**Модули ядра линукс**

Модуль ядра – фрагменты кода, которые могут быть загружены и выгружены в ядро по требованию. Они расширяют функциональность ядра без необходимости перезагрузки системы Модуль может быть настроен как встроенный в ядро, так и загружаемый во время работы ОС.

Пример модуля – драйвера устройств.

Модули ядра линукс находятся в папке lib/modules в зависимости от версии ядра создаются соответствующие папки.

Основные команды для управления модулями:

* lsmod – посмотреть загруженные модули
* modinfo – информация о модуле
* insmod – загрузить модуль
* rmmod – удалить модуль
* dpkg -S \*.ko | grep /lib/modules – вывод всех модулей в системе
* find /lib/modules/$(uname -r) -name \*.ko – для текущего ядра
* sudo modprobe modname – загрузить модуля ядра
* sudo insmod /lib/modules/kernel\_v/…/\*.ko – Загрузка модуля с помощью пути
* sudo rmmod modulename – удаление модуля
* dmesg – выводит буфер сообщений ядра (kernel ring buffer)

**Написание модулей**

* static int \_\_init name\_init(void) - Функция которая вызывается, когда модуль загружается в ядро (возвращает int)
* static void \_\_exit name\_exit(void)
* printk – печать сообщения в журнал консоли dmesg. Имеет приоритет
  + KERN\_EMERG ЯДРО - Система непригодна для использования
  + KERN\_ALERT - Действия должны быть приняты немедленно
  + КЕРН\_КРИТ - Критические условия
  + KERN\_ERR КЕРН\_ЕРР - Условия ошибки
  + KERN\_WARNING - Условия предупреждения
  + KERN\_NOTICE - Нормальное, но важное условие
  + KERN\_INFO - Информационный
  + KERN\_DEBUG - Сообщения уровня отладки

**Debugfs**

В ядре линукс многие драйверы и подсистемы поддерживают специальную файловую систему, которая называется debugfs. Она очень похожа на другие виртуальные файловые системы (например procfs), но у неё нет никакого функционального назначения, кроме предоставляемой отладочной информации. Файлы тут генерируются ядром, и их содержимое зависит от драйвера подсистемы.

Корневой каталог debugfs доступен только для root

Для того, чтобы в системе присутствовала данная директория, при сборке ядра необходимо указать:

CONFIG\_DEBUG\_FS=y

API debugfs документирован в исходных файлах ядра fs/debugfs/inode.c fs/debugfs/file.c. Для добавления драйвера необходимо сделать 3 шага:

* Определить структуру каталога
* Создать ф-и работы с файлами
* Зарегистрировать файлы в debugfs

debugfs обычно монтируется с помощью команды mount -t debugfs none /sys/kernel/debug

Код, использующий debugfs должен включать в себя «linux/debugfs.h»

Для того, чтобы создать каталог хранения файлов в debugfs необходимо создать структуру. В случае успеха создастся каталог с именем name под указанным родительским каталогом. Если parent = null то каталог будет создан от корня

struct dentry \*debugfs\_create\_dir(const char \*name, struct dentry \*parent);

Для создания файла, используем структуру. Здесь name – имя файла, mode – описание прав доступа, parent – каталог в котором должен хранится файл, fops – набор операций которые реализуют поведение файла (хотя бы чтение/запись).

struct dentry \*debugfs\_create\_file(const char \*name, umode\_t mode, struct dentry \*parent, void \*data, const struct file\_operations \*fops);

В ряде случаев создание набора файловых операций на самом деле не требуется. Debugfs предоставляет набор вспомогательных ф-й для простых ситуаций. Файлы, содержащие одно целое значение могут быть созданы с помощью любого из них.

* Значение в десятичной сс void debugfs\_create\_u64(const char \*name, umode\_t mode, struct dentry \*parent, u64 \*value);
* Значение в шестнадцатеричной сс void debugfs\_create\_x32(const char \*name, umode\_t mode, struct dentry \*parent, u32 \*value);
* Неизвестный размер значения void debugfs\_create\_size\_t(const char \*name, umode\_t mode, struct dentry \*parent, size\_t \*value);
* Создание переменной struct dentry \*debugfs\_create\_ulong(const char \*name, umode\_t mode, struct dentry \*parent, unsigned long \*value);
* Логические значения void debugfs\_create\_bool(const char \*name, umode\_t mode, struct dentry \*parent, bool \*value);

**net device**

* name - Имя этого устройства
* state - Состояние устройства. Поле включает в себя несколько флагов
* base\_addr - Базовый адрес ввода/вывода сетевого интерфейса.
* irq - Назначенный номер прерывания.
* if\_port - Порт, используемый в мультипортовых устройствах
* dma - Канал DMA, выделенный устройством.
* mtu – максималный блок передачи
* tx\_queue\_len - Максимальное количество кадров, которые могут быть поставлены в очередь в очередь передачи устройства

**PCI dev**

Шина PCI позволяет подключать одновременно большое количество устройств и часто состоит из нескольких физических шин, соедененных друг с другом с помощью PCI bridge. Каждая шина имеет свой номер, у устройства на шине также есть вой номер, у каждого устройства может быть несколько функций. Тогда путь к конкретному функционалу выглядит так <номер шины> <номер устройства> <номер функции>.

С каждымcd устройством на шине pci ос связывает структуру pci\_dev, для взаимодействия с ним.