**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

а) 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.

**Программа работы**

1. Получить набор системных вызовов для исследования

2. Ознакомиться с назначением, функциональностью, параметрами

3. Выделить исходный код, указать точно принадлежность к версии ПО, сравнить версии заданных сист. вызовов в ядрах: 2.6 и 3.х, 4.х

4. Выполнить анализ кода

5. Предложить программы, создать примеры кода программ, выполнение которых приводит к запуску заданных сист. вызовов

6. Доказать это, применив профилирование выполнения программ различными средствами

7. Trace-логи и логи представить при демонстрации и в виде результирующих файлов

8. Перехватить системный вызов, т.е. частично заменить своим обработчиком, внеся модификацию в ядрах: 2.6, 3.х, 4.х

9. Продемонстрировать его работу, для этого создать примеры кода программ, использующих новый системный вызов, - вернее повторить ранее используемые программы из п.5

10. Доказать работоспособность частично измененных обработчиков профилированием функционирования созданного набора программ

11. Вернуть систему в исходное состояние.

12. Оформить отчет, включить в набор результатов файлы: отчета, исх. кодов сист. вызова до и после модификации для двух ядер, тексты программ, использующих вызовы, вых. файлы результатов профилирования и журналирования для исходного и модифицированного вызовов

13. Подготовить презентацию и сделать доклад

**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 –
3. sys\_rt\_sigpending –

**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. Выделить исходный код, указать точно принадлежность к версии ПО, сравнить версии заданных сист. вызовов в ядрах: 2.6 и 3.х, 4.х**

**2.1. Системный вызов «sys\_rt\_sigaction»**

**2.1.1. Kernel 2.6.11**

В ядре данной версии системный вызов определен 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»**

Листинг 2.1.1.а. «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.с. 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.с.** «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.а. «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.а. «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**.а. «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**.а. «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**

...

set\_tid\_address(0xb750c768) = 3378

set\_robust\_list(0xb750c770, 12) = 0

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\_SETTLS|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**

Данная программа порождает новый процесс с помощью fork(), затем в созданный процесс загружается новый исполняемый файл с помощью execve(), а родительский процесс завершает работу с помощью exit(). Исходный код процесса-потомка:

#include <stdio.h>

#include <unistd.h>

int main () {

int pid, ppid;

pid = getpid();

ppid = getppid();

printf("\n Son param : pid = %d, ppid = %d\n", pid, ppid);

sleep (3);

return 0;

}

Запустим написанную программу и проанализируем используемые ей системные вызовы с помощью утилиты strace:

kivi@kivi-VirtualBox:~/workspace/syscalls/c$ strace -f ./parent

execve("./parent", ["./parent"], [/\* 74 vars \*/]) = 0

………

getpid() = 3126

getppid() = 3124

………

write(1, "FATHER PARAM : pid = 3126, ppid "..., 40FATHER PARAM : pid = 3126, ppid = 3124

) = 40

clone(strace: Process 3127 attached

child\_stack=0, flags=CLONE\_CHILD\_CLEARTID|CLONE\_CHILD\_SETTID|SIGCHLD, child\_tidptr=0x7f188872a9d0) = 3127

[pid 3126] wait4(-1, <unfinished ...>

[pid 3127] execve("son", ["son"], [/\* 74 vars \*/]) = 0

………

[pid 3127] exit\_group(0) = ?

[pid 3127] +++ exited with 0 +++

………

--- SIGCHLD {si\_signo=SIGCHLD, si\_code=CLD\_EXITED, si\_pid=3127, si\_uid=1000, si\_status=0, si\_utime=0, si\_stime=0} ---

exit\_group(0) = ?

+++ exited with 0 +++

Флаг –f для strace указывает, чтобы отслеживался и родительский процесс, и все процессы порожденные им. Проанализировав вывод strace мы видим:

* Сразу после запуска программы с помощью execve в процесс загружается исполняемый файл написанной программы
* Вызов функции fork() приводит к вызову системного вызова clone()
* Порожденный процесс с помощью execve() загружает исполняемый код программы-потомка
* Порожденный процесс завершается с помощью системного вызова exit\_group()
* Родительский процесс получает сигнал о завершении потомка
* Родительский процесс завершается с помощью exit\_group

Мы видим, что несмотря на отсутствие определения функции в коде ядра, системный вызов execve() вызывается при запуске новой программы. Однако, системные вызовы fork() и exit() были заменены на clone() и exit\_group() соответственно. Это происходит из-за того, вызываемые в исходном коде функции (fork(), exit(), exevce()) – это просто функции библиотеки libc, которые являются обертками над реальными системными вызовами. В последних версиях libc эти функции были изменены так, чтобы они вызывали clone() и exit\_group().

clone() – это системный вызов Linux, который позволяет создать новый процесс (аналогично fork). Однако, в отличие от fork(), он еще позволяет указать с помощью специальных флагов указать какие части процесса у потомка и родителя будут общими (например, общее адресное пространство). Данный системный вызов появился в Linux для реализации потоков (т.к. в Linux поток это обычный процесс) и заменил собой fork().

exit\_group() – это системный вызов Linux, который используется для завершения процесса. Данный системный вызов по функциональности схож с exit(), но дополнительно он завершает не только основной процесс, но и все его потоки. Он так же появился в Linux для реализации многопоточности.

