**Ядро**

Ядро Linux монолитное, большая его часть хранится в одном файле. Однако, это не признак монолитного ядра, модули вполне могут храниться отдельно. Основная его особенность заключается в том, что оно обрабатывает все процессы, кроме пользовательских приложений. То есть управление процессами и памятью, драйверы, виртуальная файловая система, сетевой стек и многое другое – это всё заботы ядра, которые к тому же имеют самый высокий уровень доступа к аппаратной части компьютера.

Также стоит понимать, что ядро хоть и монолитное, но состоит из внутренних модулей, которые загружаются только по необходимости, а не все сразу. Некоторые модули хранятся отдельно от ядра, в основном это дополнительные драйверы устройств.

Как было сказано ранее, у монолитного ядра самый широкий спектр задач. На верхнем уровне ядро обрабатывает поступающие системные вызовы, которые являются интерфейсом между ядром и пользовательскими приложениями. На нижнем уровне ядро обрабатывает аппаратные прерывания, сигналы, поступающие от периферии, процессора, памяти и так далее.

Со стороны пользовательских приложений всё это выглядит как настоящее оборудование, с той лишь разницей, что общение с процессором и памятью происходит не напрямую, а с помощью системных вызовов. Для периферийных устройств имеются символьные и блочные ссылки в каталоге /dev, последние отличает то, что ни работают с блоками фиксированного размера.

Монолитное ядро - это один большой процесс, полностью работающий в одном адресном пространстве. Это один статический двоичный файл. Все службы ядра существуют и выполняются в адресном пространстве ядра. Ядро может вызывать функции напрямую. Примеры ОС на основе монолитного ядра: Unix, Linux.

В микроядрах ядро разбито на отдельные процессы, известные как серверы. Некоторые серверы работают в пространстве ядра, а некоторые - в пространстве пользователя. Все серверы хранятся отдельно и работают в разных адресных пространствах. Серверы вызывают "службы" друг у друга, отправляя сообщения через IPC (межпроцессное взаимодействие). Такое разделение имеет то преимущество, что если один сервер выходит из строя, другие серверы все еще могут работать эффективно. Примеры операционных систем на основе микроядра: Mac OS X

**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.

Ядро Linux позволяет драйверам оборудования, файловых систем, и некоторым другим компонентам быть скомпилированными отдельно - как модули, а не как часть самого ядра. Таким образом, вы можете обновлять драйвера не пересобирая ядро, а также динамически расширять его функциональность. А еще это значит, что вы можете включить в ядре только самое необходимое, а все остальное подключать с помощью модулей. Это очень просто.

Загружаемые модули ядра являются удобным способом модифицировать ядро, это может быть использовано злоумышленником при взломе системы для предотвращения обнаружения его процессов или файлов, позволяя ему сохранить контроль над системой.

Пример модуля – драйвера устройств.

Модули ядра Linux собираются только под определенную версию ядра

Находятся все модули в папке /lib/modules/. Учитывая, что модули рассчитаны только для определенной версии ядра, то в этой папке создается отдельная подпапка, для каждой установленной в системе версии ядра. Когда функциональность, предоставляемая модулем, больше не требуется, он может быть выгружен, чтобы освободить память и другие ресурсы.

Основные команды для управления модулями:

* lsmod - посмотреть загруженные модули
* modinfo - информация о модуле
* insmod - загрузить модуль
* rmmod - удалить модуль

**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

**Iostl**

В каждой ОС существует определенное количество системных вызовов. Иногда этих системных вызовов не хватает для каких то специфических действий. Делать для таких задач обычный системный вызов – плохо, так как их станет много и ими будет сложно управлять, поэтому существует iostl – файловый интерфейс (input/output control). У нас есть какое то устройство, с ним связан файл, для этого файла мы и будем писать вызов. Наиболее распространенный способ взаимодействия приложений с драйверами устройств. Он является гибким и легко расширяется путем добавления новых команд и может передаваться через символьные устройства, блочные устройства, а также сокеты и другие специальные файловые дескрипторы.

