

# Linux kernel [module] programming

Саютин Дмитрий  
cdkrot@yandex.ru

Летняя Компьютерная Школа

30.07.2016

## 1 Введение и история

# План спецкурса

- 1 Введение и история
- 2 Практика

- 1 Введение и история
- 2 Практика
- 3 Избранные главы о внутреннем устройстве ядра

# Введение и история

# Что же такое ядро операционной системы?

Что же такое ядро и зачем оно нужно?

# Что же такое ядро операционной системы?

Что же такое ядро и зачем оно нужно?

- Выполняется в приоритетном режиме, цена ошибок гораздо дороже.

# Что же такое ядро операционной системы?

Что же такое ядро и зачем оно нужно?

- Выполняется в приоритетном режиме, цена ошибок гораздо дороже.
- Обслуживает запросы пользовательских приложений.



# Что же такое ядро операционной системы?

Что же такое ядро и зачем оно нужно?

- Выполняется в приоритетном режиме, цена ошибок гораздо дороже.
- Обслуживает запросы пользовательских приложений.
- Защищает приложения друг от друга (например с помощью виртуальной памяти)

# Что же такое ядро операционной системы?

Что же такое ядро и зачем оно нужно?

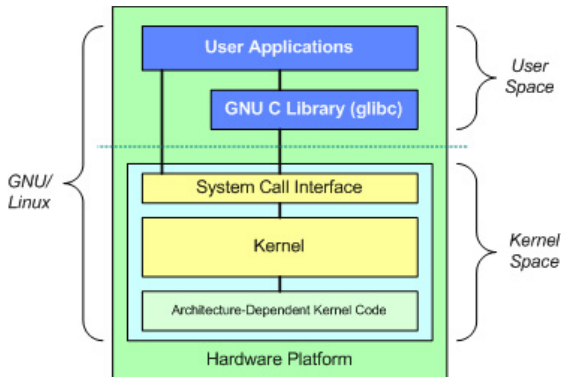
- Выполняется в приоритетном режиме, цена ошибок гораздо дороже.
- Обслуживает запросы пользовательских приложений.
- Защищает приложения друг от друга (например с помощью виртуальной памяти)
- Предоставляет единый интерфейс к оборудованию.

# Что же такое ядро операционной системы?

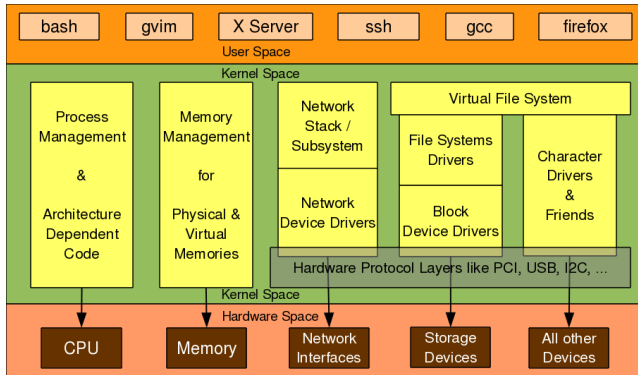
Что же такое ядро и зачем оно нужно?

- Выполняется в приоритетном режиме, цена ошибок гораздо дороже.
- Обслуживает запросы пользовательских приложений.
- Защищает приложения друг от друга (например с помощью виртуальной памяти)
- Предоставляет единый интерфейс к оборудованию.
- Обеспечивает справедливый доступ к ресурсам системы (например планировщик задач)

# Linux architecture



# Linux architecture



# Linux: Немного истории

- Эта история начинается в 1969 году, когда в Bell Labs начали разрабатывать операционную систему Unix.

# Linux: Немного истории

- Эта история начинается в 1969 году, когда в Bell Labs начали разрабатывать операционную систему Unix.
- Изначально она была написана на ассемблере, как и все другие ОС того времени, но впоследствии была переписана на язык Си.

# Linux: Немного истории

- Эта история начинается в 1969 году, когда в Bell Labs начали разрабатывать операционную систему Unix.
- Изначально она была написана на ассемблере, как и все другие ОС того времени, но впоследствии была переписана на язык Си.
- К слову сам язык Си был создан специально под Unix: чтобы было возможно портировать код на другие аппаратные платформы.



