**Тема**: Анализ исходного кода системных вызовов

и их частичная замена (перехват)

**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

**Выполнение работы**

1. **Назначение, функциональность и параметры системных вызовов**

Согласно варианту, необходимо работать со следующими системными вызовами:

* sys\_fork() – создание нового процесса
* sys\_execve() – загрузка в текущий процесс другого исполняемого файла
* sys\_exit() – завершение текущего процесса

Данные системные вызовы вызываются из соответствующих системных вызовов fork(), execve() и exit(). Эти системные вызовы являются лишь обертками над реальными функциями ядра. Рассмотрим эти функции ядра более подробно.

**sys\_fork()**

Данная функция вызывается при вызове пользователем функции fork(). Реализация sys\_fork() зависит от архитектуры машины, на которой она вызывается. Прототип функции:

int sys\_fork(struct pt\_regs \*regs)

Структура pt\_regs это набор регистров процессора, которые сохранены в памяти процесса. При вызове sys\_fork в него передаются текущие значения регистров процессора. sys\_fork() делает соответствующие проверки (при необходимости) и вызывает архитектурно-независимую функцию do\_fork(), которая выполняет создание нового процесса.

**sys\_execve()**

Данная функция выполняет загрузку в адресное пространство процесса нового исполняемого файла. Реализация архитектурно зависима. Прототип функции:

long sys\_execve(char \*name, char \*\*argv,char \*\*envp, struct pt\_regs\* regs)

Аргументы данной функции ядра практически совпадают с аргументами системного вызова execve(): путь к новому исполняемому файлу, массив аргументов и массив переменных окружения. Так же, sys\_execve() получает указатель на структуру с регистрами процессора pt\_regs. Функция выполняет необходимые проверки аргументов и далее вызывает do\_execve().

**sys\_exit()**

Данная функция используется для завершения процесса с указанным возвращаемым значением. Прототип функции:

long sys\_exit(int error\_code)

Как видим, прототип sys\_exit() совпадает с прототипом системного вызова exit().

1. **Исходный код системных вызовов**

Проанализировав исходный код ядра версий 2.4, 2.6, 3.16 и 4.10 было выяснено, что исследуемые системные вызовы в явном виде присутствуют только в ядрах версии до 2.4. В последующих версиях ядра присутствуют объявления этих функций, однако их определений в исходном коде на С нет. Возможно это связано с тем, что определения этих файлов были каким-либо образом перенесены на уровень ассемблерного кода.

Приведем определения этих системных вызовов из ядра 2.4:

**sys\_fork() (**файл <linux/arch/x86\_64/kernel/process.c>)**:**

asmlinkage long sys\_fork(struct pt\_regs regs)

{

return do\_fork(SIGCHLD, regs.rsp, &regs, 0);

}

**sys\_exit()** (файл <linux/kernel/exit.c>):

asmlinkage long sys\_exit(int error\_code)

{

do\_exit((error\_code&0xff)<<8);

}

**sys\_execve()** (файл <linux/arch/x86\_64/kernel/process.c>):

asmlinkage long sys\_fork(struct pt\_regs regs)

{

return do\_fork(SIGCHLD, regs.rsp, &regs, 0);

}

Макрос asmlinkage в определении этих функций говорит о двух вещах:

* Функция не может принимать аргументы из регистров процессора; она принимает их со стека
* Функция может компоноваться в ассемблерном виде

Так же мы видим, что как ранее говорилось все данные системные вызовы проверяют свои аргументы (если это необходимо) и затем пересылают их следующей функции do\_\*(), которая реализует всю функциональность системного вызова.

1. **Примеры использования системных вызовов**

Анализ программ проводится на Ubuntu 16.04 с версией ядра 4.4.0-72-generic. Была написана программа на языке С, которая (предположительно) использует все три системных вызова:

#include <stdio.h>

#include <unistd.h>

#include <stdlib.h>

#include <sys/wait.h>

int main () {

int pid, ppid, status;

pid = getpid();

ppid = getppid();

printf("\nFATHER PARAM : pid = %d, ppid = %d\n\n", pid, ppid);

int cpid = fork();

if (cpid < 0) {

printf("Eror in fork\n");

exit(-1);

} else if (cpid == 0) {

execl("son", "son", NULL );

} else {

wait(&status);

}

exit(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>