Для использования необходимо

* Создайте команду IOCTL в драйвере
* Напишите функцию IOCTL в драйвере
* Создайте команду IOCTL в пользовательском приложении
* Используйте системный вызов IOCTL в пользовательском пространстве

**Syscall**

Системный вызов — способ обращения программы пользовательского пространства к пространству ядра. Со стороны это может выглядеть как вызов обычной функции со своим собственным calling convention, но на самом деле процессором выполняется чуть больше действий, чем при вызове функции инструкцией call. Например, в архитектуре x86 во время системного вызова как минимум происходит увеличение уровня привилегий, замена пользовательских сегментов на сегменты ядра и установка регистра IP на обработчик системного вызова. Шаги при вызове syscall:

* Включается защита от чтения/записи/исполнения кода пользовательского пространства.
* Заменяется пользовательский стек на стек ядра, сохраняются callee-saved регистры.
* Выполняется обработка системного вызова
* Восстановление стека, регистров
* Отключение защиты
* Выход из системного вызова

Область ядра, в которую может перескочить процесс, называется system\_call. Процедура в этой области проверяет номер системного вызова, сообщая ядру о том, какое действие процесс запросил. Затем она просматривает таблицу системных вызовов (sys\_call\_table) в поиске адреса нужной функции ядра. Далее она эту функцию вызывает, и после того, как та возвращает результат, выполняет ряд системных проверок и делает возврат процессу (или к другому процессу, если время изначального истекло).

**Procfs**

Программы пространства пользователя в Linux не могут обращаться к ядру системы напрямую. Но для получения информации от ядра были созданы несколько специальных директорий с помощью которых любая программа или пользователь могут получить данные о состоянии компьютера и ядра. Это файловая система proc и sys.

Из этих папок можно получить любую информацию о вашей системе. Например сколько памяти подкачки сейчас используется, насколько велик размер кеша процессора, какие модули ядра загружены, сколько дисков или разделов доступно и т д. Все это можно получить в обычном текстовом виде из папки proc linux.

На самом деле папка proc - это совсем необычная папка. Ее не существует на диске или даже в оперативной памяти, как это делается в /tmp. Все поддиректории, файлы и хранящаяся в них информация генерируется ядром на лету, как только вы ее запрашиваете. Но работает все настолько прозрачно, что вы не заметите никакой разницы между обычной файловой системой и proc, если откроете ее с помощью файлового менеджера.

Только папка proc linux содержит файлы нулевого размера. А также у каждого файла будет текущая дата создания. Например файл /proc/meminfo будет содержать разные данные при каждом открытии, поскольку использование памяти постоянно колеблется.

**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.

debugfs - это простая в использовании файловая система на основе оперативной памяти, специально разработанная для целей отладки. Он существует как простой способ для разработчиков ядра сделать информацию доступной для пользовательского пространства. В отличие от /proc, который предназначен только для получения информации о процессе, или sysfs, которая имеет строгие правила для каждого файла с одним значением, debugfs вообще не имеет правил. Разработчики могут помещать туда любую информацию, которую они хотят.

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»

Основной интерфейс создания фалов в debugfs

* debugfs\_create\_file – для создания файла в файловой системе отладки.
* debugfs\_create\_dir – для создания каталога внутри файловой системы отладки.
* debugfs\_create\_symlink – для создания символической ссылки внутри файловой системы отладки.
* debugfs\_remove – для удаления записи debugfs из файловой системы отладки.

Также можно создавать числа (двоичные, десятичные, site\_t), blob – двоичные данные, массивы.

**net\_device**

Эта структура является ключевой структурой драйвера сетевого адаптера. Она содержит кучу разной информации, такую как имя сетевого интерфейса, адреса ввода-вывода и номер прерывания, которые использует устройство, статистическая информация, функции обратного вызова (callbacks) для взаимодействия с сетевым стеком ядра и многое другое.

На самом деле, вся эта структура - большая ошибка. Он смешивает ввод-вывод данные со строго "высокоуровневыми" данными, и он должен знать о почти каждая структура данных, используемая в модуле INET.

* name - Имя этого интерфейса
* 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
* min\_mtu: Минимальное значение MTU интерфейса
* max\_mtu: Максимальное значение MTU интерфейса
* tx\_queue\_len - Максимальное количество кадров, которые могут быть поставлены в очередь в очередь передачи устройства
* dev\_list: The global list of network devices
* dev\_id: используется для различения устройств, которые совместно используют тот же адрес канального уровня
* dev\_port: используется для разграничения устройств, которые совместно используют та же функция
* broadcast:

**PCI\_dev**

Шина PCI позволяет подключать одновременно большое количество устройств и часто состоит из нескольких физических шин, соединённых 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.