# Linux: Немного истории

- Эта история начинается в 1969 году, когда в Bell Labs начали разрабатывать операционную систему Unix.
- Изначально она была написана на ассемблере, как и все другие ОС того времени, но впоследствии была переписана на язык Си.
- К слову сам язык Си был создан специально под Unix: чтобы было возможно портировать код на другие аппаратные платформы.
- 1970 год считается 'Эпохой Unix'. И по сей день все Unix-подобные системы измеряют время в количестве секунд с полуночи первого января 1970 года

# Linux: Немного истории

- Эта история начинается в 1969 году, когда в Bell Labs начали разрабатывать операционную систему Unix.
- Изначально она была написана на ассемблере, как и все другие ОС того времени, но впоследствии была переписана на язык Си.
- К слову сам язык Си был создан специально под Unix: чтобы было возможно портировать код на другие аппаратные платформы.
- 1970 год считается 'Эпохой Unix'. И по сей день все Unix-подобные системы измеряют время в количестве секунд с полуночи первого января 1970 года
- Изначально Unix выпускалась под достаточно свободной лицензией и исходный код был доступен всем желающим.

# Linux: Немного истории

- Эта история начинается в 1969 году, когда в Bell Labs начали разрабатывать операционную систему Unix.
- Изначально она была написана на ассемблере, как и все другие ОС того времени, но впоследствии была переписана на язык Си.
- К слову сам язык Си был создан специально под Unix: чтобы было возможно портировать код на другие аппаратные платформы.
- 1970 год считается 'Эпохой Unix'. И по сей день все Unix-подобные системы измеряют время в количестве секунд с полуночи первого января 1970 года
- Изначально Unix выпускалась под достаточно свободной лицензией и исходный код был доступен всем желающим.
- В общем, Unix быстро развивался и снискал популярность в научных учреждениях и предприятиях.

# Linux: Немного истории

- Однако в районе 1983-1984 года Unix перешёл на коммерческую модель разработки, о предоставлении исходников речь уже не шла. Теперь Unix нужно было покупать за деньги.

# Linux: Немного истории

- Однако в районе 1983-1984 года Unix перешёл на коммерческую модель разработки, о предоставлении исходников речь уже не шла. Теперь Unix нужно было покупать за деньги.
- В примерно те же годы Ричард Столлман (RMS) основал проект GNU с целью создания полноценной Unix-совместимой системы, полностью состоящей из свободного программного обеспечения.

# Linux: Немного истории

- Однако в районе 1983-1984 года Unix перешёл на коммерческую модель разработки, о предоставлении исходников речь уже не шла. Теперь Unix нужно было покупать за деньги.
- В примерно те же годы Ричард Столлман (RMS) основал проект GNU с целью создания полноценной Unix-совместимой системы, полностью состоящей из свободного программного обеспечения.
- К 1990 году большая часть программ, необходимых операционной системе была написана. Но им не хватало ядра.

# Linux: Немного истории

- Однако в районе 1983-1984 года Unix перешёл на коммерческую модель разработки, о предоставлении исходников речь уже не шла. Теперь Unix нужно было покупать за деньги.
- В примерно те же годы Ричард Столлман (RMS) основал проект GNU с целью создания полноценной Unix-совместимой системы, полностью состоящей из свободного программного обеспечения.
- К 1990 году большая часть программ, необходимых операционной системе была написана. Но им не хватало ядра.
- 25 Августа 1991 года финнский студент Линус Торвальдс представил первую версию своего ядра.

- Однако в районе 1983-1984 года Unix перешёл на коммерческую модель разработки, о предоставлении исходников речь уже не шла. Теперь Unix нужно было покупать за деньги.
- В примерно те же годы Ричард Столлман (RMS) основал проект GNU с целью создания полноценной Unix-совместимой системы, полностью состоящей из свободного программного обеспечения.
- К 1990 году большая часть программ, необходимых операционной системе была написана. Но им не хватало ядра.
- 25 Августа 1991 года финнский студент Линус Торвальдс представил первую версию своего ядра.
- Оно, кстати, изначально называлось Frex – от слов Free и Unix.



# Linux: Первый email

Hello everybody out there using minix -

# Linux: Первый email

Hello everybody out there using minix -

I'm doing a (free) operating system (just a hobby, won't be big and professional like gnu) for 386(486) AT clones. This has been brewing since april, and is starting to get ready.

# Linux: Первый email

Hello everybody out there using minix -

I'm doing a (free) operating system (just a hobby, won't be big and professional like gnu) for 386(486) AT clones. This has been brewing since april, and is starting to get ready. I'd like any feedback on things people like/dislike in minix, as my OS resembles it somewhat (same physical layout of the file-system (due to practical reasons) among other things).

# Linux: Первый email

Hello everybody out there using minix -

I'm doing a (free) operating system (just a hobby, won't be big and professional like gnu) for 386(486) AT clones. This has been brewing since april, and is starting to get ready. I'd like any feedback on things people like/dislike in minix, as my OS resembles it somewhat (same physical layout of the file-system (due to practical reasons) among other things).

