System Programming CRN-13332

Project-1 Report

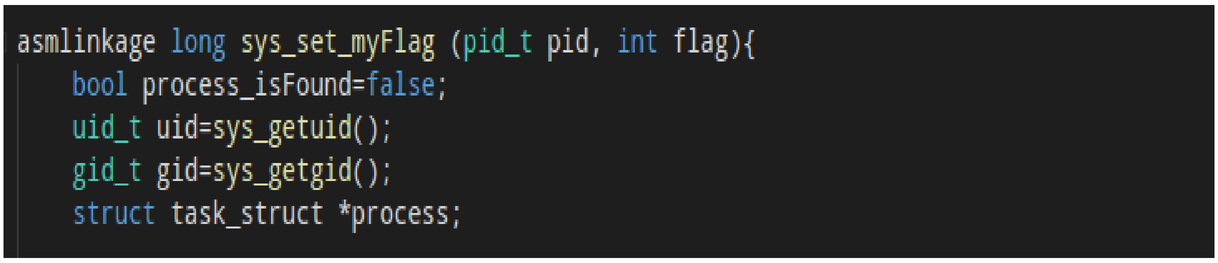
Prepared by Özgür AKTAŞ - 150160530

1. Introduction

The interface defined between the operating system and the user programs is a set of procedures defined by the operating system. These procedures defined by the operating system are called system calls. The system calls for the specified operating system may be different in the cluster operating systems. Although the names are different, the operations performed on the backplane are similar. User programs use these defined system calls for hardware or special operations. The operations to be performed in a system call are completely machine-dependent, so they are often defined in the assembly programming language. C and other programming languages define procedure libraries to use these procedures. In this project, we used the C programming language. We are required to write a system call which sets the value of a flag (*myFlag*) in the task descriptor of a process and modify the "*exit*" system call to change its behavior based on the value of the flag.

2. Details

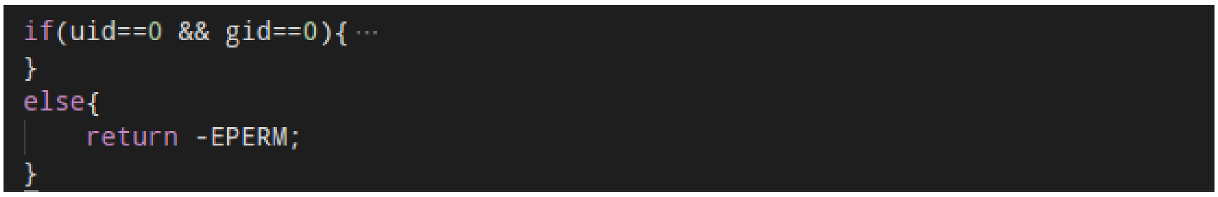
For this reason, we create *set_myFlag* folder. After that we create *set_myFlag.c* file in this folder. We write C codes on this file and the codes begin with *asmlinkage long set_myFlag (pid_t pid, int flag)* system call which can set the value of this flag for the process with the given *pid.* In our call, only processes having root privileges can successfully execute this system call. When the error occurs, the *set_myFlag* system call returns an appropriate error message, otherwise, it returns 0.



```
asmlinkage long sys_set_myFlag (pid_t pid, int flag){
    bool process_isFound=false;
    uid_t uid=sys_getuid();
    gid_t gid=sys_getgid();
    struct task_struct *process;
```

