINVAL; //invalid value
         }
      }else{
         return ESRCH; //no such process
      }
   }else{
      return EPERM;
   }
}
```

- In the set_myFlag file, also, a Makefile created and written inside ”obj-y := set_myFlag.o” .

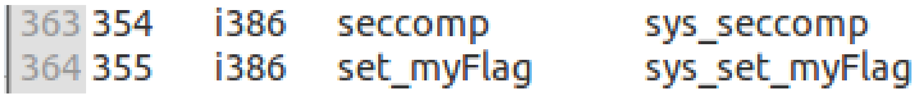
- Under linux-source-3.13.0, the Makefile modified adding the set_myFlag to core-y as below.



```
# Objects we will link into vmlinux / subdirs we need to visit
init-y          := init/
drivers-y       := drivers/ sound/ firmware/ ubuntu/
net-y           := net/
libs-y          := lib/
core-y          := usr/ set_myFlag/
endif # KBUILD_EXTMOD
```

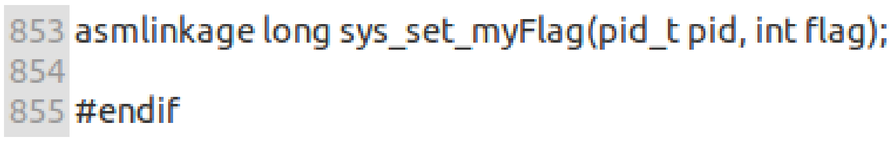
Figure 3: Makefile of linux-source-3.13.0

- System call table and system call header file was also modified as below.



```
354     i386    seccomp         sys_seccomp
355     i386    set_myFlag      sys_set_myFlag
```

Figure 4: System call table declaration



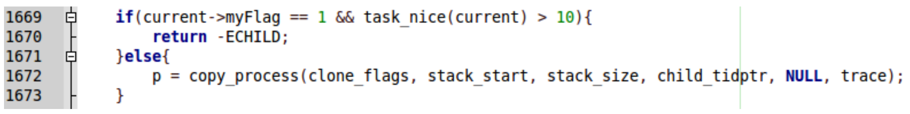
```
asmlinkage long sys_set_myFlag(pid_t pid, int flag);

#endif
```

Figure 5: System call header file declaration

## 3 Changing the Behaviour of Fork and Exit System Call

### 3.1 Fork System Call

In this part, the behaviour of fork call system changed as when a process has flag value as one, this process can not do fork operation and if it has flag value zero, it can do normal fork operation. When fork system is called, the kernel calls the do_fork function under kernel _fork.c . So, in the do_fork function, copy_process was taken into if function based on flag and nice value as below.



```
    if(current->myFlag == 1 && task_nice(current) > 10){
        return -ECHILD;
    }else{
        p = copy_process(clone_flags, stack_start, stack_size, child_tidptr, NULL, trace);
    }
```

Figure 6: do_fork function

This modification tested as below.

```
#include <stdio.h>
#include <stdlib.h>
#include <asm/errno.h>
#include <unistd.h>
#include <string.h>


#define set_myFlag 355


int main(){

   int flag = 1;

   int check_return;
   printf("getpid(): %d , getppid(): %d \n", getpid(), getppid());
   check_return = syscall(set_myFlag, getpid(), flag);//set flag value
       to mother process

   printf("Return value of set_myFlag: %s\n", strerror(check_return));

   int f;
   f = fork();
   printf("f value: %d\n", f);

   if(f == 0){ //child process
      printf("Child pid: %d child parent pid:%d\n", getpid(), getppid());

      return 0;
   }else if(f < 0){
      printf("Return value: %s\n", strerror(f));
      printf("getpid(): %d , getppid(): %d \n", getpid(), getppid());
   }else{
      printf("Parent pid: %d\n", getpid());
   }

   return 0;
}
```

## 3.2 Exit System Call

In this part, the behaviour of exit system call was changed as when a child process calls the exit function, it will exit with killing all the siblings. The Linux kernel uses a circular doubly-linked list of struct task _structs to store process descriptors.

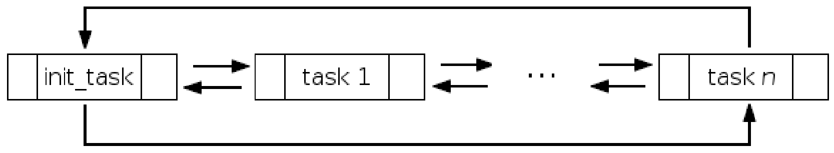
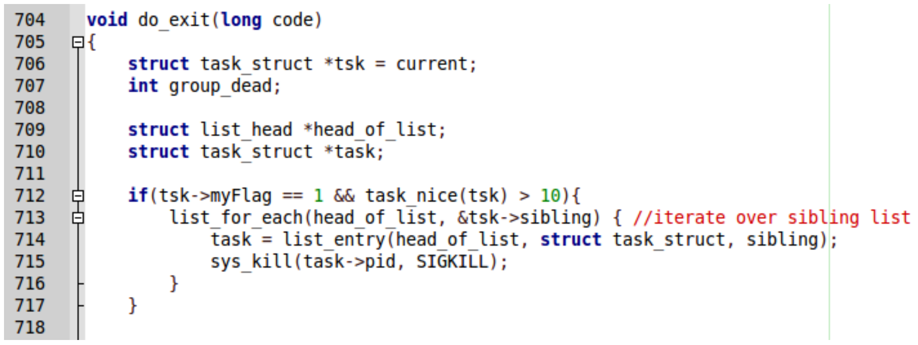


Figure 7: Circular doubly-linked list of struct task _structs

So, the sibling list of child iterated over list and killed with sys_kill function as below.



```
void do_exit(long code)
{
    struct task_struct *tsk = current;
    int group_dead;

    struct list_head *head_of_list;
    struct task_struct *task;

    if(tsk->myFlag == 1 && task_nice(tsk) > 10){
        list_for_each(head_of_list, &tsk->sibling) { //iterate over sibling list
            task = list_entry(head_of_list, struct task_struct, sibling);
            sys_kill(task->pid, SIGKILL);
        }
    }

```

Figure 8: do_exit function under kernel_exit.c

This modification tested as below.

```
#include <stdio.h>
#include <stdlib.h>
#include <asm/errno.h>
#include <unistd.h>
#include <string.h>

#define set_myFlag 355

int main(){

    int f, i, check_return, flag, tmp;
    printf("FIRST -> getpid(): %d, getppid(): %d \n", getpid(),
         getppid());

    for(i = 0; i < 5; i++){
        f = fork();

        if(i == 4){
            if(f == 0){//child process
                flag = 1;

                int check_return;
                printf("PROCESS CALLS THE EXIT -> getpid(): %d ,
                     getppid(): %d \n", getpid(), getppid());
                check_return = syscall(set_myFlag, getpid(), flag);//try
                     to set flag to child process

                printf("PROCESS CALLS THE EXIT -> Return value of
                     set_myFlag: %s\n", strerror(check_return));

                sleep(5);
                printf("The child that calls the exit call was died!\n");
                exit(0);
            }
        }

        if(f == 0){
            printf("Waiting process child: getpid(): %d , getppid(): %d
                 \n", getpid(), getppid());
            while(1);
        }
    }

    sleep(10);
    return 0;
}
```