Поля структуры

* devfn - объединяет номера устройств и номера функций вместе
* vendor – id производителя
* device – id оборудования
* subsystem\_vendor;
* subsystem\_device;
* current\_state текущее состояние
* Status — биты этого регистра хранят информацию о различных событиях PCI шины.
* Revision ID — число, ревизия конкретной платы. Полезно в тех случаях, когда есть несколько ревизий устройства и различия необходимо учитывать в коде драйвера.
* Class Code — «волшебное» число, отображающее класс устройства, например: Network Controller, Display Controller и т. п.
* Base Address Registers — эти регистры, в количестве 6 штук, служат для определения того как и сколько памяти выделется устройству для процедур ввода/вывода. Этот регистр используется pci подсистемой ядра и обычно не интересен разработчикам драйверов.

**kmalloc|kfree, vmalloc|vfree, copy\_from\_user, copy\_to\_user**

* **kmalloc** – вы гарантированно получите непрерывные физические блоки памяти. Это может привести к сбою, если требуемый непрерывный физический блок не может быть найден. Физически непрерывная память, может понадобится если к буферу будет обращаться устройство DMA по физически адресуемой шине (например, PCI). Проблема в том, что многие системные вызовы не имеют возможности узнать, будет ли их буфер в конечном итоге передан на устройство DMA: как только вы передаете буфер другой подсистеме ядра, вы действительно не можете знать, куда он попадет. kmalloc ограничен размером буфера, который он может предоставить: 128 Кбайт
* **kfree** - Функция kfree() позволяет освободить память, ранее выделенную с помощью функции kmalloc().
* **vmalloc** - запрашивает пространство памяти с последовательными логическими адресами и прерывистыми физическими адресами. работает аналогично функции kmalloc(), за исключением того, что она выделяет страницы памяти, которые только виртуально смежные и необязательно смежные физически. Аналог malloc
* **vfree** - освободить память, выделенную vmalloc

Несмотря на то что физически смежные страницы памяти необходимы только в определенных случаях, большая часть кода ядра использует для выделения памяти функцию kmalloc(), а не vmalloc(). Это, в основном, делается из соображений производительности. Для того чтобы физически несмежные страницы памяти сделать смежными в виртуальном адресном пространстве, функция vmalloc() должна соответствующим образом заполнить таблицы страниц. Хуже того, страницы памяти, которые получаются с помощью функции vmalloc(), должны отображаться посредством страниц памяти, которые принадлежат к таблицам страниц (потому что выделяемые страницы памяти физически несмежные). Это приводит к значительно менее эффективному использованию буфера TLB[64], чем в случае, когда страницы памяти отображаются напрямую. Исходя из этих соображений функция vmalloc() используется только тогда, когда она абсолютно необходима, обычно это делается для выделения очень больших областей памяти. Например, при динамической загрузке модулей ядра, модули загружаются в память, которая выделяется с помощью функции vmalloc().

Мы не можем вызвать malloc в ядре. Так как мы не можем использовать библиотеки в ядре. Никак нет. Это означает, что ЛЮБАЯ функция, которую вы вызываете в ядре, должна быть определена в ядре. Linux не определяет malloc, поэтому вы не можете его использовать.

**copy\_to\_user copy\_from\_user**

Эти две функции очень часто используются в ядре и отвечают за копирование данных из пользовательского пространства в пространство ядра и копирование данных из пространства ядра в пространство пользователя.

copy\_from\_user(void \*to, const void \_\_user \*from, unsigned long n)

put\_user() и get\_user()используются для получения и размещения отдельных значений (таких как int, char или long ) из и в пользовательское пространство. Указатель на пространство пользователя никогда не должен быть просто разыменован: данные должны быть скопированы с использованием этих процедур. Оба возвращают -EFAULT или 0.

copy\_to\_user() и copy\_from\_user()являются более общими: они копируют произвольный объем данных в пользовательское пространство и из него.