I've currently ported bash(1.08) and gcc(1.40), and things seem to work. This implies that I'll get something practical within a few months, and I'd like to know what features most people would want. Any suggestions are welcome, but I won't promise I'll implement them :)

# Linux: Первый email

Hello everybody out there using minix -

I'm doing a (free) operating system (just a hobby, won't be big and professional like gnu) for 386(486) AT clones. This has been brewing since april, and is starting to get ready. I'd like any feedback on things people like/dislike in minix, as my OS resembles it somewhat (same physical layout of the file-system (due to practical reasons) among other things).

I've currently ported bash(1.08) and gcc(1.40), and things seem to work. This implies that I'll get something practical within a few months, and I'd like to know what features most people would want. Any suggestions are welcome, but I won't promise I'll implement them :)

Linus (torvalds@kruuna.helsinki.fi)

PS. Yes - it's free of any minix code, and it has a multi-threaded fs. It is NOT protable (uses 386 task switching etc), and it probably never will support anything other than AT-harddisks, as that's all I have :-).

Модули для ядра работают с теми же привелегиями, что и само ядро.

Модули для ядра работают с теми же привелегиями, что и само ядро.

Более того, достаточно большая ядра (например драйверы) реализована именно в виде модулей.

Модули для ядра работают с теми же привелегиями, что и само ядро.

Более того, достаточно большая ядра (например драйверы) реализована именно в виде модулей.

Словом, написание модулей ядра ничуть не хуже внесения изменений в само ядро, но требует немного меньше времени для подготовки.



# Подготовим рабочее место

Чтобы писать модули нужно поставить заголовочные файлы ядра (Впрочем, возможно, что они у вас уже стоят)

Чтобы писать модули нужно поставить заголовочные файлы ядра (Впрочем, возможно, что они у вас уже стоят)

- Для дистрибутивов на основе Debian:

```
|| sudo apt-get install linux-headers-$(uname -r)
```

- Для Fedora:

```
|| sudo yum -y install kernel-devel kernel-headers
```

- Для Gentoo:

```
|| sudo emerge -a -e world # echo 'just kidding'
```

- Для другого дистрибутива поищите в интернете 'Linux headers \$distrname'

# Практика

# Hello world!

Приступим

```
mkdir -p code/01-hello  
cd code/01-hello
```

# Hello world!

Приступим

```
mkdir -p code/01-hello  
cd code/01-hello  
$EDITOR main.c
```

## Приступим

```
mkdir -p code/01-hello  
cd code/01-hello  
$EDITOR main.c  
$EDITOR Makefile
```

# Hello world: main.c

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>
```

# Hello world: main.c

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>

static int __init hello_init(void) {
    printk(KERN_INFO "Hello kernel\n");
    return 0;
}
```



# Hello world: main.c

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>

static int __init hello_init(void) {
    printk(KERN_INFO "Hello kernel\n");
    return 0;
}

static void __exit hello_exit(void) {
    printk(KERN_INFO "Good bye kernel\n");
}
```

# Hello world: main.c

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>

static int __init hello_init(void) {
    printk(KERN_INFO "Hello kernel\n");
    return 0;
}

static void __exit hello_exit(void) {
    printk(KERN_INFO "Good bye kernel\n");
}

module_init(hello_init);
module_exit(hello_exit);
```

# Hello world: main.c

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>

static int __init hello_init(void) {
    printk(KERN_INFO "Hello kernel\n");
    return 0;
}

static void __exit hello_exit(void) {
    printk(KERN_INFO "Good bye kernel\n");
}

module_init(hello_init);
module_exit(hello_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Summer Informatics School");
MODULE_DESCRIPTION("Hello world module");
```

# Hello world: Makefile

- Код мы написали, теперь осталось его скомпилировать.

# Hello world: Makefile

- Код мы написали, теперь осталось его скомпилировать.
- Для этого напишем специальный файл – Makefile. Makefile это популярный способ описания зависимостей различных задач друг от друга.

# Hello world: Makefile

- Код мы написали, теперь осталось его скомпилировать.
- Для этого напишем специальный файл – Makefile. Makefile это популярный способ описания зависимостей различных задач друг от друга.
- Он часто используется для сборок различных проектов в мире Unix (или Linux).

# Hello world: Makefile

```
|| ifneq ($(KERNELRELEASE),)
```

# Hello world: Makefile

```
ifneq ($(KERNELRELEASE),)
obj-m    := hello.o
hello-y  := main.o # this line can have more than one
           file.
else
```



# Hello world: Makefile

