fall23_hw3_LKM_for_task_info

October 23, 2023

1 HW — Linux Kernel Module for Task Information

bu dokuman kismi olarak https://linux-kernel-labs.github.io/refs/heads/master/labs/device_drivers.html sitesinden faydalanilarak OS odevi olarak yazilmistir.

2 /proc File System

https://www.kernel.org/doc/html/latest/filesystems/proc.html

/proc file system, process ve kernella alakalı farklı istatistiklere ulaşılabileceğiniz bir arayüz olarak işlev görmektedir. Her bir /proc/pid ile pid idli processin istatistiklerine yada /proc/kerneldatastructure ile kerneldatastructure kısmına isim vererek ilgili bilgilerine erişebilirsiniz. mesela

```
> cat /proc/stat
cpu  2255 34 2290 22625563 6290 127 456 0 0 0
cpu0 1132 34 1441 11311718 3675 127 438 0 0 0
cpu1 1123 0 849 11313845 2614 0 18 0 0 0
intr 114930548 113199788 3 0 5 263 0 4 [... lots more numbers ...]
ctxt 1990473
btime 1062191376
processes 2915
procs_running 1
procs_blocked 0
softirq 183433 0 21755 12 39 1137 231 21459 2263

> ls /proc/irq/
0  10  12  14  16  18  2  4  6  8  prof_cpu_mask
1  11  13  15  17  19  3  5  7  9  default_smp_affinity

> ls /proc/irq/0/
smp_affinity

> cat /proc/interrupts

           CPU0       CPU1
  0:    1243498    1214548    IO-APIC-edge  timer
  1:       8949       8958    IO-APIC-edge  keyboard
  2:          0          0          XT-PIC  cascade
```

```
  5:      11286      10161    IO-APIC-edge  soundblaster
  8:          1          0    IO-APIC-edge  rtc
  9:      27422      27407    IO-APIC-edge  3c503
 12:     113645     113873    IO-APIC-edge  PS/2 Mouse
 13:          0          0          XT-PIC  fpu
 14:      22491      24012    IO-APIC-edge  ide0
 15:       2183       2415    IO-APIC-edge  ide1
 17:      30564      30414   IO-APIC-level  eth0
 18:        177        164   IO-APIC-level  bttv
NMI:    2457961    2457959
LOC:    2457882    2457881
ERR:       2155
```

```
> cat /proc/net/dev
Inter-|Receive                                                   |[...
 face |bytes    packets errs drop fifo frame compressed multicast|[...
    lo:  908188   5596     0    0    0     0          0         0 [...
  ppp0:15475140  20721   410    0    0   410          0         0 [...
  eth0:  614530   7085     0    0    0     0          0         1 [...

...] Transmit
...] bytes    packets errs drop fifo colls carrier compressed
...]  908188     5596    0    0    0     0       0          0
...] 1375103    17405    0    0    0     0       0          0
...] 1703981     5535    0    0    0     3       0          0
```

3 Linux Kernel Modülle /proc file systeme dosya eklemek

3.1 Genel Ozet

Kernel tarafında struct file_operations ve kernel 5.6dan sonra eklenen proc_ops şeklinde data structurelar tanımlanmıştır. Bu data structureların temel özellikleri okuma ve yazma yapılırken çağrılacak fonksiyonları içermesidir.

```
struct proc_ops {
    unsigned int proc_flags;
    int (*proc_open)(struct inode *, struct file *);
    ssize_t (*proc_read)(struct file *, char __user *, size_t, loff_t *);
    ssize_t (*proc_read_iter)(struct kiocb *, struct iov_iter *);
    ssize_t (*proc_write)(struct file *, const char __user *, size_t, loff_t *);
    /* mandatory unless nonseekable_open() or equivalent is used */
    loff_t  (*proc_lseek)(struct file *, loff_t, int);
    int (*proc_release)(struct inode *, struct file *);
    __poll_t (*proc_poll)(struct file *, struct poll_table_struct *);
    long    (*proc_ioctl)(struct file *, unsigned int, unsigned long);
#ifdef CONFIG_COMPAT
    long    (*proc_compat_ioctl)(struct file *, unsigned int, unsigned long);
#endif
```

```
    int (*proc_mmap)(struct file *, struct vm_area_struct *);
    unsigned long (*proc_get_unmapped_area)(struct file *, unsigned long, unsigned long, unsigr
} __randomize_layout;
```

Temelde yapacağımız, bu data structureınproc_open, proc_realese, proc_read, proc_write pointerlarına gerekli atamaları yaptıktan sonra(bunlar file uzerinde yapilacak islemlerin davranislarini belirleyecek) aşağıdaki foksiyonla /proc file systemda dosya oluşturacağız:

```
struct proc_dir_entry *proc_create(const char *name, umode_t mode, struct proc_dir_entry *parer
```

Not: device deriver yazarken farklı olarak `/dev` altında `struct cdev mydevice...` ile device file oluşturduktan sonra fileoperations tipindeki mydevice.ops.read vb üyelerine ilgili atamalar yapılır ve device_create ile device file oluşturulur (yada terminalden mknod kullanabilirsiniz.).

