Санкт-Петербургский политехнический университет Петра Великого
Кафедра компьютерных систем и программных технологий

Отчет по дисциплине
«Проектирование ОС и их компонентов»

Анализ исходного кода системных вызовов (sys_rt_sigaction, sys_rt_sigprocmask, sys_rt_sigpending) и их частичная замена/перехват

Работу выполнил студент группы №: 13541/3 Чеботарёв М. М.
Работу принял преподаватель: _____ Душутина Е. В.

Санкт-Петербург
2017 г.

Используемая система и версия ядра

a) Linux

```
michael@michael-LIFEBOOK-AH531:~$ lsb_release  -a
No LSB modules are available.
Distributor ID:          Ubuntu
Description:             Ubuntu 16.04.1 LTS
Release:                 16.04
Codename:                xenial


michael@michael-LIFEBOOK-AH531:~$ cat /proc/version
Linux version 4.4.0-38-generic (buildd@lgw01-58) (gcc version 5.4.0 20160609 (Ubuntu
5.4.0-6ubuntu1~16.04.2) ) #57-Ubuntu SMP Tue Sep 6 15:42:33 UTC 2016
```

Цель работы

Проанализировать работу системных вызовов Linux (3-4 вызова по варианту). Изучить их исходный код. Показать на примерах работу данных системных вызовов. Перехватить и модифицировать системные вызовы.

Ход работы:

1. Назначение, функциональность и параметры системных вызовов

Данный вариант предполагает рассмотрение работы следующих системных вызовов:

1. sys_rt_sigaction – установка процессом собственного обработчика события;
2. sys_rt_sigprocmask – изменение текущего списка сигналов, который необходимо блокировать;
3. sys_rt_sigpending – проверяет (предоставляет) набор сигналов, которые получены потоком/процессом, но находятся в режиме ожидания обслуживания (замаскированы).

1.1. Системный вызов «sys_rt_sigaction»

Название

sys_rt_sigaction – установка процессом собственного обработчика события.

Синтаксис

```
long sys_rt_sigaction (   int sig,
                          const struct sigaction __user * act,
                          struct sigaction __user * oact,
                          size_t sigsetsize);
```

Аргументы

sig	сигнал, который будет обрабываться;
act	новый обработки
oact	структура для хранения предыдущего обработчика события;
sigsetsize	размер типа данных sigset_t

***Примечание*:**
Структура sigaction может быть определена следующим образом:

```
struct sigaction {
               void     (*sa_handler)(int);
               void     (*sa_sigaction)(int, siginfo_t *, void *);
               sigset_t   sa_mask;
               int        sa_flags;
               void     (*sa_restorer)(void);
          };
```

где:

sa_handler	Указатель на функцию обработчик сигнала или константа. sa_handler определят действие, которое должно производиться по сигналу, это может быть действие по умолчанию (SIG_DFL или SIG_IGN) или своя собственная функция-обработчик, принимающая 1 аргумент – номер сигнала;
sa_mask	Маска сигналов, блокируемых на время вызова обработчика.
sa_flags	Дополнительные флаги.

1.2. Системный вызов «sys_rt_sigprocmask»

Название
sys_rt_sigprocmask – изменение текущего списка сигналов, который необходимо блокировать.

Синтаксис

```
long sys_rt_sigprocmask (   int how,
                            sigset_t __user * nset,
                            sigset_t __user * oset,
                            size_t sigsetsize);
```

Аргументы

how	сигнал, который будет добавляться/удаляться из макски;
nset	указатель на **новую** маску, которая хранит актуальную версию
oset	указатель на **предыдущую версию маски сигнала** (если он не NULL)
sigsetsize	размер типа данных sigset_t

1.3. Системный вызов «sys_rt_sigpending»

Название
sys_rt_sigprocmask – проверяет (предоставляет) набор сигналов, которые получены потоком/процессом, но находятся в режиме ожидания обслуживания (замаскированы).

Синтаксис

```
long sys_rt_sigpending (    sigset_t __user * uset,
                            size_t sigsetsize);
```

Аргументы

uset	указатель на набор сигналов в режиме ожидания;
sigsetsize	размер типа данных sigset_t или более

2. Сравнить версии заданных сист. вызовов в ядрах: 2.6 и 3.x, 4.x

2.1. Системный вызов «sys_rt_sigaction»

2.1.1. Kernel 2.6.11

В ядре данной версии системный вызов определен несколько раз, в 4 файлах:

- arch/alpha/kernel/signal.c
- arch/sparc/kernel/sys_sparc.c,
- arch/sparc64/kernel/sys_sparc.c
- kernel/signal.c

2.1.1.a. Файл «arch/alpha/kernel/signal.c» (исходный, с ним будем сравнивать)

Листинг 2.1.1.a. «arch/alpha/kernel/signal.c»

```
#ifdef __ARCH_WANT_SYS_RT_SIGACTION
asmlinkage long
sys_rt_sigaction(int sig,
		 const struct sigaction __user *act,
		 struct sigaction __user *oact,
		 size_t sigsetsize)
{
	struct k_sigaction new_sa, old_sa;
	int ret = -EINVAL;

	/* XXX: Don't preclude handling different sized sigset_t's.  */
	if (sigsetsize != sizeof(sigset_t))
		goto out;

	if (act) {
		if (copy_from_user(&new_sa.sa, act, sizeof(new_sa.sa)))
			return -EFAULT;
	}

	ret = do_sigaction(sig, act ? &new_sa : NULL, oact ? &old_sa : NULL);

	if (!ret && oact) {
		if (copy_to_user(oact, &old_sa.sa, sizeof(old_sa.sa)))
			return -EFAULT;
	}
out:
	return ret;
}
#endif /* __ARCH_WANT_SYS_RT_SIGACTION */
```

2.1.1.b. Сравнение определений «arch/alpha/kernel/signal.c» и «kernel/signal.c»

Определение функции в файле «arch/alpha/kernel/sys_sparc.c» отличается следующим:

1. отсутствует аргумент ***restorer;**
2. **переменная ret инициализируется значением -EINVAL;**
3. **рефакторинг кода:** «new_sa, old_sa;» переименованы в «new_ka, old_ka;»;
4. в первом файле аргумент **ret** участвует в инициализцаии структуры **new_ka.**

Листинг 2.1.1.b. «arch/alpha/kernel/signal.c»

```
...
asmlinkage long
sys_rt_sigaction(int sig,
		 const struct sigaction __user *act,
		 struct sigaction __user *oact,
		/*void __user *restorer,*/
```

```
               size_t sigsetsize)
{
       struct k_sigaction new_sa, old_sa;
       int ret = -EINVAL;

       /* XXX: Don't preclude handling different sized sigset_t's.  */
       if (sigsetsize != sizeof(sigset_t))
              goto out;

       if (act) {
              /* new_ka.ka_restorer = restorer;*/
              if (copy_from_user(&new_sa.sa, act, sizeof(new_sa.sa)))
                     return -EFAULT;
       }
...
```