```
ifneq ($(KERNELRELEASE),)
obj-m      := hello.o
hello-y    := main.o # this line can have more than one
             file.
else
all:
    make -C /lib/modules/`uname -r`/build M=$$PWD \
modules

clean:
    make -C /lib/modules/`uname -r`/build M=$$PWD \
clean

endif
```

# Hello world: Building and running

```
|| make # or 'make all'
```

# Hello world: Building and running

```
|| make # or 'make all'  
||  
|| ls -a
```

# Hello world: Building and running

```
make # or 'make all'  
  
ls -a  
  
sudo insmod hello.ko  
dmesg | tail
```

# Hello world: Building and running

```
make # or 'make all'

ls -a

sudo insmod hello.ko
dmesg | tail

lsmod | grep hello # Our module was loaded!
```

# Hello world: Building and running

```
make # or 'make all'

ls -a

sudo insmod hello.ko
dmesg | tail

lsmod | grep hello # Our module was loaded!

sudo rmmod hello.ko
dmesg | tail
```

# Hello world: Building and running

```
make # or 'make all'

ls -a

sudo insmod hello.ko
dmesg | tail

lsmod | grep hello # Our module was loaded!

sudo rmmod hello.ko
dmesg | tail

sudo modinfo hello.ko
```

# Hello world: Building and running

```
make # or 'make all'

ls -a

sudo insmod hello.ko
dmesg | tail

lsmod | grep hello # Our module was loaded!

sudo rmmod hello.ko
dmesg | tail

sudo modinfo hello.ko

make clean
```



# О качестве кода

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>

static int __init null_init(void) {
    int* ptr = NULL;
    printk(KERN_INFO "*ptr is %d\n", *ptr);
    return 0;
}

static void __exit null_exit(void) {}

module_init(null_init);
module_exit(null_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Summer Informatics School");
MODULE_DESCRIPTION("Null bug module");
```

# Running null module

Не повторяйте этого дома!

```
|| make
```

# Running null module

Не повторяйте этого дома!

```
make
```

```
sudo insmod null.ko
```

# Running null module

Не повторяйте этого дома!

```
make
```

```
sudo insmod null.ko
```

```
# .....
```

- Одним из традиционных способов взаимодействия ядра с пользовательскими программами являются псевдофайлы. С их помощью системный администратор может узнать состояние системы. Также такими файлами зачастую пользуются скрипты и различные утилиты.

- Одним из традиционных способов взаимодействия ядра с пользовательскими программами являются псевдофайлы. С их помощью системный администратор может узнать состояние системы. Также такими файлами зачастую пользуются скрипты и различные утилиты.
- Например так делают программы `ps`, `[h]top`, `uptime`, и т.п., потому что по-другому узнать подробную информацию о процессах нельзя.

- Одним из традиционных способов взаимодействия ядра с пользовательскими программами являются псевдофайлы. С их помощью системный администратор может узнать состояние системы. Также такими файлами зачастую пользуются скрипты и различные утилиты.
- Например так делают программы `ps`, `[h]top`, `uptime`, и т.п., потому что по-другому узнать подробную информацию о процессах нельзя.
- Такие псевдофайлы можно увидеть в `/dev` (файлы устройств и псевдоустройств), в `/proc` (информация о запущенных процессах) и в `/sys` (средство конфигурации ядра и модулей в runtime).

- Одним из традиционных способов взаимодействия ядра с пользовательскими программами являются псевдофайлы. С их помощью системный администратор может узнать состояние системы. Также такими файлами зачастую пользуются скрипты и различные утилиты.
- Например так делают программы `ps`, `[h]top`, `uptime`, и т.п., потому что по-другому узнать подробную информацию о процессах нельзя.
- Такие псевдофайлы можно увидеть в `/dev` (файлы устройств и псевдоустройств), в `/proc` (информация о запущенных процессах) и в `/sys` (средство конфигурации ядра и модулей в runtime).
- В качестве примера мы напишем здесь очень простой, но полноценный псевдофайл.



# Proc example: part 1

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/proc_fs.h>
#include <linux/seq_file.h>
```

# Proc example: part 1

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/proc_fs.h>
#include <linux/seq_file.h>

int file_show(struct seq_file* f, void* p) {
    seq_printf(f, "Hello world\n");
    return 0;
}
```

# Proc example: part 1

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/proc_fs.h>
#include <linux/seq_file.h>

int file_show(struct seq_file* f, void* p) {
    seq_printf(f, "Hello world\n");
    return 0;
}

int file_open(struct inode* i, struct file* f) {
    return single_open(f, file_show, NULL);
}
```

# Proc example: part 1

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/proc_fs.h>
#include <linux/seq_file.h>

int file_show(struct seq_file* f, void* p) {
    seq_printf(f, "Hello world\n");
    return 0;
}

int file_open(struct inode* i, struct file* f) {
    return single_open(f, file_show, NULL);
}

struct file_operations file_ops = {
    .owner      = THIS_MODULE,
    .open       = file_open,
    .read       = seq_read,
    .llseek     = seq_lseek,
    .release    = single_release
};
```