3.2 module ile procfs'e file ekleme/çıkarma

Daha önceki ödevde modul başlarken ve biterken hangi fonksiyonların çalıştırılabileceklerini `module_init()` ve `module_exit()` ile yapmıştık.

Burada proc file systemda dosya oluşturma kısmını `module_init()`'e; bu dosyayı kaldırma kısmınıda `module_exit()`e argüman olarak vereceğiz. Bunun için öncelikli olarak `my_module_init()` ve `my_module_exit()` şeklinde iki tane fonksiyon tanımlayalım. Bunlarda temel olarak dosya oluşturup kaldıracağız (/include/linux/proc_fs.h):

```
/* my_module.c */
#include <linux/init.h>    /* Needed for the macros */
#include <linux/kernel.h>  /* Needed for pr_info() */
#include <linux/module.h>  /* Needed by all modules */
#include <linux/proc_fs.h> /*proc_ops, proc)create, proc_remove, remove_proc_entry...*/

#define PROCF_NAME "mytaskinfo"

const struct proc_ops my_ops = {
    .proc_read = NULL,
    .proc_write = NULL,
    .proc_open = NULL,
    .proc_release = NULL,
    /*bunlari kullanarak dosya davranislarini belirleyebilirsiniz*/
};

/* This function is called when the module is loaded. */
static int __init my_module_init(void)
{
    /* creates the [/proc/procf] entry*/
    proc_create(PROCF_NAME, 0666, NULL, &my_ops);

    printk(KERN_INFO "/proc/%s created\n", PROCF_NAME);

    return 0;
}
```

```
/* This function is called when the module is removed. */
static void __exit my_module_exit(void)
{
    /* removes the [/proc/procf] entry*/
    remove_proc_entry(PROCF_NAME, NULL);

    printk(KERN_INFO "/proc/%s removed\n", PROCF_NAME);
}

/* Macros for registering module entry and exit points.
 */
module_init(my_module_init);
module_exit(my_module_exit);

MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("My Task Info Module");
MODULE_AUTHOR("kendi isminiz");
```

Bu adımdan sonra sudo insmod ile modülü yüklediğinizde /proc fs de kendi oluşturduğunuz dosyayı görebilmeniz gerekiyor.

```
$ ls /proc/mytaskinfo
/proc/mytaskinfo
```

3.3 Olusturdugumuz file'in open ve closeda yapacaklarini belirleme

[/proc/procf] olusturdugumuz dosya acildiginda ve kapandiginda sistem defualtlarindan farkli olarak ne yapilacagini belirleyebiliriz. Bunun icin yukaridaki proc_ops data structure'inda tanimli pointerlara uygun olarak; bizde asagidaki fonksiyon tanimlamalarini kullanacagiz

```
int my_open(struct inode *inode, struct file *file)
{
 printk(KERN_INFO "my_ropen() for /proc/%s \n", PROCF_NAME);
 return 0;
}
int my_release(struct inode *inode, struct file *file)
{
  printk(KERN_INFO "my_release() for /proc/%s \n", PROCF_NAME);
  return 0;
}
```

Yukarida yazdigimiz module'de eger asagidaki degisikligi yaparsak

```
const struct proc_ops my_ops = {
    .proc_read = NULL,
    .proc_write = NULL,
    .proc_open = my_open,
    .proc_release = my_release,
    /*bunlari kullanarak dosya davranislarini belirleyebilirsiniz*/
```

```
};
```

O zaman my_open ve my_relase fonksiyonlari [/proc/mytaskinfo] uzerinde yapilacak open() ve close() islemlerinde cagrilacak. Bunu gormek icin modulu derleyip ve yukledikten sonra

```
$make
$sudo insmod mytaskinfo.ko
```

bir tane user_test.c programi yazalim:

```
/** user_test.c
 *
 */
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
int main()
{
    int fd = open("/proc/mytaskinfo", O_RDWR);
    if (fd == -1)
    {
        printf("Couldn't open file\n");
        return -1;
    }
    close(fd);
    return 0;
}
```

Bu programi calistirdiktan sonra `bash $sudo dmesg` ile loga bakarak printk ile yukarida belirlemis oldugumuz mesajlarin yazildigini teyit edebilirsiniz.

