**1 - определения**

1 - код

1 – ссылки на статьи связанные с Python

1 - важно

**1**

**1**

**1**

**1**

**Udev** – демон, менеджер устройств.

Свойства:

* Работает на уровне пользователя
* Управляется событиями (пример события: подключение устройства к USB)
* Работает с файлами конфигурации
* Хранит только те файлы устройств, которые подключены к системе
* Запоминает конкретную информацию о подключенном устройстве

**Управление устройствами с помощью утилиты udevadm:**

Udevadm info – Вывод информации о свойствах устройств:

Lsmod – показывает информацию о модулях

Lsinfo – показывает информацию о конкретном модуле

Lsusb – показывает информацию об устройствах шины USB

Опции команды udevadmv:

Info - вывод информации из БД

Trigger - запрос события для устройства

Settle - ожидания завершения окончания всех событий

Control - управление демоном

Monitor - мониторинг событиями

Test - симуляция запуска события

Пример использования утилиты udevadm:

Udevadm info –a –p /sys/class/net/ipV4/enp

-a – выводит атрибуты

-p – по такому пути

Udevadm monitor --property –kernel

Мониторинг ядра по состоянию.

Udev работает с 3-мя виртуальными файловыми системами:

devfs, procfs, sysfs

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Виртальные файловые системы** | | |
| **Devfs** | **Sysfs** | **Procfs** |
| **Функции** | Содержит специальные файлы устройств, которые можно считать указателями на драйверы этих устройств, когда процесс обращается к файлу устройства, он начинает работать с драйвером этого устройства | Выводит на уровень пользователя информацию о присутствующих устройствах и драйверах | Содержит иерархическое представление о всех процессах в системе |
| **Точка монтирования** | /dev | /sys | /proc |
| **Каталоги** | * + **Block –** файлы блочных устройств   + **Bus** – файлы устройств шин   + **Pts –** файлы устройств псевдо-терминалов | * **devices** – все устройства зарегистрированные в ядре * **bus** – шины * **block** – блочные устройства * **drivers** – драйверы * **class** – классы устройств | **PID**[[1]](#footnote-1) – информация о процессе по PID  **Devices** – перечень настроенных устройств  **Mounts** – смонтированные файловые системы  **Sys[[2]](#footnote-2) –** доступная для редактирования информация о системе |
| **Примеры файлов** | **Mem** – образ физической памяти  **Null** – пустое устройство  **Urandom** – файл (символьное псевдоустройство)псевдогенератора случайных чисел  **Random –** файл генератора случайных чисел  **Tty** – терминал  **Pts** – псевдотерминал  **Sdx –** блочное устройство |  | **proc/Cpuinfo** – сведения о ЦПУ  **/proc/sys/net/ipv4/ip\_forward** – файл определяющий включена ли фукнция айпи форвадрдинга |

Таблица 1. Виртуальные файловые системы.

**Dev/**

В каталоге файловой системы devfs храняться все **файлы** подключенных к компьютеру **устройств**. Эти **файлы устройств**, иначе называемые специальными файлами устройств, представляют собой ссылки на драйвера устройств.

Файлы каталога /dev разделены по типам:

* -link
* -directory
* -блочное устройство (отправляет и принимает данные блоками)
* -character, символьное устройство (потоком данных)

**Работа приложения c устройством:**

Для работы приложения с устройством, приложение обращается к файлу устройства, потом драйвер устройства считывает запрос со специального файла устройства и передает данные на физическое устройство. Обратный путь в приложение проходит по аналогичному пути в обратную сторону. На рисунке 1 представлен пример схемы передачи данных для выполнения команды CAT вывода содержимого файла /etc/resolv.conf

1. Запрос команды CAT в терминале
2. Утилита выполняет запрос файла к драйверу через файл блочного устройства sda2 (жесткий диск)
3. Драйвер ищет файл на физическом диске и читает его содержимое
4. Драйвер передает данные через файл устройства псевдо-терминала в эмулятор терминала 6, где они будут отображены

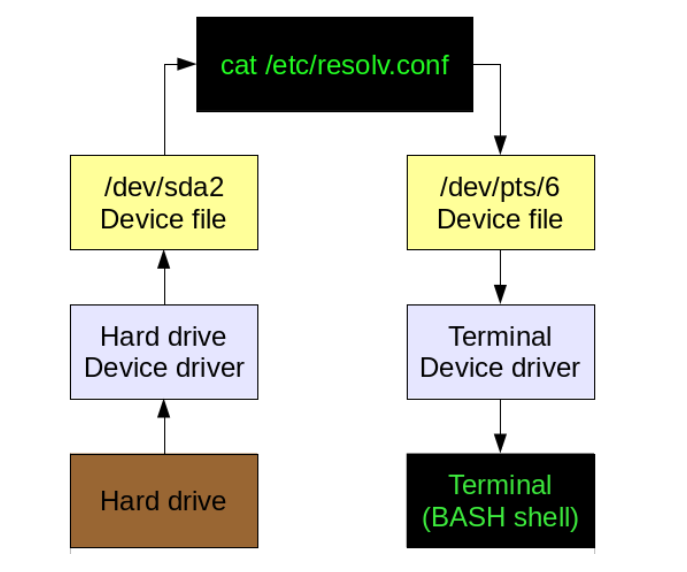


Рисунок 1. Схема передачи данных физическому устройству для выполнения команды CAT.