Figure- 1

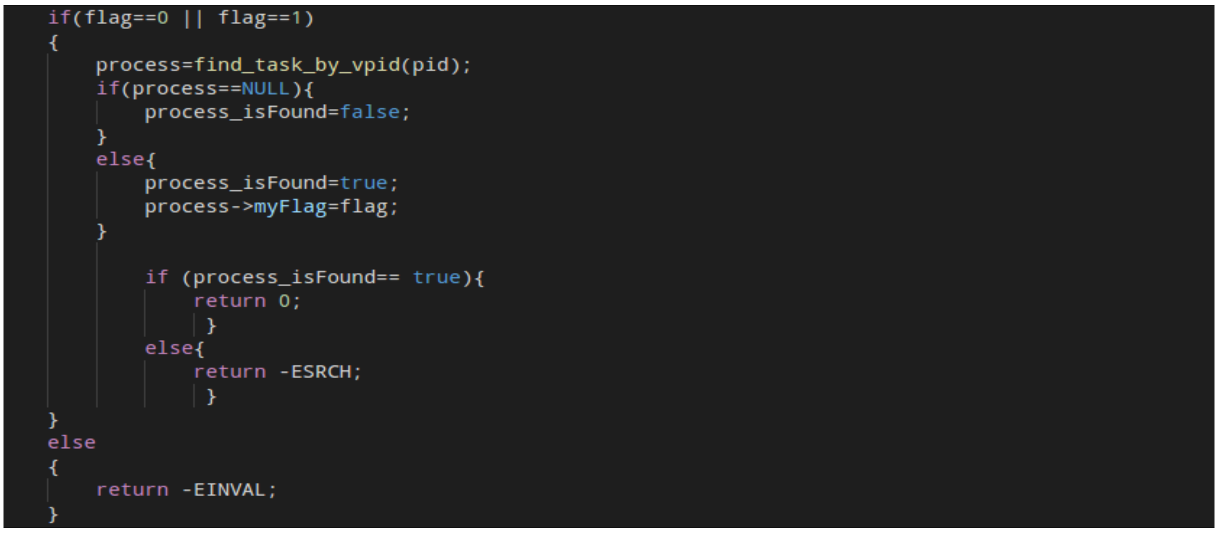
In Figure-1, our system call is shown. We get the *uid* and *gid* values by *sys_getuid ( )* and *sys_getgid ( )* operations. The purpose of getting these values is to determine the processes have root privileges or not.



```
if(uid==0 && gid==0){···
}
else{
    return -EPERM;
}
```

Figure- 2

In Figure-2, if both uid and gid values are equal to 0 (zero) it means that this user has root privileges and do normal operations, else it means that this user does not have root privileges and it gives an error “Operation not permitted” named –EPERM. When we put the figures we cleaned up the comment lines because the figures were too long and took up extra space in the report.



```
if(flag==0 || flag==1)
{
    process=find_task_by_vpid(pid);
    if(process==NULL){
        process_isFound=false;
    }
    else{
        process_isFound=true;
        process->myFlag=flag;
    }

        if (process_isFound== true){
            return 0;
            }
        else{
            return -ESRCH;
            }
}
else
{
    return -EINVAL;
}
```

Figure- 3

In Figure-3, we control the flag number which can be either 1 or 0 to determine argument are taken correctly or not.

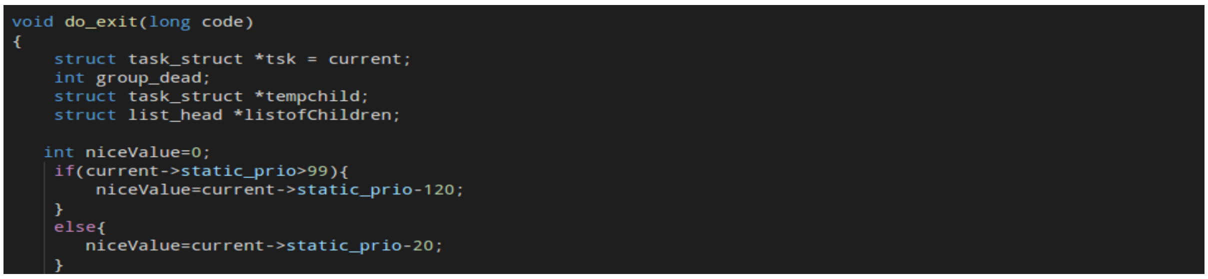
```
✓ if the arguments are correct:
        then it gets the process in task_struct type shown in Figure-1.
            if process is equal to zero it means “Process is NOT found”
                it makes process_isFound value to “false”
            else it means “Process is found”
                it makes process_isFound value to “true”
                gives myFlag of process to flag value
                if process_isFound is “true”
                    it returns 0 and exits normally
                else
                    it returns –ESRCH value. It means “No such process”
  else
        It returns the –EINVAL value. It means “Invalid argument”.
```

For example, if the flag number is entered 2 by user, then it will give EINVAL error. Basically, our system works as described in Figure-1, 2 and 3. We will talk all possibilities in "Test" part.

After system call, we followed the steps below to achieve this project:

- ✓ We added *int myFlag* at the end of task_struct in /include/linux/sched.h folder.
- ✓ To initialize *myFlag* value, we wrote *myFlag = 0* \ into the *define_init_task* in the */include/linux/init_task.h* file with appropriate format.
- ✓ We created *Makefile* file in *set_myFlag* folder and write in *Makefile:*
    - o *obj-y := set_myFlag.o*
- ✓ We wrote *core-y: = usr / mycall / set_myFlag* on line 540 of the *Makefile* in the linux file.
- ✓ We added *i386 set_myFlag sys_set_myFlag* expression to the end of *arch/x86/syscalls/syscall_32.tbl* file
- ✓ We added our system call *asmlinkage long set_myFlag (pid_t pid, int flag);* include/linux/syscalls.h file.