3.4 Oluşturduğumuz file'dan read ve write yapma

Dikkat ettiyseniz yukarıda `.proc_read = NULL, .proc_write = NULL,` şeklinde başlatıldı. Buraya atadığımız değerler /proc/my_taskinfo dan read/write yapıldığında çağrılıyor. proc_read ve write aşağıdaki şekilde tanımlanmış:

```
ssize_t (*proc_read)(struct file *, char __user *, size_t, loff_t *);
ssize_t (*proc_write)(struct file *, const
char __user *, size_t, loff_t *);
```

Bunun için proc_read ve proc_write pointerlarının tiplerine uygun olarak iki fonksiyon tanımlamamız gerekiyor.

```
ssize_t my_read(struct file *file, char __user *usr_buf, size_t size, loff_t *offset)
{


}
```

Yukaridaki fonksiyona baktigimiz zaman parametrelerinde bulunan >- file: kullanilan dosyayi(daha sonra bunu kullanarak datasini vs belirleyecegiz) >- usr_buf: kullanici tarafi bufferi >- size: bu

bufferin size'ini >- *offset: en son kernel tarafindan kac karakter okundugu (bu bir nevi file cursor oluyor, bu degerin guncellenmesi my_read icerisinde yapilacak. Baslangicta deger 0)

3.5 Kernel space'den User Space'e data kopyalama

[/proc/mytaskinfo] uzerinden okuma islemini hem terminal üzerinden hemde herhangi bir dille yazılan user programı ile yapabiliriz. Her nasıl olursa olsun sonuçta yazdığımız modul kernel'ın bir parçası olduğu için kernel space'den user space'e (yada aksi yönde) kopyalama yapmamız gerekiyor. Bunun için system call yazarken kullanmış olduğumuz aşağıdaki fonksiyonları kullanacağız(linux/uaccess.h):

```
unsigned long copy_to_user (void __user *to,
    const void *from,
    unsigned long n);

unsigned long copy_from_user (void *to,
    const void __user *from,
    unsigned long n);
```

Okurken `strncpy_from_user` da kullanabilirsiniz.

/proc/file dan read yapma

```
#define MYBUF_SIZE 256
static ssize_t my_read(struct file *file, char __user *usr_buf, size_t size, loff_t *offset)
{

    char buf[MYBUF_SIZE] = {'\0'};
    int len = sprintf(buf, "Hello World\n");

    /* copy len byte to userspace usr_buf
     Returns number of bytes that could not be copied.
     On success, this will be zero.
    */
    if(copy_to_user(usr_buf, buf, len)) return -EFAULT;

    return len; /*the number of bytes copied*/
}
```

Tipik olarak oluşturduğumuz dosyadan okuma yapmak için örnek olarak:

```
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>

int main()
{

    int fd = open("/proc/mytaskinfo", O_RDWR);
    if (fd == -1)
```

```
    {
        printf("Couldn't open file\n");
        return -1;
    }
    char buf[256];

    int r = read(fd, &buf, 256);
    printf("return value: %d\n buf: %.256s\n", r, buf);
    close(fd);
    return 0;
}
```

Yukarıdaki örnekte hem user programı buffer size'i yeterli büyüklükte olduğu için, kopyalama işlemi tek seferde bitti. Bazen bu durum öyle olmayabilir ve user programı normal bir dosyadan belirli büyüklüklerde birden fazla cagri ile okuma yapmak isteyebilir.

1. Bu durumda `*offset` degerini her seferinde set ederek, sonraki cagrilarda kaldigimiz yerden okumaya devam etmeliyiz (yani `buf+ *offset` degerinden).

```
if (copy_to_user(usr_buf, buf + *offset, len))
    return -EFAULT;
*offset = *offset + len; /*offset(cursor position) degeri guncellendi*/
```

2. Yine kullanicinin vermis oldugu `size` ile data size'ni karsilastirilip buffer overflow yapmamak icin kontrol etmeliyiz

```
int len = min(len - *offset, size);
if (len <= 0)
    return 0; /*end of file*/
```

