

Linux kernel [module] programming

Саютин Дмитрий
cdkrot@yandex.ru

Летняя Компьютерная Школа

30.07.2016

Это handout версия доклада.

Она отличается от основной отсутствием пауз – содержимое всех слайдов показывается сразу полностью, а также несколькими дополнительными слайдами, в которых записана устная часть.

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

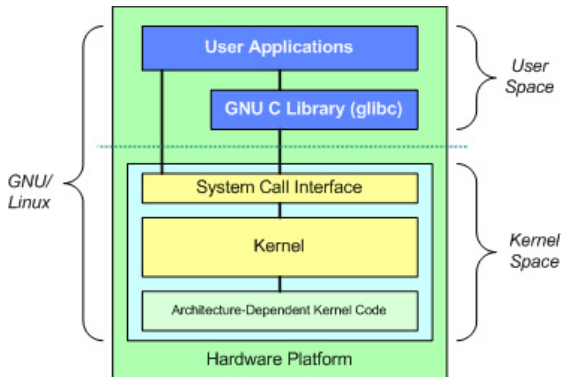
Введение и история

Что же такое ядро операционной системы?

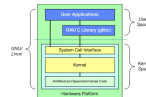
Что же такое ядро и зачем оно нужно?

- 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



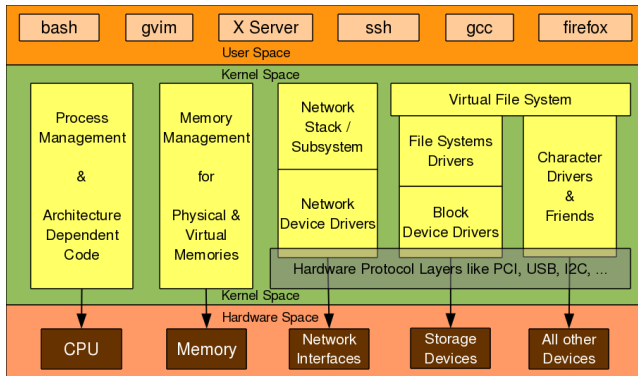
Linux architecture



Картинка иллюстрирует систему взаимодействия ядра с пользовательским пространством.

- Сначала пользовательское приложение, само или с помощью библиотеки языка C, делает системный вызов.
- Системный вызов – это такой интерфейс к ядру, по поведению как функция (принимает аргументы и возвращает значение), но технически более сложная.
- Представьте, что вы пишете hello world, вы даже не задумываетесь, что ваш printf в конце концов закончится как системный вызов записи в ядре.
- Продолжаем идти вниз, ядро принимает системный вызов и исполняет его. Ядро состоит из обобщённого кода на C, а также архитектурно-зависимой части.
- И в самом конце запрос доходит до железа.

Linux architecture



└ Linux architecture



Вторая картинка больше фокусируется на различных подсистемах ядра, а их довольно много:

- Управление процессами
- Управление памятью
- Сетевой стек
- Драйвера устройств
- Драйвера файловых систем
- ...

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.

- 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 -

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 :-).

└─ Linux: Первый email

```
Hallo 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 my
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 more people would
want. Any suggestions are welcome, but I won't promise I'll
implement them :)

Linux (torvalds@osna.helsinki.fi)

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

Minix - Это 'образовательная' операционная система, была популярной в те времена.

Minix была платной ОС, но с открытым исходным кодом.

Вносить изменения в код было можно, но шарить изменения можно было только в формате патчей к изначальному коду.

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.

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

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>

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: 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("Dmitry Informatics School");
MODULE_DESCRIPTION("Hello world module");
```

- Как и любой код на C, этот начинается с подключения заголовочных файлов.
- Дальше мы пишем функцию инициализации модуля, `__init` означает, что функцию можно удалить после завершения загрузки.
- `Printk` – это аналог `printf`, печатает в системный лог, `KERN_INFO` определяет приоритетность сообщения.
- `Ноль` означает отсутствие ошибок, ошибки же кодируются отрицательными числами.