## Proc example: part 2

```
static int __init proc_init(void) {  
    if (proc_create("greets", 0666, NULL, &file_ops)  
        == NULL)  
        return -EIO;  
    return 0;  
}
```

## Proc example: part 2

```
static int __init proc_init(void) {
    if (proc_create("greet", 0666, NULL, &file_ops)
        == NULL)
        return -EIO;
    return 0;
}

static void __exit proc_exit(void) {
    remove_proc_entry("greet", NULL);
}
```

## Proc example: part 2

```
static int __init proc_init(void) {
    if (proc_create("greets", 0666, NULL, &file_ops)
        == NULL)
        return -EIO;
    return 0;
}

static void __exit proc_exit(void) {
    remove_proc_entry("greets", NULL);
}

MODULE_AUTHOR("Summer Informatics School");
MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("Proc file example");
```

## Proc example: part 2

```
static int __init proc_init(void) {
    if (proc_create("greets", 0666, NULL, &file_ops)
        == NULL)
        return -EIO;
    return 0;
}

static void __exit proc_exit(void) {
    remove_proc_entry("greets", NULL);
}

MODULE_AUTHOR("Summer Informatics School");
MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("Proc file example");

module_init(proc_init);
module_exit(proc_exit);
```



# Proc example: running

```
|| make  
|| sudo insmod proc.ko
```

# Proc example: running

```
make  
sudo insmod proc.ko  
  
cat /proc/greets  
ls -l /proc/greets
```

# Proc example: running

```
make
sudo insmod proc.ko

cat /proc/greets
ls -l /proc/greets

sudo rmmod proc.ko
```

# Proc example: running

```
make
sudo insmod proc.ko

cat /proc/greets
ls -l /proc/greets

sudo rmmod proc.ko
ls /proc/greets
```

# Params example: motivation

Следующий пример посвящается параметрам модулей. Они являются самым простым способом конфигурации кода в ядре.

# Params example: motivation

Следующий пример посвящается параметрам модулей. Они являются самым простым способом конфигурации кода в ядре. Как передать параметры в ядро?

# Params example: motivation

Следующий пример посвящается параметрам модулей. Они являются самым простым способом конфигурации кода в ядре. Как передать параметры в ядро?

- Для динамически загружаемых модулей (как мы пишем) можно передать прямо через `insmod`.

# Params example: motivation

Следующий пример посвящается параметрам модулей. Они являются самым простым способом конфигурации кода в ядре. Как передать параметры в ядро?

- Для динамически загружаемых модулей (как мы пишем) можно передать прямо через `insmod`.
- Также параметры можно менять в процессе работы модуля через `/sys`, если модуль это разрешает.



Следующий пример посвящается параметрам модулей. Они являются самым простым способом конфигурации кода в ядре. Как передать параметры в ядро?

- Для динамически загружаемых модулей (как мы пишем) можно передать прямо через `insmod`.
- Также параметры можно менять в процессе работы модуля через `/sys`, если модуль это разрешает.
- Для модулей, статически вкомпилированных в ядро, можно передавать через `kernel commandline`.

Следующий пример посвящается параметрам модулей. Они являются самым простым способом конфигурации кода в ядре. Как передать параметры в ядро?

- Для динамически загружаемых модулей (как мы пишем) можно передать прямо через `insmod`.
- Также параметры можно менять в процессе работы модуля через `/sys`, если модуль это разрешает.
- Для модулей, статически вкомпилированных в ядро, можно передавать через `kernel commandline`.
- Когда в следующий раз будете включать свой ноутбук войдите в `grub`-е в режим редактирования нужного профиля, там вы обнаружите строку начинающуюся со слова `'linux'`. Это она и есть.