2.1.1.c. Сравнение определений «arch/alpha/kernel/signal.c» и «arch/sparc64/kernel/sys_sparc.c»

Определение функции в файле «arch/sparc64/kernel/sys_sparc.c» **отличается только порядком аргументов** (последние 2 аргумента поменяны местами)

Листинг 2.1.1.c. «arch/sparc64/kernel/sys_sparc.c»

```
asmlinkage long
sys_rt_sigaction(int sig,
              const struct sigaction __user *act,
              struct sigaction __user *oact,
              void __user *restorer,
              size_t sigsetsize)
{
...
}
```

2.1.1.d. Сравнение определений «arch/alpha/kernel/signal.c» и «arch/sparc/kernel/sys_sparc.c»

Определение функции в файле «arch/sparc/kernel/sys_sparc.c» **отличается порядком аргументов** (последние 2 аргумента поменяны местами), и **установкой доп. флага:**

Листинг 2.1.1.d. «arch/sparc/kernel/sys_sparc.c»

```
asmlinkage long
sys_rt_sigaction(int sig,
              const struct sigaction __user *act,
              struct sigaction __user *oact,
              void __user *restorer,
              size_t sigsetsize)
{
...
       /* All tasks which use RT signals (effectively) use
        * new style signals.
        */
       current->thread.new_signal = 1;

...
       return ret;
}
```

Примечание: данный флаг указывает всем процессам, которые используют real-time сигналы обязательно использовать новые сигналы.

В файле **kernel/signal.c** присутствуют следующие комментарии:

```
/*
 *  linux/kernel/signal.c
 *
 *  Copyright (C) 1991, 1992  Linus Torvalds
 *
 *  1997-11-02  Modified for POSIX.1b signals by Richard Henderson
 *
 *  2003-06-02  Jim Houston - Concurrent Computer Corp.
 *          Changes to use preallocated sigqueue structures
 *          to allow signals to be sent reliably.
 */
```

Так как последний раз данный файл модифицировался в 2003 году – сначала сравним версию 2.6 ядра с версией 4.1, и если они отличаются – сравним так же с версией ядра 3.

2.1.2. Kernel 4.12-rc3

В ядре данной версии системный вызов определен будем интересоваться только архитектурой i386. В файле «kernel/signal.c» представлено определение данного системной вызова.

Листинг 2.1.2.a. «kernel/signal.c»

```
#ifndef CONFIG_ODD_RT_SIGACTION
/**
 *  sys_rt_sigaction - alter an action taken by a process
 *  @sig: signal to be sent
 *  @act: new sigaction
 *  @oact: used to save the previous sigaction
 *  @sigsetsize: size of sigset_t type
 */
SYSCALL_DEFINE4(rt_sigaction, int, sig,
              const struct sigaction __user *, act,
              struct sigaction __user *, oact,
              size_t, sigsetsize)
{
...
out:
       return ret;
}
```

При сравнении с версий установлено, что версии ядра 2.6 и 4.13 отличаются только способом определения системного вызова **rt_sigaction, определение метода за последние 14 лет не изменялось.**

2.2. Системный вызов «sys_rt_sigprocmask»

2.2.1. Kernel 2.6.11

Листинг 2.2.1.a. «kernel/signal.c»

```
/*
 * The OSF/1 sigprocmask calling sequence is different from the
 * C sigprocmask() sequence..
 *
 * how:
 * 1 - SIG_BLOCK
 * 2 - SIG_UNBLOCK
 * 3 - SIG_SETMASK
 *
 * We change the range to -1 .. 1 in order to let gcc easily
 * use the conditional move instructions.
 *
 * Note that we don't need to acquire the kernel lock for SMP
 * operation, as all of this is local to this thread.
 */
asmlinkage unsigned long
do_osf_sigprocmask(int how, unsigned long newmask, struct pt_regs *regs)
{
	unsigned long oldmask = -EINVAL;

	if ((unsigned long)how-1 <= 2) {
		long sign = how-2;		/* -1 .. 1 */
		unsigned long block, unblock;

		newmask &= _BLOCKABLE;
		spin_lock_irq(&current->sighand->siglock);
		oldmask = current->blocked.sig[0];

		unblock = oldmask & ~newmask;
		block = oldmask | newmask;
		if (!sign)
			block = unblock;
		if (sign <= 0)
			newmask = block;
		if (_NSIG_WORDS > 1 && sign > 0)
			sigemptyset(&current->blocked);
		current->blocked.sig[0] = newmask;
		recalc_sigpending();
		spin_unlock_irq(&current->sighand->siglock);

		regs->r0 = 0;			/* special no error return */
	}
	return oldmask;
}
```

2.2.2. Kernel 3.12.74 & Kernel 4.12-rc3

Код представленный **в ядрах 3.* и 4.* версий абсолютно полностью совпадают** для системного вызова `rt_sigprocmask`.

Листинг 2.2.3.c. «kernel/signal.c»

```
/**
 *  sys_rt_sigprocmask - change the list of currently blocked signals
 *  @how: whether to add, remove, or set signals
 *  @nset: stores pending signals
 *  @oset: previous value of signal mask if non-null
```

```
 *  @sigsetsize: size of sigset_t type
 */
SYSCALL_DEFINE4(rt_sigprocmask, int, how, sigset_t __user *, nset,
             sigset_t __user *, oset, size_t, sigsetsize)
{
       sigset_t old_set, new_set;
       int error;

       /* XXX: Don't preclude handling different sized sigset_t's.  */
       if (sigsetsize != sizeof(sigset_t))
              return -EINVAL;

       old_set = current->blocked;

       if (nset) {
              if (copy_from_user(&new_set, nset, sizeof(sigset_t)))
                     return -EFAULT;
              sigdelsetmask(&new_set, sigmask(SIGKILL)|sigmask(SIGSTOP));

              error = sigprocmask(how, &new_set, NULL);
              if (error)
                     return error;
       }

       if (oset) {
              if (copy_to_user(oset, &old_set, sizeof(sigset_t)))
                     return -EFAULT;
       }

       return 0;
}
```

2.3. Системный вызов «sys_rt_sigpending»

2.3.1. Kernel 2.6.11

Листинг 2.2.1.a. «kernel/signal.c»

```
long do_sigpending(void __user *set, unsigned long sigsetsize)
{
       long error = -EINVAL;
       sigset_t pending;

       if (sigsetsize > sizeof(sigset_t))
              goto out;

       spin_lock_irq(&current->sighand->siglock);
       sigorsets(&pending, &current->pending.signal,
                 &current->signal->shared_pending.signal);
       spin_unlock_irq(&current->sighand->siglock);

       /* Outside the lock because only this thread touches it.  */
       sigandsets(&pending, &current->blocked, &pending);

       error = -EFAULT;
       if (!copy_to_user(set, &pending, sigsetsize))
              error = 0;
```

```
out:
	return error;
}

...

asmlinkage long
sys_rt_sigpending(sigset_t __user *set, size_t sigsetsize)
{
	return do_sigpending(set, sigsetsize);
}
```