Для того, чтобы посмотреть, как работают данные системные вызовы, была написана программа на ассемблере NASM, которая напрямую вызывает системные вызовы ядра. Код программы:

section .text ; Code section.

global main ; the standard gcc entry point

main: ; the program label for the entry point

mov eax, 57 ; SYS\_FORK Op Code

syscall

cmp eax, 0 ;If the return value is 0, we are in the child process

jz child

parent:

mov edx, len ;Move msg length to edx

mov ecx, msg ;Move msg to ecx

call print ;Print

call exit ;Exit

jmp end

child:

mov edx, clen ;Move msg length to edx

mov ecx, cmsg ;Move msg to ecx

call print ;Print

call exit ;Exit

jmp end

print:

mov ebx,1 ;first argument: file handle (stdout)

mov eax,4 ;system call number (SYS\_WRITE)

int 0x80

ret

exit:

mov ebx,0 ; Exit code

mov eax,60 ; SYS\_EXIT

syscall

end:

mov edx, lnotexit

mov ecx, notexit

mov ebx,1 ;first argument: file handle (stdout)

mov eax,4 ;system call number (SYS\_WRITE)

int 0x80

section .data

timeval:

tv\_sec dd 0

tv\_usec dd 0

msg db "I AM YOUR FATHER",0xa

len equ $ - msg

cmsg db "I AM YOUR CHILD",0xa

clen equ $ - cmsg

notexit db "EXIT NOT EXITED",0xa

lnotexit equ $ - notexit

Системный вызов на ассемблере производится следующим образом:

* В регистр EAX записывается номер желаемого системного вызова
* В другие регистры процессора записываются передаваемые аргументы
* Вызывается функция syscall, которая инициирует системный вызов
* Возвращаемое значение системного вызова в результате записывается в регистр EAX

Таким образом, для выполнения желаемых системных вызовов нам необходимо выяснить номера системных вызовов для 64-битной системы:

* sys\_fork() – номер 57
* sys\_exit() – номер 60

Написанная программа работает по следующему алгоритму:

* Родительский процесс вызывает sys\_fork()
* Родительский процесс выводит сообщение “I AM THE FATHER”
* Процесс потомок выводит сообщение “I AM THE CHILD”
* Оба процесса завершают работу с помощью sys\_exit()
* Если по какой-либо причине sys\_exit() не срабатывает, оба процесса выводят сообщение “EXIT NOT EXITED”

Makefile для компиляции ассемблерной программы:

.PHONY: all

all:

nasm -f elf64 hello64.asm

gcc -m64 -o hello\_64 hello64.o

clean:

rm \*.o

rm nasm\_hello hello\_64

Выполним трассировку программы с помощью strace:

kivi@kivi-VirtualBox:~/workspace/syscalls/asm$ strace -f ./hello\_64

execve("./hello\_64", ["./hello\_64"], [/\* 74 vars \*/]) = 0

………

fork() = 4298

strace: Process 4298 attached

………

[pid 4298] I AM YOUR CHILD

………

[pid 4298] exit(1) = ?

I AM YOUR FATHER

………

[pid 4297] exit(1) = ?

[pid 4298] +++ exited with 1 +++

+++ exited with 1 +++

Мы видим, что программа работает как предполагалось, и при этом вызываются именно те системные вызовы, которые мы хотели.

1. **Перехваты системных вызовов**

Существует несколько способов осуществить перехват системных вызовов. Наиболее очевидным и небезопасным способом является изменения адреса обработчика системного вызова в таблице системных вызовов. Т.е. основная идея в том, чтобы написать свою функцию в ядре, которая будет иметь такой же прототип, как и обработчик системного вызова. Данная функция будет выполнять необходимые действия, а затем вызывать сам обработчик системного вызова. Далее в таблице системных вызовов по соответствующему номеру меняется адрес функции-обработчика.