# Params example: code

```
// ...  
char* name = "world";  
int magic = 2016;
```

# Params example: code

```
// ...  
char* name = "world";  
int magic = 2016;  
  
// variable, type, sysfs mode  
module_param(name, charp, 0000);
```

# Params example: code

```
// ...  
char* name = "world";  
int magic = 2016;  
  
// variable, type, sysfs mode  
module_param(name, charp, 0000);  
// user-friendly name, variable, type, sysfs mode.  
module_param_named(SuperFlag, magic, int, 0600);
```

# Params example: code

```
// ...
char* name = "world";
int magic = 2016;

// variable, type, sysfs mode
module_param(name, charp, 0000);
// user-friendly name, variable, type, sysfs mode.
module_param_named(SuperFlag, magic, int, 0600);

int file_show(struct seq_file* f, void* p) {
    seq_printf(f, "Hello %s, and magic is %d\n", name,
               magic);
    return 0;
}
// ...
```

# Params example: running

```
|| make  
|| sudo insmod params.ko
```

# Params example: running

```
make  
sudo insmod params.ko  
  
cat /proc/greets
```



# Params example: running

```
make
sudo insmod params.ko

cat /proc/greets

echo 2017 | sudo tee -a /sys/module/params/parameters/
    SuperFlag
```

# Params example: running

```
make
sudo insmod params.ko

cat /proc/greets

echo 2017 | sudo tee -a /sys/module/params/parameters/
    SuperFlag

cat /proc/greets
```

# Params example: running

```
make
sudo insmod params.ko

cat /proc/greets

echo 2017 | sudo tee -a /sys/module/params/parameters/
    SuperFlag

cat /proc/greets

sudo rmmod params.ko
sudo insmod params.ko name=LKSH SuperFlag=1999
```

# Params example: running

```
make
sudo insmod params.ko

cat /proc/greets

echo 2017 | sudo tee -a /sys/module/params/parameters/
    SuperFlag

cat /proc/greets

sudo rmmod params.ko
sudo insmod params.ko name=LKSH SuperFlag=1999

cat /proc/greets
```

# Params example: running

```
make
sudo insmod params.ko

cat /proc/greets

echo 2017 | sudo tee -a /sys/module/params/parameters/
    SuperFlag

cat /proc/greets

sudo rmmod params.ko
sudo insmod params.ko name=LKSH SuperFlag=1999

cat /proc/greets

sudo rmmod params.ko
```

## Избранные главы о внутреннем устройстве ядра

Как работают системные вызовы?

- Начнём с регистров.

Как работают системные вызовы?

- Начнём с регистров. Регистр это такая ячейка памяти, находящаяся внутри процессора. Это самая быстрая память в компьютере. Регистры являются именованными, нет такого понятия, как адрес регистра.



Как работают системные вызовы?

- Начнём с регистров. Регистр это такая ячейка памяти, находящаяся внутри процессора. Это самая быстрая память в компьютере. Регистры являются именованными, нет такого понятия, как адрес регистра.
- Для того, чтобы процессор мог обработать какие-то данные, их надо загрузить в регистры. Однако языки высокого уровня (C, C++, ...) делают это за нас.

## Как работают системные вызовы?

- Начнём с регистров. Регистр это такая ячейка памяти, находящаяся внутри процессора. Это самая быстрая память в компьютере. Регистры являются именованными, нет такого понятия, как адрес регистра.
- Для того, чтобы процессор мог обработать какие-то данные, их надо загрузить в регистры. Однако языки высокого уровня (C, C++, ...) делают это за нас.
- Чтобы вызвать системный вызов необходимо:
  - Поместить в определённые регистры параметры вызова, включая номер системного вызова.
  - Сделать прерывание. Прерывание – это специальный механизм, передающий управление в ядро.
  - После этого ядро вызывает вызов и возвращает управление назад в программу пользователя.

Большинство системных вызовов имеют свои функции-спутники в стандартной библиотеке C (`libc`).

Например данные функции осуществляют вызов одноимённых системных вызовов ядра.

- `open` – открывает файл для чтения/записи.

Большинство системных вызовов имеют свои функции-спутники в стандартной библиотеке C (`libc`).

Например данные функции осуществляют вызов одноимённых системных вызовов ядра.

- `open` – открывает файл для чтения/записи.
- `write` – осуществляет запись в файл.

Большинство системных вызовов имеют свои функции-спутники в стандартной библиотеке C (`libc`).

Например данные функции осуществляют вызов одноимённых системных вызовов ядра.

- `open` – открывает файл для чтения/записи.
- `write` – осуществляет запись в файл.
- `read` – осуществляет чтение.

Большинство системных вызовов имеют свои функции-спутники в стандартной библиотеке C (`libc`).

Например данные функции осуществляют вызов одноимённых системных вызовов ядра.

- `open` – открывает файл для чтения/записи.
- `write` – осуществляет запись в файл.
- `read` – осуществляет чтение.
- `exit` – завершить программу

Большинство системных вызовов имеют свои функции-спутники в стандартной библиотеке C (`libc`).

Например данные функции осуществляют вызов одноимённых системных вызовов ядра.