2.3.2. Kernel 3.12.74 & Kernel 4.12-rc3

Листинг 2.2.1.a. «kernel/signal.c»

```
static int do_sigpending(void *set, unsigned long sigsetsize)
{
	if (sigsetsize > sizeof(sigset_t))
		return -EINVAL;

	spin_lock_irq(&current->sighand->siglock);
	sigorsets(set, &current->pending.signal,
		  &current->signal->shared_pending.signal);
	spin_unlock_irq(&current->sighand->siglock);

	/* Outside the lock because only this thread touches it.  */
	sigandsets(set, &current->blocked, set);
	return 0;
}

...

SYSCALL_DEFINE2(rt_sigpending, sigset_t __user *, uset, size_t, sigsetsize)
{
	sigset_t set;
	int err = do_sigpending(&set, sigsetsize);
	if (!err && copy_to_user(uset, &set, sigsetsize))
		err = -EFAULT;
	return err;
}
```

Отличие 2 и 3-4 версий состоит только в том, что вызов copy_to_user **в версии 2.6 производится внутри** rt_sigpending, а в случае версий ядер 3-4 внутри вызываемой функции do_sigpending. Алгоритм работы у обоих методов идентичный. Так же отличием является способ объявления функции системного вызова.

Макрос asmlinkage в определении этих функций говорит о двух вещах:

- функция не может принимать аргументы из регистров процессора; она принимает их из стека;
- функция может компоноваться в ассемблерном виде.

Так же мы видим, что как ранее говорилось системные вызовы проверяют свои аргументы (если это необходимо) и затем пересылают их следующей функции do_*(), которая реализует всю функциональность системного вызова (метод `do_sigpending`).

3. Примеры использования системных вызовов

Написаны программы на языке С, которые используют данные системные вызовы:

Программа №1. sigthr2.cpp

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>

void term_handler(int i) {
	printf ("Interruption!\n");
	exit(EXIT_SUCCESS);
}

int main(int argc, char ** argv) {
	usleep(1);
	struct sigaction sa;
	sigset_t newset;
	sigemptyset(&newset);
	sigaddset(&newset, SIGHUP);
	sigprocmask(SIG_BLOCK, &newset, 0);
	sa.sa_handler = term_handler;
	sigaction(SIGINT, &sa, 0);
	printf("My pid is %i\n", getpid());
	printf("Waiting...\n");
	while(true)
		sleep(1);

	usleep(17);
	return EXIT_FAILURE;
}
```

Результат исполнения (терминал):

```
osboxes@osboxes:~/Desktop/lab6-systemCall$ strace -o result2.txt ./sigthr2
My pid is 4252
Waiting...
^CInterruption!
```

Результат исполнения(strace):

```
...
set_thread_area({entry_number:-1 -> 6, base_addr:0xb756c940, limit:1048575,
seg_32bit:1, contents:0, read_exec_only:0, limit_in_pages:1, seg_not_present:0,
useable:1}) = 0
mprotect(0xb7716000, 8192, PROT_READ)   = 0
mprotect(0x8049000, 4096, PROT_READ)    = 0
mprotect(0xb7756000, 4096, PROT_READ)   = 0
munmap(0xb771c000, 80298)               = 0
rt_sigprocmask(SIG_BLOCK, [HUP], NULL, 8) = 0
rt_sigaction(SIGINT, {0x80485dd, [],
SA_STACK|SA_NOCLDSTOP|SA_NOCLDWAIT|0x486e8}, NULL, 8) = 0
getpid()                                = 4252
fstat64(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 0), ...}) = 0
mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
0xb772f000
write(1, "My pid is 4252\n", 15)        = 15
write(1, "Waiting...\n", 11)            = 11
rt_sigprocmask(SIG_BLOCK, [CHLD], [HUP], 8) = 0
rt_sigaction(SIGCHLD, NULL, {SIG_DFL, [], 0}, 8) = 0
```

```
rt_sigprocmask(SIG_SETMASK, [HUP], NULL, 8) = 0
nanosleep({1, 0}, 0xbfd2a2fc)           = 0
rt_sigprocmask(SIG_BLOCK, [CHLD], [HUP], 8) = 0
rt_sigaction(SIGCHLD, NULL, {SIG_DFL, [], 0}, 8) = 0
rt_sigprocmask(SIG_SETMASK, [HUP], NULL, 8) = 0
nanosleep({1, 0}, {0, 901774199})       = ? ERESTART_RESTARTBLOCK (Interrupted by
signal)
--- SIGINT {si_signo=SIGINT, si_code=SI_KERNEL} ---
write(1, "Interruption!\n", 14)         = 14
exit_group(0)                           = ?
+++ exited with 0 +++
```

Анализ исполнения:

- Данная команда **rt_sigprocmask(SIG_BLOCK, [HUP], NULL, 8) = 0** устанавливает маску для всего процесса, и маска равна **[HUP].** Так передается указание, что данный набор сигналов (он только 1) необходимо блокировать **SIG_BLOCK.**

- Данный блок команд исполняется **каждый раз после пробуждения процесса:**

  rt_sigprocmask(SIG_BLOCK, [CHLD], [HUP], 8) = 0
  rt_sigaction(SIGCHLD, NULL, {SIG_DFL, [], 0}, 8) = 0
  rt_sigprocmask(SIG_SETMASK, [HUP], NULL, 8) = 0
  nanosleep({1, 0}, 0xbfd2a2fc)

Первая команда устанавливает новую маску сигналов, а вторая указывает как реагировать на сигнал **SIGCHLD,** который посылается процесса для изменения статуса. Т.к. процесса «спит». Третья команда возвращает маску, которая задана в код. Таким образом процесс реагирует на сигналы только в тем моменты, пока не спит.

Программа №2. sigthr.cpp

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <signal.h>
#include <pthread.h>
int quitflag = 0;
sigset_t mask;
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t  wait = PTHREAD_COND_INITIALIZER;

// функция потока
void* threadfunc ( void* data ) {
   int signo;
   while( true ) {
      // ожидание сигнала
      if( sigwait( &mask, &signo ) != 0 )
         perror( "sigwait:" ), exit( EXIT_FAILURE );

      // проверка, какой сигнал поступил
      switch( signo ) {
         case SIGINT:     // сигнал прерывание (Ctrl+C)
            printf( "  ... signal SIGINT\n" );
            break;
```

```
         case SIGQUIT:    // сигнал завершения с дампом памяти (Ctrl+/)
            printf( "  ... signal SIGQUIT\n" );
            pthread_mutex_lock( &lock );
            quitflag = 1;
            pthread_mutex_unlock( &lock );
            pthread_cond_signal( &wait );
            return NULL;
         default:
            printf( "undefined signal %d\n", signo ), exit( EXIT_FAILURE );
      }
   };
};

int main() {
   printf( "process started with PID=%d\n", getpid() );
   // настройка маски сигналов
   sigemptyset( &mask );
   sigaddset( &mask, SIGINT );
   sigaddset( &mask, SIGQUIT );
   sigset_t oldmask;

   // блокировка сигналов для основного потока
   if( sigprocmask( SIG_BLOCK, &mask, &oldmask ) < 0 )
      perror( "signals block:" ), exit( EXIT_FAILURE );

   // создание потока-обработки сигналов
   pthread_t tid;
   if( pthread_create( &tid, NULL, threadfunc, NULL ) != 0 )
      perror( "thread create:" ), exit( EXIT_FAILURE ); ;
   pthread_mutex_lock( &lock ); // блокировка мьютекса
   while( 0 == quitflag )
      pthread_cond_wait( &wait, &lock ); // ожидание выполнения условия
   pthread_mutex_unlock( &lock );
   // SIGQUIT был перехвачен, но к этому моменту снова заблокирован
   if( sigprocmask( SIG_SETMASK, &oldmask, NULL ) < 0 )
      perror( "signals set:" ), exit( EXIT_FAILURE );
   return EXIT_SUCCESS;
};
```