Еще один способ – использование Linux Security Modules (LSM) – это фреймворк, который позволяет ядру Linux поддерживать некоторый набор модулей безопасности. Данный фреймворк является частью ядра Linux с версии 2.6 и выше. LSM реализуют более безопасный вариант предыдущей идеи: LSM уже содержит набор функций-заглушек, которые встроены в таблицу системных вызовов. По умолчанию данные заглушки ничего не выполняют. Однако LSM предоставляет инструменты встроить в эти функции дополнительные операции. LSM имеет заглушки практически на системные вызовы. Для перехвата системных вызовов необходимо добавить свой LSM-модуль в ядро и пересобрать его.

SystemTap — это средство, которое позволяет собирать и анализировать информацию о работающей Linux системе. SystemTap представляет из себя интерфейс командной строки и скриптовый язык программирования. Разработчики могут использовать SystemTap для мониторинга и анализа производительности системы, а разработчики программного обеспечения могут использовать SystemTap для анализа поведения приложения в работающей системе.

Основная идея SystemTap состоит в том, чтобы обозначить события и назначить для них обработчики. Во время выполнения скрипта, SystemTap занимается мониторингом событий и, как только произойдет событие, ядро системы выполнит обработчик. Событиями могут быть начало или конец сессии SystemTap, срабатывание таймера и другие.

Обработчиком является последовательность скриптовых операторов, которые будут выполнены после срабатывания события. Обычно обработчики извлекают информацию из контекста события или выводят информацию на экран. Сессия SystemTap начинается тогда, когда мы выполняем скрипт. В это время происходит следующая последовательность действий:

1. Сначала SystemTap проверяет библиотеку «тапсетов» на наличие использованных в скрипте;

2. Потом SystemTap транслирует скрипт в Си (язык программирования) и запускает системный компилятор, чтобы создать модуль ядра из скрипта;

3. SystemTap загружает модуль и активирует все события в скрипте;

4. Как только происходит событие выполняется обработчик данного события;

5. Когда все события выполнены, модуль выгружается и сессия завершается;

В SystemTap есть 2 типа событий: синхронные и асинхронные. Синхронные события привязаны к инструкции в определенном месте в коде ядра. Примеры синхронных событий:

— syscall.system\_call

— vfs.file\_operation

— kernel.function(«function»)

— module(«module»).function(«function»)

Асинхронные события не привязаны к определенной инструкции или определенному месту в коде ядра. Примеры асинхронных событий:

— begin — начало сессии SystemTap

— end — конец сессии SystemTap

— timer.event() — отсчет таймера (timer.s(4) — событие будет срабатывать каждые 4 секунды )

Таким образом, SystemTap еще более упрощает задачу перехвата системных вызовов и обеспечивает большую безопасность. Установить SystemTap можно из официальных репозиториев Ubuntu. Так же для его работоспособности необходимо установить пакет linux-x.x.x.x-dbgsym (x.x.x.x заменить на версию ядра), который содержит символы отладки для используемого ядра. После этого необходимо выполнить инициализацию SystemTap вызвав команду stap-prep (от имени администратора).

Для перехвата системных вызовов был написан простейший скрипт, который на каждый интересующий нас системный вызов будет выводить имя и PID процесса, осуществившего вызов.

probe syscall.fork

{

printf ("%s(%d) fork\n", execname(), pid())

}

probe syscall.execve

{

printf ("%s(%d) execve\n", execname(), pid())

}

probe syscall.exit

{

printf ("%s(%d) exit\n", execname(), pid())

}

Запускаем скрипт и в другом окне вызываем написанные ранее программы. Вывод SystemTap:

kivi@kivi-VirtualBox:~/workspace/syscalls$ sudo stap ./capture.stp

[sudo] password for kivi:

bash(4394) execve

hello\_64(4394) fork

hello\_64(4394) exit

hello\_64(4395) exit

bash(4398) execve

parent(4399) execve

Мы видим, что ассемблерная программа (hello\_64) выполняет вызовы fork и exit, а программа на С (parent) вызывает execve (но не вызывает fork и exit).

**Вывод**

В данной работе были изучены системные вызовы ядра Linux. Была изучения организация обработки системных вызовов в ядре (в различных версиях ядра). Было обнаружено, что два анализируемых системных вызова более не используются в ядре (т.к. они не поддерживают многопоточность). Были написаны программы на С и на ассемблере NASM, которые выполняют вызовы необходимых системных вызовов. Так же были проанализированы способы перехвата системных вызовов ядра. Существует несколько способов для этого. В данной работе была использована утилита SystemTap.

**Список использованных источников**

1. Linux Kernel: <https://www.kernel.org/>
2. SystemTap: <https://sourceware.org/systemtap/>
3. Сайт для индексирования различных версий ядра Linux <http://lxr.free-electrons.com/ident?i=sys_fork>
4. Официальный репозиторий ядра Linux: <https://github.com/torvalds/linux>