- `open` – открывает файл для чтения/записи.
- `write` – осуществляет запись в файл.
- `read` – осуществляет чтение.
- `exit` – завершить программу
  - на самом деле, когда вы делаете `return 0` – `libc` вызывает системный вызов выхода, просто так выйти программа не может.
- ...

- Теперь давайте напишем программу, которая будет печатать на консоль 'Hello world'.
- Для того, чтобы выполнить описанные требования достаточно двух системных вызовов: `write`, `exit`.



- Теперь давайте напишем программу, которая будет печатать на консоль 'Hello world'.
- Для того, чтобы выполнить описанные требования достаточно двух системных вызовов: `write`, `exit`.
- Мы напишем данную программу на разных языках, и посмотрим на число вызовов, которые они сделают.

# Системные вызовы: Assembler

```
.globl _start
# x86_64 only
.data
msg:
    .ascii "Hello, world\n"
    len = . - msg

.text
_start:
    # SYS_write = 1
    movq $1, %rax
    movq $1, %rdi
    movq $msg, %rsi
    movq $len, %rdx
    syscall

    # SYS_exit = 60
    movq $60, %rax
    movq $0, %rdi
    syscall
```

# Системные вызовы: Assembler

```
.globl _start
# x86_64 only
.data
msg:
    .ascii "Hello, world\n"
    len = . - msg

.text
_start:
    # SYS_write = 1
    movq $1, %rax
    movq $1, %rdi
    movq $msg, %rsi
    movq $len, %rdx
    syscall

    # SYS_exit = 60
    movq $60, %rax
    movq $0, %rdi
    syscall
```

```
gcc -nostdlib asm.s -o asm
```

# Системные вызовы: Assembler

```
.globl _start
# x86_64 only
.data
msg:
    .ascii "Hello, world\n"
    len = . - msg

.text
_start:
    # SYS_write = 1
    movq $1, %rax
    movq $1, %rdi
    movq $msg, %rsi
    movq $len, %rdx
    syscall

    # SYS_exit = 60
    movq $60, %rax
    movq $0, %rdi
    syscall
```

```
gcc -nostdlib asm.s -o asm
./asm
```

# Системные вызовы: Assembler

```
.globl _start
# x86_64 only
.data
msg:
    .ascii "Hello, world\n"
    len = . - msg

.text
_start:
    # SYS_write = 1
    movq $1, %rax
    movq $1, %rdi
    movq $msg, %rsi
    movq $len, %rdx
    syscall

    # SYS_exit = 60
    movq $60, %rax
    movq $0, %rdi
    syscall
```

```
gcc -nostdlib asm.s -o asm
./asm
strace ./asm # 2 syscalls
```

Сколько вызовов сделает данный код?

```
#include <string.h>
#include <unistd.h>

const char* msg = "Hello, world\n";

int main() {
    write(1, msg, strlen(msg));
    return 0;
}
```

Сколько вызовов сделает данный код?

```
#include <string.h>
#include <unistd.h>

const char* msg = "Hello, world\n";

int main() {
    write(1, msg, strlen(msg));
    return 0;
}
```

```
strace ./C # ~ 25 syscalls.
```

Сколько вызовов сделает данный код?

```
// same, but compiled as c++.  
#include <string.h>  
#include <unistd.h>  
  
const char* msg = "Hello, world\n";  
  
int main() {  
    write(1, msg, strlen(msg));  
    return 0;  
}
```



Сколько вызовов сделает данный код?

```
// same, but compiled as c++.
#include <string.h>
#include <unistd.h>

const char* msg = "Hello, world\n";

int main() {
    write(1, msg, strlen(msg));
    return 0;
}
```

```
strace ./Cpp # ~50 calls
```

Сколько вызовов сделает данный код?

```
public class Java {  
    public static void main(String[] args) {  
        System.out.println("Hello World");  
    }  
}
```

Сколько вызовов сделает данный код?

```
public class Java {  
    public static void main(String[] args) {  
        System.out.println("Hello World");  
    }  
}
```

```
strace -f java Java # ~ 3378 calls (highly random  
    number), 17 threads.
```

Да, теперь вы знаете, что для того, чтобы написать Hello world на Java нужно 17 потоков и 3378 вызовов.

# О неиспользовании языка C++ в линуксе

Почему C++ не используется в линуксе?

# О неиспользовании языка C++ в линуксе

Почему C++ не используется в линуксе?

- Исторические причины: C появился раньше и первые версии писали на нём.

Почему C++ не используется в линуксе?

- Исторические причины: C появился раньше и первые версии писали на нём.

Дальше тезисные выжимки из письма Линуса Торвальдса по теме.