Yukaridaki degisiklikleri yaptiktan sonra, make ve insmod ile modulunuzu tekrar yukleyerek mesela user_test2.c ile test ediniz:

```
/** user_test2.c
 *
 */
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
int main()
{

    int fd = open("/proc/my_taskinfo", O_RDWR);
    if (fd == -1)
    {
        printf("Couldn't open file\n");
        return -1;
    }
    char buf;
    int r;
```

```
    /* On success, read returns the number of bytes read
     (zero indicates end of file)*/
    while ((r = read(fd, &buf, 1)) > 0)
    {
        printf("return value: %d\n buf: %c\n", r, buf);
    }
    close(fd);
    return 0;
}
```

3. Dikkat ederseniz her cagrida datayi sprintf ile buffera alma sonra size'ini bulma gibi gereksiz islemler yaptik. Bu durumu ortadan kaldirmak icin, my_openda dosya datasi atayarak bu dataya my_readde vs tekrardan erisebiliriz.

- Oncelikle bir tane data structure tanimlayalim:

```
struct my_data
{
    int size;
    char *buf; /* my data starts here */
};
```

- Sonra my_open() da sprintfle daha once my_read icerisinde yapmis oldugumuz kismi, my_open'a alalim:

```
static int my_open(struct inode *inode, struct file *file)
{
    struct my_data *my_data = kmalloc(sizeof(struct my_data) * MYBUF_SIZE, GFP_KERNEL);
    my_data->buf = kmalloc(sizeof(char) * MYBUF_SIZE, GFP_KERNEL);
    my_data->size = MYBUF_SIZE;
    my_data->size = sprintf(my_data->buf, "Hello World\n");
    /* validate access to data */
    file->private_data = my_data;
    return 0;
}
```

Yukaridaki kodda kmalloc, malloc'a benzer olarak kernel space calismaktadir. struct file pointer kullanilarak dosyamizin datasini vs belirleyebiliyoruz. Yani my_read()'de okuma islemini `file->private_data` uzerinden yapacagiz.

```
static ssize_t my_read(struct file *file, char __user *usr_buf, size_t size, loff_t *offset)
{

    struct my_data *my_data = (struct my_data *)file->private_data;

    int len = min((int)(my_data->size - *offset), (int)size);
    if (len <= 0)
        return 0; /*end of file*/

    if (copy_to_user(usr_buf, my_data->buf + *offset, len))
        return -EFAULT;
```

```
    *offset = *offset + len;

    return len; /*the number of bytes copied*/
}
```

3.6 Write(User spaceden kernel space'e kopyalama)

my_read()'e benzer olarak my_write()'ida asagidaki sekilde tanimlayabiliriz:

```
ssize_t my_write(struct file *file, const char __user *usr_buf, size_t size, loff_t *offset)
{
    char *buf = kmalloc(size + 1, GFP_KERNEL);

    /* copies user space usr_buf to kernel buffer */
    if (copy_from_user(buf, usr_buf, size))
    {
        printk(KERN_INFO "Error copying from user\n");
        return -EFAULT;
    }
    /* *offset += size;/*yine offseti bazi durumlarda set etmeniz vs gerekebilir, user tekrar

    buf[size] = '\0';

    printk(KERN_INFO "the value of kernel buf: %s", buf);

    kfree(buf);
    return size;
}
```

4 Yapılması İstenenler

4.1 1.

/proc file systemde **mytaskinfo** isminde bir dosya oluşturarak daha önce verilen process state grubunu kullanarak, verilen durumdaki processleri ve bunların çalışma zamanlarını listeleyen bir module oluşturmanız istenmektedir. Process state grupları aşağıdaki şekilde belirlenmiştir:

```
 "R (running)",     /* 0x00 */
 "S (sleeping)",        /* 0x01 */
 "D (disk sleep)",  /* 0x02 */
 "T (stopped)",     /* 0x04 */
 "t (tracing stop)",    /* 0x08 */
 "X (dead)",        /* 0x10 */
 "Z (zombie)",      /* 0x20 */
 "P (parked)",      /* 0x40 */
 "I (idle)",        /* 0x80 */
```

Çalışırken sadece baştaki karakteri kullanacağız: Mesela

```
$ echo "R" > /proc/mytaskinfo
```

```
$ cat /proc/mytaskinfo
process running times
1.pid = ... state = ... utime = ..., stime = ..., utime+stime = ..., vruntime = ...
2.pid = ... state = ... utime = ..., stime = ..., utime+stime = ..., vruntime = ...
...
```

Linux kernelde, for_each_process() macro kullanarak sistemdeki mevcut taskler üzerinde iterasyon oluşturabilirsiniz:

```
struct task_struct *task;
for_each_process(task) {
/* on each iteration task points to the next task */
}
```

Burada, task_struct <linux/sched.h> de tanimli bir data structuredir. İçerisinde bir çok bilgiyi barındırmasına rağmen biz bu ödevde sadece aşağıda verilen memberlarını kullanacağız:

```
struct task_struct{
    ...
    unsigned int            __state;
    pid_t pid;
    u64 utime;
    u64 stime;
    struct scheduled_entry se;
    ...
}
```

Yine herhangi bir process p icin `vruntime` degerine, `p->se.vruntime` ile erisebilirsiniz.