└ Hello world: main.c

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("Dmitry Informatics School");
MODULE_DESCRIPTION("Hello world module");
```

- Дальше следует функция выхода. В ней надо освободить все ресурсы, что удерживает модуль, но у нас таких пока нет.
- `__exit` – это аналог `__init` для выхода.
- `module_init` и `module_exit` помечают соответствующие функции как точки входа и выхода.
- И в самом конце следует мета информация, не следует её недооценивать.
- Её потом можно восстановить из собранного модуля.

- 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),)
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: 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 H=$PWD \
modules
clean:
make -C /lib/modules/$(uname -r)/build H=$PWD \
clean
endif
```

- Данный makefile работает в двух режимах:
 - первый режим используется, когда нам нужно рассказать системе сборки ядра из какие файлы собирать.
 - второй, собственно, запускает систему сборки ядра, тем самым позволяя просто собирать модуль из консоли.
 - Использование if-а и специальной переменной, которую определяет система сборки ядра, позволяет определить какой режим нужен.
- obj-m определяет какой конечный модуль мы хотим собрать.
- hello-y определяет, что модуль hello состоит из файла main.o (который собирается из main.c), сюда можно дописать другие файлы модуля, если они есть.

└ 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
```

- Теперь ко второму режиму, здесь мы определяем две 'цели':
all и clean.
- В обоих из них мы просим систему сборки ядра о помощи.
- С помощью 'uname -r' мы подставляем актуальную версию ядра в путь
- С помощью поля M мы передаём текущую директорию, чтобы ядро поняло какой именно модуль надо собирать.
- Примечание по поводу слэшей. Такой слэш съедает символ перевода строки. Я его поставил, чтобы Makefile влез в слайд, в реальной жизни его можно убрать и записать всё в одну строку.

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
```

```
#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");
```

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.
- 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>

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 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
};
```

- Как всегда начинаем с заголовочных файлов
- Затем следует функция, цель которой вывести наш файл.
- Здесь нужно отметить, что мы пользуемся вспомогательной утилитой `seq_file`, которая позволяет нам с помощью минимума кода создать простой файл. Чтобы сообщить ей содержимое файла мы вызываем `seq_printf`.
- Вообще, чтобы определить файл нужно написать поддержку многих системных вызовов:
 - `open`
 - `read`
 - `write` (который, возможно, просто откажется записывать что-либо)
 - `seek`

└─ 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
};
```

- Но с помощью `seq_file` нам достаточно написать функцию, показывающую файл.
- Дальше идёт функция – `file_open`, она открывает файл для дальнейшего взаимодействия. Здесь мы просто просим `seq_file` о помощи.
- Дальше объявляем структуру `file_ops` – в неё записаны указатели на обработчики различных событий.
- Синтаксис здесь немного необычный, но так в GNU C можно объявлять структуры с инициализацией полей.

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: 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("Dmitry Informatica School");  
MODULE_LICENSE("GPL");  
MODULE_DESCRIPTION("Proc file example");  
  
module_init(proc_init);  
module_exit(proc_exit);
```

- Далее следует функция загрузки модуля, мы пытаемся создать новый файл, и если не удалось, то обрываем загрузку модуля с ошибкой.
- 0666 - это права доступа, в частности доступ всем на чтение и запись.
- NULL - Это указатель на директорию предка, в данном случае мы кладём файл в корень прос, поэтому NULL.
- Затем функция выгрузки модуля. Удаляем созданный файл.
- Ну и традиционная мета информация.

Proc example: running

```
make
sudo insmod proc.ko

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

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

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;

// 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: code

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

// variable, type, syafe mode
module_param(name, charp, 0000);
// user-friendly name, variable, type, syafe 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;
}
// ...
```

- Данный пример основан на предыдущем, поэтому часть кода пропущена.
- Объявляем параметры, обязательно нужно написать значение параметра по-умолчанию.
- С помощью `module_param{, _named}` мы сообщаем ядру о существовании параметров.
- 0000 означает не давать менять данный параметр во время работы модуля.
- 0600 позволяет менять параметр только пользователю root

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. 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.
- `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`.
- 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
```

```
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;
}
```

```
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;
}
```

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

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.