After these steps, we made some modifications in "exit" file which are given below:



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

   int niceValue=0;
    if(current->static_prio>99){
        niceValue=current->static_prio-120;
    }
    else{
       niceValue=current->static_prio-20;
    }
```

Figure- 4

In Figure-3, we made all modifications on *do_exit* part. We created our variables firstly as you see on figure. We initialized *niceValue* as 0.

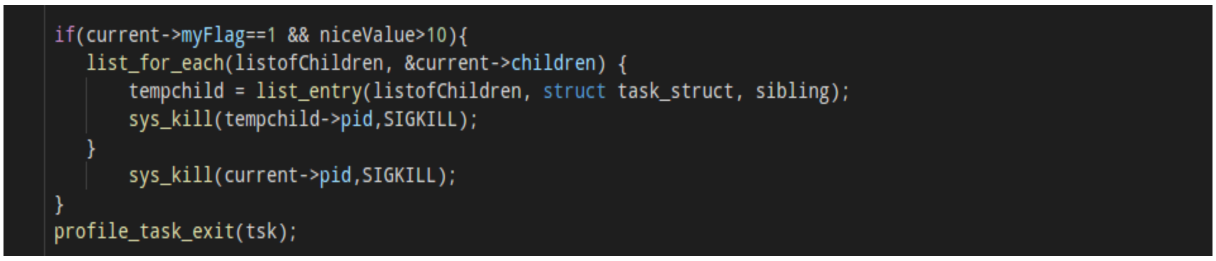
- ✓ if the *static_prio* value of current process greater than 99

    then we reduce this value by 120 and give it to *niceValue*. It's about relationship between priority and nice value of the process.

  else

    then we reduce this value by 20 and give it to *niceValue*.

After these steps, we checked the priority if it is larger than 30 or not by these steps which are given below in Figure-5.