**Зачем нужно разделение пространств, стек под ядро, lowmem, highmem.**

Если нет защиты между пространством ядра и пространством пользователя или между различными пользовательскими процессами, тогда любой может написать некоторый код, который преднамеренно или случайно изменит память в пространстве ядра или память в пространстве другого пользовательского процесса. Посмотрите на старый добрый MS-DOS, в котором не было этой защиты.

Стек - это структура данных LIFO. Мы можем выполнить push, чтобы добавить элемент в стек, и pop, чтобы удалить его. Структура данных стека может быть полезна при вычислении / преобразовании выражений, проверке синтаксиса, изменении порядка следования, обратном отслеживании и вызовах функций.

**Стек ядра является частью пространства ядра. Следовательно, он не доступен напрямую из пользовательского процесса.** Всякий раз, когда пользовательский процесс использует системный вызов, режим процессора переключается в режим ядра. Во время системного вызова используется стек ядра запущенного процесса.

Размер стека ядра настраивается во время компиляции и остается фиксированным. Обычно это две страницы (8 КБ) для каждого потока. Кроме того, для обработки внешних прерываний используются дополнительные [стеки прерываний](https://translated.turbopages.org/proxy_u/en-ru.ru.17878009-6399aa14-e8e8fc27-74722d776562/https/www.kernel.org/doc/html/latest/x86/kernel-stacks.html) для каждого процессора. Пока процесс выполняется в пользовательском режиме, эти специальные стеки не содержат никаких полезных данных.

Стек, используемый процессом пользовательского режима, находится в линейном адресном пространстве процесса;

Стек ядра - это когда процесс входит в пространство ядра из пользовательского пространства, изменяется уровень привилегий и необходимо переключить стек, тогда этот стек ядра используется в пространстве ядра. Поскольку путь управления ядром использует очень мало места в стеке, требуется всего несколько тысяч байтов стека режима ядра.

Следует отметить, что стек режима ядра используется только для процедур ядра, а ядро ​​Linux дополнительно предоставляет отдельные стеки аппаратных и программных прерываний для прерываний.

Ядро Linux разделяет это на 3/1 (также может быть 2/2 или 1/3 1) на пространство пользователя (высокая память) и пространство ядра (низкая память) соответственно.

Диапазон пользовательского пространства:

0x00000000 - 0xbfffffff

Каждый вновь созданный пользовательский процесс получает адрес (диапазон) внутри этой области. Пользовательские процессы, как правило, ненадежны и, следовательно, им запрещен доступ к пространству ядра. Кроме того, они считаются несрочными, поскольку, как правило, ядро пытается отложить выделение памяти для этих процессов.

Диапазон пространства ядра:

0xc0000000 - 0xffffffff

Процессы ядра получают здесь свой адрес (диапазон). Ядро может напрямую обращаться к этому 1 ГБ адресов (ну, не к полному 1 ГБ, для большого доступа к памяти зарезервировано 128 МБ).

Процессы, порожденные в пространстве ядра, являются надежными, срочными и предположительно безошибочными, запрос памяти обрабатывается мгновенно.

Каждый процесс ядра также может получить доступ к пользовательскому пространству, если пожелает. И для достижения этой цели ядро сопоставляет адрес из пользовательского пространства (с большим объемом памяти) со своим пространством ядра (с низким объемом памяти), упомянутые выше 128 МБ специально зарезервированы для этого.

**Примитивы синхронизации (Спинлок, семафор, мьютекс)**

Примитивы синхронизации - базовые алгоритмы, реализующую логику взаимодействия между двумя потоками управления. На базе примитивов либо за счёт системных средств планировщика строятся уже объекты синхронизации, т.е. повышается уровень абстракции.

Главное отличие примитива от объекта синхронизации в том, что логика работы объектов синхронизации скрыта от пользователя. Объект даёт наружу лишь интерфейс взаимодействия с ним, что происходит под капотом - скрыто. С примитивом же пользователь взаимодействует напрямую и на базе примитивов может строить свои объекты синхронизации. В разной среде с одним и тем же объектом синхронизации может быть ассоциирована разная логика (в Java сделано так, а в C# иначе). Примитивы синхронизации всегда ведут себя одинаково.

Обычно примитивы создаются с использованием механизмов более низкого уровня (например, атомарные операции, барьеры памяти, спин-локи, переключение контекста и т.д.).