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.

- 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.

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 сложны. Они особенно сложны 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 рис).
 - После перехода на C++ в '92 сразу появилось много жалоб о падении производительности.

Ссылка на текст письма

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

И отдельно:

- Linus Torvalds – Just for fun

Спасибо за внимание!

```

1  linux/kernel/panic.c
2  Copyright (C) 1991, 1992 Linus Torvalds
3
4  This function is used through-out the kernel (including the initrd) to
5  indicate a major problem.
6
7  #include <linux/config.h>
8  #include <linux/module.h>
9  #include <linux/sched.h>
10 #include <linux/kernel.h>
11 #include <linux/slab.h>
12 #include <linux/sysvipc.h>
13 #include <linux/initrd.h>
14 #include <linux/init.h>
15 #include <linux/initrd.h>
16
17 #if !defined(CONFIG_SMP)
18 #define panic_timeout 60
19 #else
20 #define panic_timeout 30
21 #endif
22
23 #define panic_timeout 60
24 #define panic_timeout 30
25
26 EXPORT_SYMBOL(panic_timeout);
27
28 struct notifier_block *panic_notifier_list;
29
30 EXPORT_SYMBOL(panic_notifier_list);
31
32 static int __init panic_setup(char *str)
33 {
34     panic_timeout = simple_strtoul(str, NULL, 10);
35     return 0;
36 }
37
38 #define panic_timeout 60
39 #define panic_timeout 30
40
41 #define panic_timeout 60
42 #define panic_timeout 30
43
44 #define panic_timeout 60
45 #define panic_timeout 30
46
47 #define panic_timeout 60
48 #define panic_timeout 30
49
50 #define panic_timeout 60
51 #define panic_timeout 30
52
53 #define panic_timeout 60
54 #define panic_timeout 30
55
56 #define panic_timeout 60
57 #define panic_timeout 30
58
59 #define panic_timeout 60
60 #define panic_timeout 30
61
62 #define panic_timeout 60
63 #define panic_timeout 30
64
65 #define panic_timeout 60
66 #define panic_timeout 30
67
68 #define panic_timeout 60
69 #define panic_timeout 30
70
71 #define panic_timeout 60
72 #define panic_timeout 30
73
74 #define panic_timeout 60
75 #define panic_timeout 30
76
77 #define panic_timeout 60
78 #define panic_timeout 30
79
80 #define panic_timeout 60
81 #define panic_timeout 30
82
83 #define panic_timeout 60
84 #define panic_timeout 30
85
86 #define panic_timeout 60
87 #define panic_timeout 30
88
89 #define panic_timeout 60
90 #define panic_timeout 30
91
92 #define panic_timeout 60
93 #define panic_timeout 30
94
95 #define panic_timeout 60
96 #define panic_timeout 30
97
98 #define panic_timeout 60
99 #define panic_timeout 30
100
101 #define panic_timeout 60
102 #define panic_timeout 30
103
104 #define panic_timeout 60
105 #define panic_timeout 30
106
107 #define panic_timeout 60
108 #define panic_timeout 30
109
110 #define panic_timeout 60
111 #define panic_timeout 30
112
113 #define panic_timeout 60
114 #define panic_timeout 30
115
116 #define panic_timeout 60
117 #define panic_timeout 30
118
119 #define panic_timeout 60
120 #define panic_timeout 30
121
122 #define panic_timeout 60
123 #define panic_timeout 30
124
125 #define panic_timeout 60
126 #define panic_timeout 30
127
128 #define panic_timeout 60
129 #define panic_timeout 30
130
131 #define panic_timeout 60
132 #define panic_timeout 30
133
134 #define panic_timeout 60
135 #define panic_timeout 30
136
137 #define panic_timeout 60
138 #define panic_timeout 30
139
140 #define panic_timeout 60
141 #define panic_timeout 30
142