```
if(current->myFlag==1 && niceValue>10){
   list_for_each(listofChildren, &current->children) {
       tempchild = list_entry(listofChildren, struct task_struct, sibling);
       sys_kill(tempchild->pid,SIGKILL);
   }
       sys_kill(current->pid,SIGKILL);
}
profile_task_exit(tsk);
```

Figure- 5

As seen in Figure-5, we controlled niceValue and myFlag values of current process.

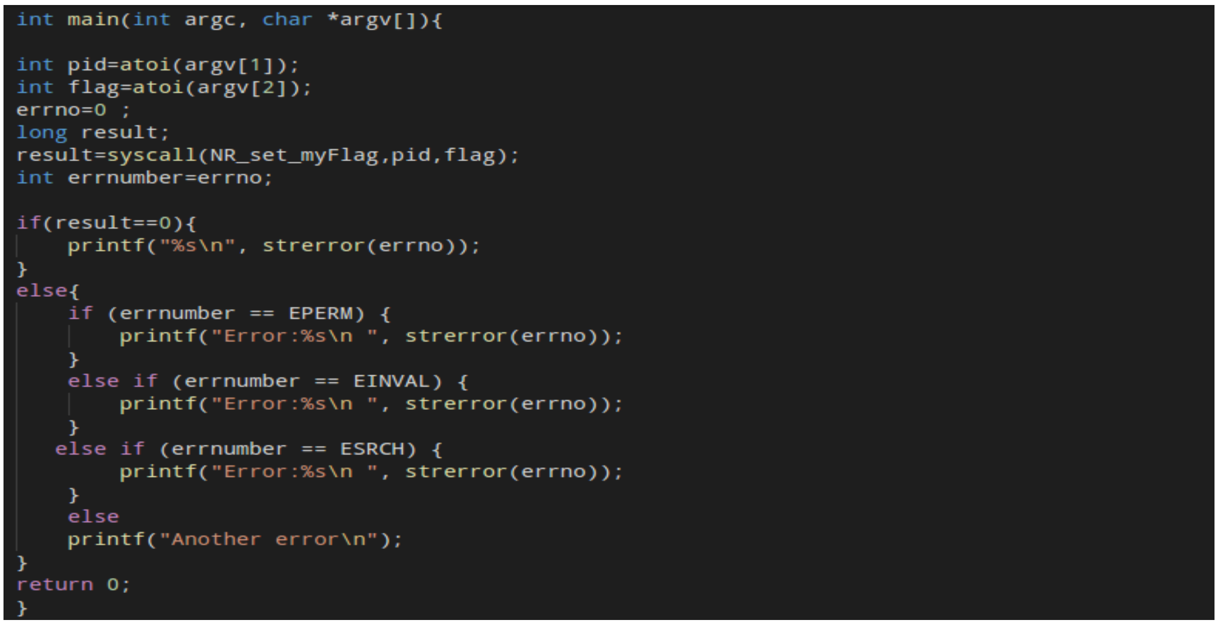
- ✓ if they *myFlag* is equals to 1 and *niceValue* is greater than 10
  then we traversed on all child processes and killed them with *sys_kill ( )* function in *list_for_each* structure.

  In the last step, the process exits normally with *profile_task_exit(tsk)* operation.

After all these changes, we compiled the kernel as shown in Practice Session and we coded the test part.

3. Test

For the test part, we wrote codes which are in Figure-6 given below:



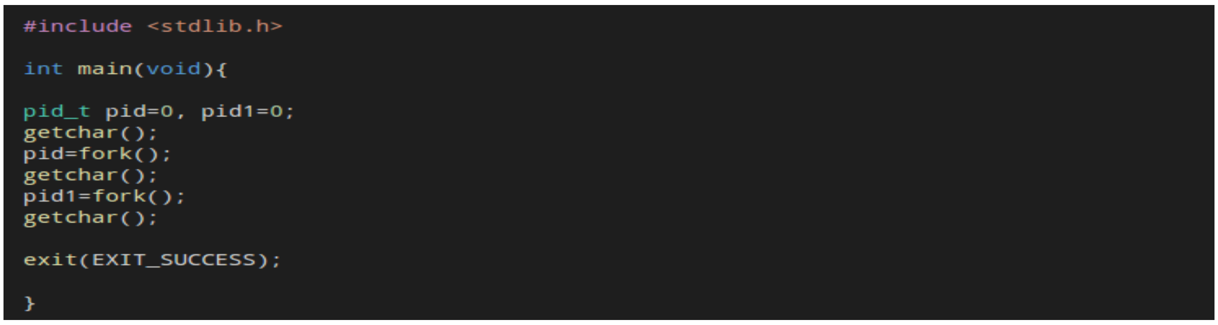
```
int main(int argc, char *argv[]){

int pid=atoi(argv[1]);
int flag=atoi(argv[2]);
errno=0 ;
long result;
result=syscall(NR_set_myFlag,pid,flag);
int errnumber=errno;

if(result==0){
    printf("%s\n", strerror(errno));
}
else{
    if (errnumber == EPERM) {
        printf("Error:%s\n ", strerror(errno));
    }
    else if (errnumber == EINVAL) {
        printf("Error:%s\n ", strerror(errno));
    }
   else if (errnumber == ESRCH) {
        printf("Error:%s\n ", strerror(errno));
    }
    else
    printf("Another error\n");
}
return 0;
}
```

Figure- 6

In Figure-6, we get *pid* and *flag* values from linux console. We created a variable named *errnumber* and give *errno* value to *errnumber.*

- ✓ if the *result* value which is result of *syscall* equals to 0
  it prints to console to *strerror(errno)* value. It means the system call is successful

  else it prints the appropriate error massages to the linux console.



```
#include <stdlib.h>

int main(void){

pid_t pid=0, pid1=0;
getchar();
pid=fork();
getchar();
pid1=fork();
getchar();

exit(EXIT_SUCCESS);

}
```

Figure- 7

In Figure-7, we created 2 different child process with *pid=fork( )* command. *getchar ()* command is just for keeping process is open. And it exits with exit (EXIT_SUCCESS) command.

4. Conclusion

As a result, we made a system call which can determine the process has root privileges or not. And if the process has root privileges, our *system call* exits with normal way. If there are a problem like being root, entering nonsense myFlag number or nonsense process ID, our program produces appropriate error massages to the linux console. Our system call made all tests correctly in demo session except killing parent process. When we killed parent process, all child processes are also killed. It was supposed to happen that when the parent process is killed, the child processes must be connected to the process zero which is parent of all child processes. We saw this problem in our demo session and learned how it should be. With all of this, we learned to write a proper *system call*. With these all reasons, we can say that the project was useful.