4.1.1 Task state'in okunmasi(bitler halinde verilmekte)

Burada verilen `__state` aşağıdaki şekilde tanımlanmıştır(https://elixir.bootlin.com/linux/latest/source/include/li
):

```
* Used in tsk->state: */
#define TASK_RUNNING            0x00000000
#define TASK_INTERRUPTIBLE      0x00000001
#define TASK_UNINTERRUPTIBLE        0x00000002
#define __TASK_STOPPED          0x00000004
#define __TASK_TRACED           0x00000008
/* Used in tsk->exit_state: */
#define EXIT_DEAD           0x00000010
#define EXIT_ZOMBIE         0x00000020
#define EXIT_TRACE          (EXIT_ZOMBIE | EXIT_DEAD)
/* Used in tsk->state again: */
#define TASK_PARKED         0x00000040
#define TASK_DEAD           0x00000080
#define TASK_WAKEKILL           0x00000100
#define TASK_WAKING         0x00000200
#define TASK_NOLOAD         0x00000400
#define TASK_NEW            0x00000800
```

```
#define TASK_RTLOCK_WAIT        0x00001000
#define TASK_FREEZABLE          0x00002000
#define __TASK_FREEZABLE_UNSAFE       (0x00004000 * IS_ENABLED(CONFIG_LOCKDEP))
#define TASK_FROZEN             0x00008000
#define TASK_STATE_MAX          0x00010000

#define TASK_ANY                (TASK_STATE_MAX-1)
```

Buradaki bilgileri okumak için aşağıdaki şekilde bir array tanımlayınız (https://elixir.bootlin.com/linux/latest/source/fs/proc/array.c#L126):

```
static const char * const task_state_array[] = {

    /* states in TASK_REPORT: */
    "R (running)",       /* 0x00 */
    "S (sleeping)",      /* 0x01 */
    "D (disk sleep)",    /* 0x02 */
    "T (stopped)",       /* 0x04 */
    "t (tracing stop)", /* 0x08 */
    "X (dead)",      /* 0x10 */
    "Z (zombie)",        /* 0x20 */
    "P (parked)",        /* 0x40 */

    /* states beyond TASK_REPORT: */
    "I (idle)",      /* 0x80 */
};
```

Sonra sched.h te tanımlı `task_state_index()` fonksiyonunu kullanarak arraydeki string karşılığını bulunuz

```
  return task_state_array[task_state_index(tsk)];
```

4.2 2.

Bu programi test eden bir tane user_test.c yaziniz.user_test.c Dosyayi bir defa actiktan sonra, her gruptan processi liste halinde yazdırmalı.

5 Teslim

1. tüm c dosyalarınızı
2. Aşağıdaki çalıştırmanın terminal görüntüsünü

```
$ echo "R" > /proc/mytaskinfo & cat /proc/mytaskinfo
$ echo "S" > /proc/mytaskinfo & cat /proc/mytaskinfo
$ echo "D" > /proc/mytaskinfo & cat /proc/mytaskinfo
$ echo "T" > /proc/mytaskinfo & cat /proc/mytaskinfo
$ echo "t" > /proc/mytaskinfo & cat /proc/mytaskinfo
$ echo "X" > /proc/mytaskinfo & cat /proc/mytaskinfo
$ echo "Z" > /proc/mytaskinfo & cat /proc/mytaskinfo
```

```
$ echo "P" > /proc/mytaskinfo & cat /proc/mytaskinfo
$ cat /proc/mytaskinfo
```

3. Ve usertest.c nin çalışma görüntüsünü

6 Değerlendirme

1. 30 puan: write kısmı >. private_datanin guncellenmesi gerekiyor
2. 40 puan: read kısmı 40
3. 30 puan: user_test.c
4. -20 puana kadar cat output olmaması yada eksik olması
5. diğer her bir hata -10 puan, küçük hatalar -5 puan
6. -10 puan warnings ve kodlama standartları.