Результат исполнения:

```
osboxes@osboxes:~/Desktop/lab6-systemCall$ strace -o result.txt ./sigthr
process started with PID=3378
^C  ... signal SIGINT
^\  ... signal SIGQUIT
```

result.txt

```
...
futex(0xbfd83854, FUTEX_WAIT_BITSET_PRIVATE|FUTEX_CLOCK_REALTIME, 1, NULL, b750c700)
= -1 EAGAIN (Resource temporarily unavailable)
rt_sigaction(SIGRTMIN, {0xb76c27d0, [], SA_SIGINFO}, NULL, 8) = 0
rt_sigaction(SIGRT_1, {0xb76c2850, [], SA_RESTART|SA_SIGINFO}, NULL, 8) = 0
rt_sigprocmask(SIG_UNBLOCK, [RTMIN RT_1], NULL, 8) = 0
getrlimit(RLIMIT_STACK, {rlim_cur=8192*1024, rlim_max=RLIM_INFINITY}) = 0
```

```
uname({sys="Linux", node="osboxes", ...}) = 0

/* непосредственный запуск пользовательского кода */

nanosleep({0, 1000}, NULL)              = 0
fstat64(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 0), ...}) = 0
mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
0xb76ec000
write(1, "process started with PID=3378\n", 30) = 30
nanosleep({0, 2000}, NULL)              = 0
nanosleep({0, 3000}, NULL)              = 0
nanosleep({0, 4000}, NULL)              = 0
rt_sigprocmask(SIG_BLOCK, [INT QUIT], [], 8) = 0
nanosleep({0, 5000}, NULL)              = 0
mmap2(NULL, 8392704, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS|MAP_STACK, -1,
0) = 0xb6d0b000
brk(0)                                  = 0x9f86000
brk(0x9fa7000)                          = 0x9fa7000
mprotect(0xb6d0b000, 4096, PROT_NONE)   = 0
clone(child_stack=0xb750b424,
flags=CLONE_VM|CLONE_FS|CLONE_FILES|CLONE_SIGHAND|CLONE_THREAD|CLONE_SYSVSEM|CLONE_SE
TTLS|CLONE_PARENT_SETTID|CLONE_CHILD_CLEARTID, parent_tidptr=0xb750bba8,
{entry_number:6, base_addr:0xb750bb40, limit:1048575, seg_32bit:1, contents:0,
read_exec_only:0, limit_in_pages:1, seg_not_present:0, useable:1},
child_tidptr=0xb750bba8) = 3379
futex(0x804a144, FUTEX_WAIT_PRIVATE, 1, NULL) = 0
futex(0x804a120, FUTEX_WAKE_PRIVATE, 1) = 0
rt_sigprocmask(SIG_SETMASK, [], NULL, 8) = 0
exit_group(0)                           = ?
+++ exited with 0 +++
```

Анализ исполнения:

Процесс приостанавливается (или мог бы выполнять другую полезную работу) до тех пор, пока поток обработчика сигналов не сообщит о завершении.

Сигналы не могут направляться отдельным потокам процесса — сигналы направляются процессу в целом, как оболочке, обрамляющей несколько потоков. Точно так же, для каждого сигнала может быть переопределена функция-обработчик, но это переопределение действует глобально в рамках процесса.

Тем не менее, каждый из потоков (в том числе и главный поток процесса main() {...}) могут независимо **определить собственную маску реакции на сигналы**. Таким образом оказывается возможным:

- **распределить потоки**, ответственные за обработку каждого сигнала;
- **динамически изменять потоки**, в которых (в контексте которых) обрабатывается реакция на сигнал;
- **создавать обработчики сигналов в виде отдельных потоков**, специально для того предназначенных.

В данном блоке происходит установка дефолтных обработчиков (**SA_SIGINFO)** для **SIGRTMIN(минимальный диапазон пользовательских сигналов). Для**

```
rt_sigaction(SIGRTMIN, {0xb76c27d0, [], SA_SIGINFO}, NULL, 8) = 0
rt_sigaction(SIGRT_1, {0xb76c2850, [], SA_RESTART|SA_SIGINFO}, NULL, 8) = 0
rt_sigprocmask(SIG_UNBLOCK, [RTMIN RT_1], NULL, 8) = 0
```

Итог: определен ряд программ, который можно использовать в качестве тестов по перехвату/обработке системного вызова.

4. Сборка ядра

Сбора системы производится на системе Ubuntu 16.10 Yakkety Yak (Final) [2]

Установка зависимостей [1]:

```
osboxes@osboxes: ~$ sudo apt update                  # Обновить список пакетов
osboxes@osboxes: ~$ sudo apt upgrade                 # Скачать обновления
osboxes@osboxes: ~$ sudo apt -d dist-upgrade         # Скачать обновления 2
osboxes@osboxes: ~$ sudo apt dist-upgrade            # Установить обновления
osboxes@osboxes: ~$ sudo apt-get install git gcc make bc fakeroot dpkg-dev
libncurses5-dev libssl-dev
```

4.1. Получение исходного кода ядра

Переходим в home (например) командой **`cd ~.`**

Исходники ядра Ubuntu можно получить двумя способами:

- Установив архив из репозитория, с автоматическим наложением последних официальных патчей;
- скачать самую последнюю исходную версию с git

В первом случае при этом скачается пакет размером ~150 Мб в текущую папку. Чтобы получить исходники ядра, версия которого установлена на данном компьютере выполним команду:

```
osboxes@osboxes: ~$ apt-get source linux-image-`uname -r`
```

Примечание: вместо `uname -r` можно указать конкретную версию из имеющихся в репозитории. Список имеющихся в репозитории версий можно увидеть набрав команду: «apt-get source linux-image-» и нажать несколько раз клавишу Tab.