143 #define panic_timeout 60
144 #define panic_timeout 30
145
146 #define panic_timeout 60
147 #define panic_timeout 30
148
149 #define panic_timeout 60
150 #define panic_timeout 30
151
152 #define panic_timeout 60
153 #define panic_timeout 30
154
155 #define panic_timeout 60
156 #define panic_timeout 30
157
158 #define panic_timeout 60
159 #define panic_timeout 30
160
161 #define panic_timeout 60
162 #define panic_timeout 30
163
164 #define panic_timeout 60
165 #define panic_timeout 30
166
167 #define panic_timeout 60
168 #define panic_timeout 30
169
170 #define panic_timeout 60
171 #define panic_timeout 30
172
173 #define panic_timeout 60
174 #define panic_timeout 30
175
176 #define panic_timeout 60
177 #define panic_timeout 30
178
179 #define panic_timeout 60
180 #define panic_timeout 30
181
182 #define panic_timeout 60
183 #define panic_timeout 30
184
185 #define panic_timeout 60
186 #define panic_timeout 30
187
188 #define panic_timeout 60
189 #define panic_timeout 30
190
191 #define panic_timeout 60
192 #define panic_timeout 30
193
194 #define panic_timeout 60
195 #define panic_timeout 30
196
197 #define panic_timeout 60
198 #define panic_timeout 30
199
200 #define panic_timeout 60
201 #define panic_timeout 30
202
203 #define panic_timeout 60
204 #define panic_timeout 30
205
206 #define panic_timeout 60
207 #define panic_timeout 30
208
209 #define panic_timeout 60
210 #define panic_timeout 30
211
212 #define panic_timeout 60
213 #define panic_timeout 30
214
215 #define panic_timeout 60
216 #define panic_timeout 30
217
218 #define panic_timeout 60
219 #define panic_timeout 30
220
221 #define panic_timeout 60
222 #define panic_timeout 30
223
224 #define panic_timeout 60
225 #define panic_timeout 30
226
227 #define panic_timeout 60
228 #define panic_timeout 30
229
230 #define panic_timeout 60
231 #define panic_timeout 30
232
233 #define panic_timeout 60
234 #define panic_timeout 30
235
236 #define panic_timeout 60
237 #define panic_timeout 30
238
239 #define panic_timeout 60
240 #define panic_timeout 30
241
242 #define panic_timeout 60
243 #define panic_timeout 30
244
245 #define panic_timeout 60
246 #define panic_timeout 30
247
248 #define panic_timeout 60
249 #define panic_timeout 30
250
251 #define panic_timeout 60
252 #define panic_timeout 30
253
254 #define panic_timeout 60
255 #define panic_timeout 30
256
257 #define panic_timeout 60
258 #define panic_timeout 30
259
260 #define panic_timeout 60
261 #define panic_timeout 30
262
263 #define panic_timeout 60
264 #define panic_timeout 30
265
266 #define panic_timeout 60
267 #define panic_timeout 30
268
269 #define panic_timeout 60
270 #define panic_timeout 30
271
272 #define panic_timeout 60
273 #define panic_timeout 30
274
275 #define panic_timeout 60
276 #define panic_timeout 30
277
278 #define panic_timeout 60
279 #define panic_timeout 30
280
281 #define panic_timeout 60
282 #define panic_timeout 30
283
284 #define panic_timeout 60
285 #define panic_timeout 30
286
287 #define panic_timeout 60
288 #define panic_timeout 30
289
290 #define panic_timeout 60
291 #define panic_timeout 30
292
293 #define panic_timeout 60
294 #define panic_timeout 30
295
296 #define panic_timeout 60
297 #define panic_timeout 30
298
299 #define panic_timeout 60
300 #define panic_timeout 30
301
302 #define panic_timeout 60
303 #define panic_timeout 30
304
305 #define panic_timeout 60
306 #define panic_timeout 30
307
308 #define panic_timeout 60
309 #define panic_timeout 30