Механизмы:

**Атомарные операции** - Операция в общей области памяти называется атомарной, если она завершается в один шаг относительно других потоков, имеющих доступ к этой памяти. Во время выполнения такой операции над переменной, ни один поток не может наблюдать изменение наполовину завершенным. Атомарная загрузка гарантирует, что переменная будет загружена целиком в один момент времени. Когда мы говорим, что операция атомарна, мы имеем в виду, что она не может быть прервана. Это означает, что во время выполнения операции не может произойти переключение потока, операция не может частично завершиться. Неатомарные операции не дают такой гарантии.

Алгоритм использует низкоуровневые машинные инструкции, такие как сравнение и замена (CAS, compare-and-swap, что обеспечивает целостность данных и по ним уже существует большое количество исследований).

**Спинлок** - цикл, проверяющий, не изменилась ли какая-то переменная, реализует активное ожидание. Нюанс: надо, чтобы доступ к этой переменной был организован определённым образом. Проблема в том, что разные потоки могут писать в переменную похожие значения, и нужно выяснить, кто это сделал и сделал ли.

Принцип невероятно прост. Спинлок — это просто переменная, которая содержит ноль или единицу (бывают варианты). Если ноль — спинлок свободен, и его можно захватить. Если не ноль — спинлок заперт, и нить, которая желает его захватить, будет ждать, крутясь (spin — вращение) в небольшом цикле непрерывной проверки освобождения спинлока. Вот,

spinlock создает блокировку, которая будет получена для защиты общего ресурса от изменения более чем одним процессом. В результате другие процессы, которые пытаются получить текущую блокировку, останавливаются ( "вращаются на месте"). Переключение контекста не разрешено, чтобы избежать взаимоблокировок. В результате spinlock следует использовать только в том случае, если блокировка будет получена только на очень короткий период времени, в противном случае количество загруженного ожидания, накопленное другими процессами, приводит к крайне неэффективной работе.

**Барьер памяти** - это класс инструкций, которые означают, что чтение / запись в память происходят в ожидаемом порядке. Например, "полный барьер" означает, что все операции чтения / записи до него будут  гарантированно сделаны до тех, которые стоят после барьера.

**Мьютексы** - более высокоуровневые спинлоки. Спинлок - всегда активное ожидание, а мьютексы могут быть и пассивными. Реализация мьютекса скрыта (инкапсулирована), в отличие от полностью открытого к управлению спинлока. Программный интерфейс мьютекса - это две команды - mutex.lock() и mutex.unlock(). Когда первый поток приходит на мьютекс, он блокирует его при помощи lock(), затем выполняется полезный код и мьютекс освобождается при помощи unlock(). Любой другой поток, пришедший до выполнения первым потоком операции unlock(), остановится на команде lock() и будет ждать освобождения мьютекса

**Семафор** - это целочисленное значение, которое может использоваться для передачи сигналов между процессами. Интерфейс семафора поддерживает три атомарные команды - инициализация, инкремент и декремент. При декременте может возникнуть блокировка потока (если после декремента значение семафора оказалось меньше либо равно 0). Таким образом, семафор инициализируется значением, равным количеству потоков, которые одновременно могут войти в участок кода.

Для блокировок, которые необходимо приобретать в течение относительно длительного периода времени, мы обращаемся к семафору. Как обычный примитив синхронизации, семафор основан на переменной. Эта переменная может быть увеличена или уменьшена, и ее состояние будет представлять возможность получения блокировки. Обратите внимание, что значение переменной не ограничено 0 и 1. Существует два типа семафоров: двоичный семафор; обычный семафор.

В первом случае значение семафора может быть только 1 или 0. Во втором случае значение семафора любое неотрицательное число. Это семафор подсчета и позволяет получить блокировку более чем для 1 процесса. Это позволяет нам вести учет доступных ресурсов, в то время как spinlock позволяет удерживать блокировку только для одной задачи. Помимо всего этого, есть еще одна важная вещь - семафор позволяет процессам которые его заняли перейти в спящий режим. Более того, когда процессы ожидают блокировки, планировщик может включить другой процесс.

Структура семафора состоит из трех полей:

* lock - spinlock для защиты данных семафора;
* count - количество доступных ресурсов;
* wait\_list - список процессов, которые ожидают получения блокировки