Во втором случае, возможно получить самую свежую версию ядра, которая доступна на данный момент (но это не надежно). Размер скачиваемого пакета ~500—800 Мб. Для этого можно использовать команду:

```
osboxes@osboxes: ~$ git clone git://kernel.ubuntu.com/ubuntu/ubuntu-
<releaseCodename>.git
```

Где < releaseCodename > - имя релиза, например:
git clone git://kernel.ubuntu.com/ubuntu/ubuntu-xenial.git
Или аналогичной командой:
wget https://cdn.kernel.org/pub/linux/kernel/v4.x/linux-4.8.0.tar.xz

Примечание по работе с другими ядрами: существуют ядра, работоспособность которых **в Ubuntu не гарантируется**:

- Архив с базовой версий без патчей, т.е. например «4.8.0», «4.8.17»:
  `sudo apt-get install linux-source`
- Ядра тестовой линии Mainline: http://kernel.ubuntu.com/~kernel-ppa/mainline/
- Общие ядра Linux, не адаптированные к Ubuntu: kernel.org

Вывод:
Поэтому **лучше использовать исходный код, адаптированный под Ubuntu**. Стоит отметить, что только некоторые системы (такие как **Arch**) не накладывают дополнительных ограничений на linux-ядро, поэтому в Ubuntu подходят только адаптированные версии ядра (проверено), а в Arch – все.

В рамках данной работы использован способ №1 (получить исходники уже установленного ядра, для начала). После того, как исходники были загружены (kernel 4.8.0) выполняем следующую команду по распаковке:

```
osboxes@osboxes: ~$ tar -xvf  linux-4.8.17.tar.xz
```

4.2. Получение необходимых для сборки пакетов

Для установки основных пакетов выполняем команды:

```
osboxes@osboxes: ~$ sudo apt-get update
osboxes@osboxes: ~$ sudo apt-get build-dep linux
osboxes@osboxes: ~$ sudo apt-get install kernel-package
```

Примечание: лучше лишний раз сделать update, чем не делать этого.

Далее всё зависит от того, каким способом мы хотим произвести конфигурацию ядра. Это можно сделать следующими способами:

- **config** - традиционный способ конфигурирования. Программа выводит параметры конфигурации по одному, предлагая вам установить для каждого из них свое значение. Не рекомендуется для неопытных пользователей.
- **oldconfig** - файл конфигурации создаётся автоматически, основываясь на текущей конфигурации ядра. **Рекомендуется для начинающих. (самое то)**
- **defconfig** - файл конфигурации создаётся автоматически, основываясь на значениях по-умолчанию**. (тоже не плохой вариант)**
- **menuconfig** - псевдографический интерфейс ручной конфигурации, не требует последовательного ввода значений параметров. Удобно при работе в терминале.
- **xconfig** - графический (X) интерфейс ручной конфигурации, не требует последовательного ввода значений параметров.
- **gconfig** - графический (GTK+) интерфейс ручной конфигурации, не требует последовательного ввода значений параметров. Рекомендуется для использования в среде GNOME.
- **localmodconfig** - файл конфигурации, создающийся автоматически, в который включается только то, что нужно данному конкретному устройству. При вызове данной команды большая часть ядра будет замодулирована
- **localyesconfig** - файл конфигурации, похожий на предыдущий, но здесь большая часть будет включена непосредственно в ядро. Идеальный вариант для начинающих.

В случае, если нужно использовать **config, oldconfig, defconfig, localmodconfig или localyesconfi**g, нам больше не нужны никакие дополнительные пакеты.
В случае же с оставшимися тремя вариантами (подсвечены красным) необходимо установить также дополнительные пакеты:

- Для установки пакетов, необходимых для использования **menuconfig** выполняем:
  `sudo apt-get install libncurses5-dev`
- Для установки пакетов, необходимых для использования **gconfig** выполняем:
  `sudo apt-get install libgtk2.0-dev libglib2.0-dev libglade2-dev`
- Для установки пакетов, необходимых для использования **xconfig** выполняем:
  `sudo apt-get install libqt5-dev`

Примечание: для установки не подогнанных под Ubuntu ядер (а порой и для подогнанных тоже), необходимо использовать патчи. Подробнее об этом можно узнать в источнике [3].

4.3. Настройка и сборка

На мой взгляд, самый правильный способ, начать сборку точно такого же ядра, что установлено сейчас в системе по следующим причинам:

- имеющаяся конфигурация ядра будет полностью (на самом деле почти полностью) удовлетворять сборке и шансы на успех будут выше;
- стоит начинать с простого, а собранные «бинарники» далее буду использованы как кеш, поэтому время зря потрачено не будет.

Приступаем к сборке. Переходим внутрь разархивированной папки командой:

```
osboxes@osboxes:$ cd linux-4.8.0/
```

Для создания файла конфигурации, который основывается на текущей конфигурации ядра, выполняем команду:

```
osboxes@osboxes: ~/linux-4.8.0$ make oldconfig
```

Примечание: все способы конфигурирования описаны выше.

Будет сконфигурирован файл **.config** с содержимым, напоминающем содержимое листинга 4.1.

Листинг 4.1. Файл конфигурации сборки ядра «.config»

```
#
# Automatically generated file; DO NOT EDIT.
# Linux/x86_64 4.8.17-custom Kernel Configuration
#
CONFIG_64BIT=y
CONFIG_X86_64=y
CONFIG_X86=y
CONFIG_INSTRUCTION_DECODER=y
CONFIG_OUTPUT_FORMAT="elf64-x86-64"
CONFIG_ARCH_DEFCONFIG="arch/x86/configs/x86_64_defconfig"
CONFIG_LOCKDEP_SUPPORT=y
CONFIG_STACKTRACE_SUPPORT=y
CONFIG_MMU=y
CONFIG_ARCH_MMAP_RND_BITS_MIN=28
CONFIG_ARCH_MMAP_RND_BITS_MAX=32
CONFIG_ARCH_MMAP_RND_COMPAT_BITS_MIN=8
CONFIG_ARCH_MMAP_RND_COMPAT_BITS_MAX=16

/*большая часть содержимого скрыта*/
```

Теперь запускаем компиляцию ядра командой:

```
osboxes@osboxes: ~/linux-4.8.0$ fakeroot make-kpkg -j 5 --initrd --append-to-
version=-custom kernel_image kernel_headers
```

Примечание: параметр –j указывает количество выделенных ядер +1.
ВАЖНОЕ Примечание: сборки длится от 15 минут до 4 часов (в зависимости от ПК). В моем случае использовался Core I5 2450M (все 4 потока) + SSD, компиляция заняла ~2.5часа.

4.4. Исправление возникающих ошибок

В ходе работы возникли некоторые ошибки, исправление которых так же приведено.