- C++ – Это ужасный язык программирования. И от того, что множество неумелых программистов пишут на нём, он становится только хуже. Следовало бы выбрать язык C, даже если бы единственной целью этого было бы удержать программистов C++ от дел.

## О неиспользовании языка C++ в линуксе (part 2)

- На самом деле они попробовали однажды перевести линукс на C++, в 1992. Попытка провалилась. Конечно компиляторы C++ стали лучше, но основное (по словам Торвальдса) не изменилось:

## О неиспользовании языка C++ в линуксе (part 2)

- На самом деле они попробовали однажды перевести линукс на C++, в 1992. Попытка провалилась. Конечно компиляторы C++ стали лучше, но основное (по словам Торвальдса) не изменилось:
  - Исключения C++ фундаментально сломаны. Они особенно сломаны для разработки ядер операционных систем.



## О неиспользовании языка C++ в линуксе (part 2)

- На самом деле они попробовали однажды перевести линукс на C++, в 1992. Попытка провалилась. Конечно компиляторы C++ стали лучше, но основное (по словам Торвальдса) не изменилось:
  - Исключения C++ фундаментально сломаны. Они особенно сломаны для разработки ядер операционных систем.
  - Если компилятор или язык любит прятать такие детали, как выделение памяти, то это не очень хороший язык (компилятор).

## О неиспользовании языка C++ в линуксе (part 2)

- На самом деле они попробовали однажды перевести линукс на C++, в 1992. Попытка провалилась. Конечно компиляторы C++ стали лучше, но основное (по словам Торвальдса) не изменилось:
  - Исключения C++ фундаментально сломаны. Они особенно сломаны для разработки ядер операционных систем.
  - Если компилятор или язык любит прятать такие детали, как выделение памяти, то это не очень хороший язык (компилятор).
  - Можно писать ООП код и на C (например используя указатели на функции, см. пример с proc).

## О неиспользовании языка C++ в линуксе (part 2)

- На самом деле они попробовали однажды перевести линукс на C++, в 1992. Попытка провалилась. Конечно компиляторы C++ стали лучше, но основное (по словам Торвальдса) не изменилось:
  - Исключения C++ фундаментально сломаны. Они особенно сломаны для разработки ядер операционных систем.
  - Если компилятор или язык любит прятать такие детали, как выделение памяти, то это не очень хороший язык (компилятор).
  - Можно писать ООП код и на C (например используя указатели на функции, см. пример с proc).
  - После перехода на C++ в '92 сразу появилось много жалоб о падении производительности.

- Что такое виртуальная память?

- Что такое виртуальная память?
- Виртуальная память означает то, что каждому процессу выделяется виртуальное пространство размером в  $2^{32}$  (или  $2^{64}$ ) байт.

- Что такое виртуальная память?
- Виртуальная память означает то, что каждому процессу выделяется виртуальное пространство размером в  $2^{32}$  (или  $2^{64}$ ) байт.
- TLB

- Что такое виртуальная память?
- Виртуальная память означает то, что каждому процессу выделяется виртуальное пространство размером в  $2^{32}$  (или  $2^{64}$ ) байт.
- TLB
- Page fault

- Что такое виртуальная память?
- Виртуальная память означает то, что каждому процессу выделяется виртуальное пространство размером в  $2^{32}$  (или  $2^{64}$ ) байт.
- TLB
- Page fault
- Kernel обрабатывает Page fault, загружая нужную страницу.
- VDSO.



Что дальше?

- Use the source, Luke!

Что дальше?

- Use the source, Luke!
- Linux Kernel Development (книга про ядро, можно использовать как справочник или путеводитель)

Что дальше?

- Use the source, Luke!
- Linux Kernel Development (книга про ядро, можно использовать как справочник или путеводитель)
- [kernelnewbies.org](http://kernelnewbies.org) – сайт, посвящённый начинающим ядерщикам.

## Что дальше?

- Use the source, Luke!
- Linux Kernel Development (книга про ядро, можно использовать как справочник или путеводитель)
- [kernelnewbies.org](http://kernelnewbies.org) – сайт, посвящённый начинающим ядерщикам.
- [eudypsula-challenge.org](http://eudypsula-challenge.org) – серия задач, с помощью которых можно научиться писать код для ядра.

Что дальше?

- Use the source, Luke!
- Linux Kernel Development (книга про ядро, можно использовать как справочник или путеводитель)
- [kernelnewbies.org](http://kernelnewbies.org) – сайт, посвящённый начинающим ядерщикам.
- [eudypsula-challenge.org](http://eudypsula-challenge.org) – серия задач, с помощью которых можно научиться писать код для ядра.

И отдельно:

- Linus Torvalds – Just for fun

Спасибо за внимание!

[illegible]