310
311 #define panic_timeout 60
312 #define panic_timeout 30
313
314 #define panic_timeout 60
315 #define panic_timeout 30
316
317 #define panic_timeout 60
318 #define panic_timeout 30
319
320 #define panic_timeout 60
321 #define panic_timeout 30
322
323 #define panic_timeout 60
324 #define panic_timeout 30
325
326 #define panic_timeout 60
327 #define panic_timeout 30
328
329 #define panic_timeout 60
330 #define panic_timeout 30
331
332 #define panic_timeout 60
333 #define panic_timeout 30
334
335 #define panic_timeout 60
336 #define panic_timeout 30
337
338 #define panic_timeout 60
339 #define panic_timeout 30
340
341 #define panic_timeout 60
342 #define panic_timeout 30
343
344 #define panic_timeout 60
345 #define panic_timeout 30
346
347 #define panic_timeout 60
348 #define panic_timeout 30
349
350 #define panic_timeout 60
351 #define panic_timeout 30
352
353 #define panic_timeout 60
354 #define panic_timeout 30
355
356 #define panic_timeout 60
357 #define panic_timeout 30
358
359 #define panic_timeout 60
360 #define panic_timeout 30
361
362 #define panic_timeout 60
363 #define panic_timeout 30
364
365 #define panic_timeout 60
366 #define panic_timeout 30
367
368 #define panic_timeout 60
369 #define panic_timeout 30
370
371 #define panic_timeout 60
372 #define panic_timeout 30
373
374 #define panic_timeout 60
375 #define panic_timeout 30
376
377 #define panic_timeout 60
378 #define panic_timeout 30
379
380 #define panic_timeout 60
381 #define panic_timeout 30
382
383 #define panic_timeout 60
384 #define panic_timeout 30
385
386 #define panic_timeout 60
387 #define panic_timeout 30
388
389 #define panic_timeout 60
390 #define panic_timeout 30
391
392 #define panic_timeout 60
393 #define panic_timeout 30
394
395 #define panic_timeout 60
396 #define panic_timeout 30
397
398 #define panic_timeout 60
399 #define panic_timeout 30
400
401 #define panic_timeout 60
402 #define panic_timeout 30
403
404 #define panic_timeout 60
405 #define panic_timeout 30
406
407 #define panic_timeout 60
408 #define panic_timeout 30
409
410 #define panic_timeout 60
411 #define panic_timeout 30
412
413 #define panic_timeout 60
414 #define panic_timeout 30
415
416 #define panic_timeout 60
417 #define panic_timeout 30
418
419 #define panic_timeout 60
420 #define panic_timeout 30
421
422 #define panic_timeout 60
423 #define panic_timeout 30
424
425 #define panic_timeout 60
426 #define panic_timeout 30
427
428 #define panic_timeout 60
429 #define panic_timeout 30
430
431 #define panic_timeout 60
432 #define panic_timeout 30
433
434 #define panic_timeout 60
435 #define panic_timeout 30
436
437 #define panic_timeout 60
438 #define panic_timeout 30
439
440 #define panic_timeout 60
441 #define panic_timeout 30
442
443 #define panic_timeout 60
444 #define panic_timeout 30
445
446 #define panic_timeout 60
447 #define panic_timeout 30
448
449 #define panic_timeout 60
450 #define panic_timeout 30
451
452 #define panic_timeout 60
453 #define panic_timeout 30
454
455 #define panic_timeout 60
456 #define panic_timeout 30
457
458 #define panic_timeout 60
459 #define panic_timeout 30
460
461 #define panic_timeout 60
462 #define panic_timeout 30
463
464 #define panic_timeout 60
465 #define panic_timeout 30
466
467 #define panic_timeout 60
468 #define panic_timeout 30
469
470 #define panic_timeout 60
471 #define panic_timeout 30
472
473 #define panic_timeout 60
474 #define panic_timeout 30
475
476 #define panic_timeout 60
477 #
```