4.4.1. Ошибка openssl

Рисунок 1. fatal error: openssl

```
scripts/sign-file.c: 23: 30: fatal error: openssl / opensslv.h: No such file or
directory
compilation terminated.
scripts/Makefile.host: 91: recipe for target ‘scripts / sign-file’ failed
```

Решение: устанавливаем следующий пакет:

```
osboxes@osboxes: ~/linux-4.8.0$  apt-get install libssl-dev
```

4.4.2. Ошибка VBox №1

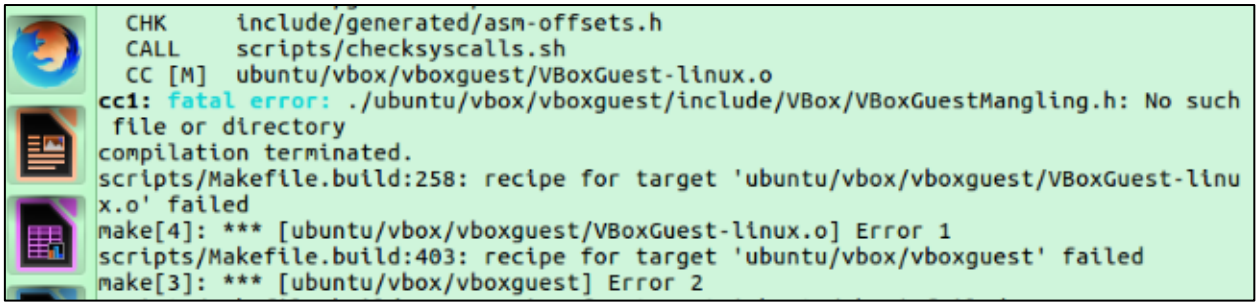

Рисунок 2. Ошибка VBox №1

```
cc1: fatal error: ./ubuntu/vbox/vboxguest/include/VBox/VBoxGuestMangling.h: No such
file or directory
compilation terminated.
scripts / Makefile.build: 258: recipe for target ‘ubuntu / vbox / vboxguest /
VBoxGuest-linux.o’ failed
```

Решение:

Создать 4 символьные ссылки

```
osboxes@osboxes: ~/linux-4.4.0# cd ubuntu/vbox
osboxes@osboxes: ~/linux-4.4.0/ubuntu/vbox# ln -s ../include ./r0drv/include
osboxes@osboxes: ~/linux-4.4.0/ubuntu/vbox# ln -s ../include ./vboxsf/include
osboxes@osboxes: ~/linux-4.4.0/ubuntu/vbox# ln -s ../include ./vboxguest/include
osboxes@osboxes: ~/linux-4.4.0/ubuntu/vbox# ln -s ../include ./vboxvideo/include
```

4.4.3. Ошибка VBox №2

```
make [4]: *** No rule to make target ‘ubuntu / vbox / vboxguest / vboxguest / r0drv /
alloc-r0drv.o’, needed by ‘ubuntu / vbox / vboxguest / vboxguest.o’. Stop.
```

Решение:

```
osboxes@osboxes ~/linux-4.4.0# cd ubuntu/vbox/vboxguest
```

```
osboxes@osboxes ~/linux-4.4.0/ubuntu/vbox/vboxguest# ln -s ../r0drv/
```

4.4.4 Ошибка сборки PIC

error 2: kernel build error: "code model kernel **does not support PIC mode**"

Решение [6]: (применить патч к linux-x.x.x/Makefile)

```
---
 Makefile | 6 ++++++
 1 file changed, 6 insertions(+)

diff --git a/Makefile b/Makefile
index dda982c..f96b174 100644
--- a/Makefile
+++ b/Makefile
@@ -608,6 +608,12 @@ endif # $(dot-config)
 # Defaults to vmlinux, but the arch makefile usually adds further targets
 all: vmlinux

+# force no-pie for distro compilers that enable pie by default
+KBUILD_CFLAGS += $(call cc-option, -fno-pie)
+KBUILD_CFLAGS += $(call cc-option, -no-pie)
+KBUILD_AFLAGS += $(call cc-option, -fno-pie)
+KBUILD_CPPFLAGS += $(call cc-option, -fno-pie)
+
 # The arch Makefile can set ARCH_{CPP,A,C}FLAGS to override the default
 # values of the respective KBUILD_* variables
 ARCH_CPPFLAGS :=
--
```

4.5. Установка нового (скомпилированного) ядра

Компиляция ядра завершается приблизительно следующим образом:

```
cp -pf debian/control debian/control.dist
k=`find /home/osboxes/linux-4.8.0/debian/linux-headers-4.8.17-custom -type f | ( while read i; do
\
          if file -b $i | egrep -q "^ELF.*executable.*dynamically linked" ; then \
            j="$j $i";                                                    \
           fi;                                                             \
        done; echo $j; )`; test -z "$k" || dpkg-shlibdeps $k;           \
        echo "Elf Files: $K" >              /home/osboxes/linux-4.8.0/debian/linux-headers-4.8.17-
custom/usr/share/doc/linux-headers-4.8.17-custom/elffiles;          \
        test -n "$k" || perl -pli~ -e 's/\$\{shlibs:Depends\}\,?//g' debian/control
test ! -e debian/control~ || rm -f debian/control~
dpkg-gencontrol -isp -DArchitecture=amd64 -plinux-headers-4.8.17-custom \
                                        -P/home/osboxes/linux-4.8.0/debian/linux-headers-4.8.17-custom/
dpkg-gencontrol: warning: -isp is deprecated; it is without effect
create_md5sums_fn () { cd $1 ; find . -type f ! -regex './DEBIAN/.*' ! -regex './var/.*'      -printf
'%P\0' | xargs -r0 md5sum > DEBIAN/md5sums ; if [ -z "DEBIAN/md5sums" ] ; then rm -f "DEBIAN/md5sums" ; fi
; } ; create_md5sums_fn                   /home/osboxes/linux-4.8.0/debian/linux-headers-4.8.17-custom
chown -R root:root                  /home/osboxes/linux-4.8.0/debian/linux-headers-4.8.17-custom
chmod -R og=rX                      /home/osboxes/linux-4.8.0/debian/linux-headers-4.8.17-custom
dpkg --build                        /home/osboxes/linux-4.8.0/debian/linux-headers-4.8.17-custom ..
dpkg-deb: building package 'linux-headers-4.8.17-custom' in '../linux-headers-4.8.17-custom_4.8.17-custom-
10.00.Custom_amd64.deb'.
cp -pf debian/control.dist          debian/control
make[2]: Leaving directory '/home/osboxes/linux-4.8.0'
make[1]: Leaving directory '/home/osboxes/linux-4.8.0'
osboxes@osboxes:~/linux-4.8.0$
```

Для установки скомпилированного ядра выполняем следующие команды:

```
osboxes@osboxes ~/linux-4.4.0# cd ..
```

Установка ядра

```
osboxes@osboxes:~$ sudo dpkg -i linux-image-4.8.17-custom_4.8.17-custom-
10.00.Custom_amd64.deb
[sudo] password for osboxes:
Selecting previously unselected package linux-image-4.8.17-custom.
(Reading database ... 225730 files and directories currently installed.)
Preparing to unpack linux-image-4.8.17-custom_4.8.17-custom-10.00.Custom_amd64.deb
...
Found linux image: /boot/vmlinuz-4.8.0-26-generic
Found initrd image: /boot/initrd.img-4.8.0-26-generic
Found memtest86+ image: /boot/memtest86+.elf
Found memtest86+ image: /boot/memtest86+.bin
done
```

Установка заголовочных файлов

```
osboxes@osboxes:~$ sudo dpkg -i linux-headers-4.8.17-custom_4.8.17-custom-
10.00.Custom_amd64.deb
Selecting previously unselected package linux-headers-4.8.17-custom.
(Reading database ... 225807 files and directories currently installed.)
Preparing to unpack linux-headers-4.8.17-custom_4.8.17-custom-10.00.Custom_amd64.deb
...
Unpacking linux-headers-4.8.17-custom (4.8.17-custom-10.00.Custom) ...
Setting up linux-headers-4.8.17-custom (4.8.17-custom-10.00.Custom) ...
Examining /etc/kernel/header_postinst.d.
```

Примечание: скомпилированное ядро лежит уровнем выше, и версия несколько отличается: вместо «заказанной» 4.8.0 получена 4.8.17 [2]. Как пел Влади Каста «замышлялось что-то такое, но вышло другое» [4]. Это не проблема, для уточнение по какой причине это происходит рекомендую заглянуть в источник.

Сначала повторно проверяем текущую версию текущего ядра:

```
osboxes@osboxes:~/linux-4.8.0$ uname -r
4.8.0-38 generic
```

Ядро обновится только после перезагрузки. Теперь необходимо перезагрузиться командой **sudo reboot**, а затем проверить версию нового ядра

```
osboxes@osboxes:~/linux-4.8.0$ uname -r
4.8.17-custom
```

5. Перехват системных вызовов

Создание модуля ядра во многом напоминает написание обычной пользовательской программы на C, кроме некоторых отличий связанных с ядром:

- Ядро не имеет доступа к стандартным библиотекам языка C. Причина этого – скорость выполнения и объем кода. Часть функций, однако, можно найти в исходниках ядра. Например, обычные функции работы со строками описаны в файле lib/string.c
- Отсутствие защиты памяти. Если обычная программа предпринимает попытку некорректного обращения к памяти, ядро может аварийно завершить процесс. Если ядро предпримет попытку некорректного обращения к памяти, результаты будут

менее контролируемыми. К тому же ядро не использует замещение страниц: каждый байт, используемый в ядре, – это один байт физической памяти.

- В ядре нельзя использовать вычисления с плавающей точкой. Активизация режима вычислений с плавающей точкой требует сохранения и проставления регистров устройства поддержки вычислений с плавающей точкой, помимо других рутинных операций.
- Фиксированный стек(причём довольно небольшой). Именно поэтому не рекомендуется использовать рекурсию в ядре.

Так же стоит уточнить, что существует 2 способа загрузки модуля в ядро:

1. **Сборка модуля вместе с ядром.** В таком случае загрузка модуля происходит как часть запуска системы, а сам модуль становится частью кода ядра.
2. **Динамическая загрузка в уже запущенной системе.** Вышеописанный способ создания модуля предполагает именно такой способ загрузки. В этом случае загрузка модуля больше похожа на запуск обычной пользовательской программы.

5.1. Два способа перехвата системного вызова

5.1.1. Небезопасный способ (старый)

Когда-то давно, ещё до ядра версии 2.6, для того, чтобы перехватить системный вызов, писали функцию-хук, которая её заменяла: выполняла другой код + вызывала непосредственно сам syscall(чтобы не нарушить работоспособность системы).
Так как каждый системный вызов подобно функции имеет свой адрес, а в Linux есть специальная таблица где эти адреса хранятся, **задача сводилась к тому, чтобы в этой самой таблице заменить адрес системного вызова на адрес нашей функции.**
Позже, разработчики Linux пытались устранить возможность такого способа, но до сих пор существуют хаки, которые позволяют реализовать этот метод.
Тем не менее, он очень небезопасный, и поэтому мы перейдем ко второму способу, который предоставляет более изящное и безопасное решение.

5.1.2. LSM

LSM — это фреймворк для разработки модулей безопасности ядра. Он был создан для того, чтобы расширить стандартную модель безопасности DAC, сделать её более гибкой. Этот **фреймворк использует известный модуль безопасности SELinux**, а также ещё несколько других, встроенных в ядро.
Самое ценное для нас в данном фреймворке то, что он **реализован через набор заранее предустановленных в ядро хуков**(по сути, тот способ, который я описывал выше, но безопасный, потому что ядро заранее рассчитано на наличие таких хуков).
LSM позволяет вставлять в код своих хуков вызов пользовательских, что позволяет безопасно работать с системными вызовами без изменения таблицы символов.

Полный список вызовов, поддерживаемых LSM-фреймворком представлен в источнике [8] (только примеры функций, который надо использовать).

```
...
      .task_setnice                   =           ptlsm_task_setnice,
      .task_setioprio                 =           ptlsm_task_setioprio,
      .task_getioprio                 =           ptlsm_task_getioprio,
      .task_setscheduler              =           ptlsm_task_setscheduler,
      .task_getscheduler              =           ptlsm_task_getscheduler,
```

```
	.task_movememory		=		ptlsm_task_movememory,
	.task_kill			=		ptlsm_task_kill,
	.task_wait			=		ptlsm_task_wait,
...
```

Название вызовов по большей части имеет говорящий характер: `task_kill, task_wait, task_setnice, inode_mkdir и т.д.`
К большому сожалению, не все системный вызовы представлены так очевидно, так например не хватает именно тех 3 вызовов, что рассматриваются в лабораторной работе ☹

5.2. Создание модуля (тестового – mkdir [5])

Находим в исходниках ядра папку security, создаём в ней папку для нашего модуля, а в ней — его исходный код foobar.c:

```
// /security/foobar/foobar.c
//---INCLUDES
#include <linux/module.h>
#include <linux/lsm_hooks.h>

//---HOOKS
//mkdir hook
static int foobar_inode_mkdir(struct inode *dir, struct dentry *dentry, umode_t
mask)
{
	printk("%s\n","<my_tag> mkdir hook");
	return 0;
}

//---HOOKS REGISTERING
static struct security_hook_list foobar_hooks[] =
{
	LSM_HOOK_INIT(inode_mkdir, foobar_inode_mkdir),
};

//---INIT
void __init foobar_add_hooks(void)
{
	security_add_hooks(foobar_hooks, ARRAY_SIZE(foobar_hooks));
}
```

Файл lsm_hooks.h содержит заголовки тех самых предустановленных хуков, LSM_HOOK_INIT регистрирует соответствие foobar_inode_mkdir() хуку inode_mkdir(), а security_add_hooks() добавляет нашу функцию в общий список пользовательских хуков LSM.

Примечание: при каждом вызове **mkdir** будет вызываться ф-ция **foobar_inode_mkdir()**.

Теперь добавляем заголовок нашей функции в файл **“/include/linux/lsm_hooks.h”:**

```
#ifdef CONFIG_SECURITY_FOOBAR
	extern void __init foobar_add_hooks(void);
#else
	static inline void __init foobar_add_hooks(void) { }
#endif
```

В файле “/**security/security.c**” находим функцию “**int __init security_init(void)**” и добавляем в её тело следующий вызов:

```
foobar_add_hooks();
```

Далее в папке с нашим модулем(/security/foobar/) создадим файл **Kconfig**:

```
config SECURITY_FOOBAR
bool "FooBar security module"
	default y
help
	Any help text here
```

Теперь добавим в файл «/security/Kconfig» и следующий текст сразу за строчкой “menu «Security options»":

```
source security/foobar/Kconfig
```

Примечание: это добавит наш пункт меню в глобальное меню настроек ядра.

Создадим Makefile в папке с нашим модулем:

```
obj-$(CONFIG_SECURITY_FOOBAR) += foobar.o
```

Откроем Makefile всего раздела безопасности(/security/Makefile) и добавим в него следующие строчки(по аналогии с такими же строчками для других модулей):

```
subdir-$(CONFIG_SECURITY_FOOBAR) += foobar
obj-$(CONFIG_SECURITY_FOOBAR) += foobar/
```

Теперь протестируем: запустим конфигурирование в псевдографическом режиме:

```
make menuconfig
```

Передем в подменю “Security options”, первым пунктом мы увидим наш модуль (рис., отмеченный символом “y” (мы установили это значение по умолчанию, когда создавали файл Kconfig), что означает, что мы интегрируем наш модуль непосредственно в код ядра.

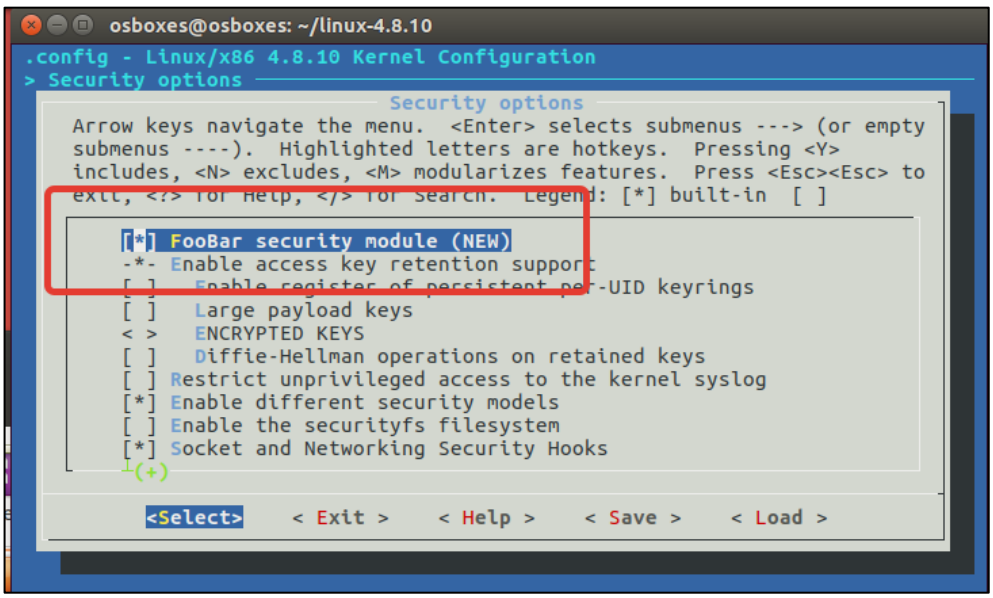

Рисунок 3. Подключенный модуль

Обновляем ядро уже известными командами:

```
osboxes@osboxes:~$ sudo dpkg -i linux-image-4.8.17-custom_4.8.17-custom-10.00.Custom_amd64.deb
osboxes@osboxes:~$ sudo dpkg -i linux-headers-4.8.17-custom_4.8.17-custom-10.00.Custom_amd64.deb
```

Еще во время загрузки мы увидим высплывающие теги `'<my_tag>'`. После загрузки выполняем следующее:

```
osboxes@osboxes:~$ dmesg | grep -m 5 '<my_tag>'
[    0.364037] <my_tag> mkdir hook
[    0.364129] <my_tag> mkdir hook
[    0.364313] <my_tag> mkdir hook
[    0.364406] <my_tag> mkdir hook
[    0.364563] <my_tag> mkdir hook
```

5.3. Создание модуля №2 (тест

Так как вызовы определить не удалось попробуем подобрать их наиболее вероятными обработчиками. Заменяем содержимое ./security/foobar/foobar.c на листинг, приведенный ниже.

```
// /security/foobar/foobar.c
//---INCLUDES
#include <linux/module.h>
#include <linux/lsm_hooks.h>

//---HOOKS
static int func_task_kill(struct task_struct *p, struct siginfo *info, int sig, u32
secid)
{
             printk("%s\n","<my_tag> task_kill!");
        return 0;
}
static int func_inode_setattr(struct dentry *dentry, struct iattr *iattr)
{
             printk("%s\n","<my_tag> inode_SETattr!");
        return 0;
}

static int func_inode_setsecurity(struct inode *inode, const char *name, const void
*value, size_t size, int flags)
{
        return 0;
}

//---HOOKS REGISTERING
static struct security_hook_list foobar_hooks[] =
{
       LSM_HOOK_INIT(task_kill, func_task_kill),
       //LSM_HOOK_INIT(ptlsm_inode_permission, func_ptlsm_inode_permission),
       LSM_HOOK_INIT(inode_setattr, func_inode_setattr),
       //LSM_HOOK_INIT(inode_getattr, func_inode_getattr),
       LSM_HOOK_INIT(inode_setsecurity, func_inode_setsecurity),
};

//---INIT
void __init foobar_add_hooks(void)
{
       security_add_hooks(foobar_hooks, ARRAY_SIZE(foobar_hooks));
}
```

Перекомпилируем ядро, и устанавливаем его, как описано чуть выше.

Источники:

1. Ubuntu 16.10 Yakkety Yak (Final)
http://www.osboxes.org/ubuntu/#ubuntu-16-10-vmware

2. Ubuntu 16.04, Kernel compile and default setting
http://technote.thispage.me/index.php/2016/12/20/ubuntu-16-04-kernel-compile-on-default-setting/

3. HOW-TO: Сборка ядра Linux
http://help.ubuntu.ru/wiki/%D1%81%D0%B1%D0%BE%D1%80%D0%BA%D0%B0_%D1%8F%D0%B4%D1%80%D0%B0

4. Sentido
http://www.sentido.ru/songs.php?id_song=4774

5. Перехват системных вызовов Linux с помощью LSM
https://habrahabr.ru/post/318106/

6. [xenial/master-next 1/1] UBUNTU: SAUCE: (no-up) disable -pie when gcc has it enabled by default
https://lists.ubuntu.com/archives/kernel-team/2016-May/077178.html

7. Relay Fork Module
https://lwn.net/Articles/122446/

8. Empty LSM
https://pastebin.com/Cst0VVQh

9. Встраивание в ядро Linux: перехват функций
https://habrahabr.ru/company/securitycode/blog/237089/

10. Встраивание в ядро Linux: перехват системных вызовов
https://habrahabr.ru/company/securitycode/blog/245539/