Примечание: В каталоге /dev не содержаться файлы устройств сетевой карты и видеокарты так как они работают с пакетами, а не с битами.

**Практика:**

Код команды перенаправления вывода действия команды (например, htop) в пустую папку null. Также перенаправление всех ошибок в ту же папку.

Htop > dev/null 2&1(CentOS)

> - перезапись

>> - до запись

Пример вывода 1й строки файла random:

Head -1 /dev/random > ~/tests

Выводится 1я строка файла random и записывается в файл test.

Команда head с ключом -1 выводит первую строку файла, если использовать другие (-2, -3 и т д.) числа то будет выводить все строки до определенного в ключе номера строки

**Управление модулями ядра:**

Модули ядра – объекты, содержащие код, который расширяет функционал ядра. (драйверы оборудования, файловые системы и другие компоненты расширения ядра)

Все модули находятся в папке /lib/modules

Управление модулями осуществляется посредством данных команд:

Lsmod – выводит информацию о модулях ядра

Modinfo – информацию о конкретном модуле ядра

Rmmod – жесткое удаление модуля ядра

Insmod – жесткая установка модуля ядра

Modprobe – деликатное удаление и установка модуля ядра

**Процесс загрузки компьютера с OS Linux.**

**Определения:**

**MBR и GPT – это таблицы разделов**

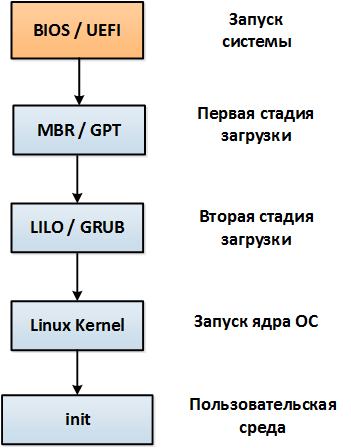


Рисунок 2. Этапы загрузки компьютера.

На рисунке 2 изображены все этапы загрузки компьютера, описание каждого этапа приведено ниже:

1. **Запуск системы BIOS/UEFI**

При включении компьютера центральный процессор переходит на адрес BIOS’а и загружает его. BIOS или UEFI проводит проверки оборудования и выбирает согласно своим настройкам носитель информации.

1. **Первая стадия загрузки**

На носителе BIOS или UEFI находит главную загрузочную запись MBR (Master boot record) или GPT (Guide partition table) в котором находиться загрузчик

**MBR** — это главная загрузочная запись, хранящаяся на жестком диске

**«GUID Partition Table (GPT)** является стандартным форматом размещения таблиц разделов на физическом жестком диске. Он является частью **Extensible Firmware Interface (EFI)** (Расширяемый Микропрограммный Интерфейс) — стандарта, предложенного Intel на смену отжившего BIOS, одного из последних реликтов первозданной IBM PC. EFI использует GPT там, где BIOS использует Главную загрузочную запись (MBR)

1. **Вторая стадия загрузки**

Загрузчик загружает ядро и initrd.

initrd — это Initial RAM Disk, он же временный диск в оперативной памяти

initrd используется самим ядром в качестве временной корневой файловой системы, пока kernel не загрузится в реальную примонтированную файловую систему. Этот временный диск также содержит необходимые для загрузки драйверы, позволяющие получить доступ к разделам дисков и другому оборудованию

1. **Запуск ядра ОС**

Проходят все инициализационные процедуры ядра и initrd. Далее запускается иницализационный процесс **init**

1. **Запуск процесса init**

Процесс **init** является родительским процессом, который запускает все последующие процессы.

Когда система стартует, вначале запускается ядро, которое тоже представляет собой процесс. Для него, как и для всех процессов, создается структура task\_struct, которая имеет идентификатор PID равный 0. В конце процесса инициализации ядра порождается процесс init, который имеет идентификатор 1. Он первоначально запускается как поток ядра, выполняет некоторые системные операции (такие как монтирование корневой файловой системы и открытие системной консоли), после чего выполняет программу инициализации системы, которую он ищет в одном из файлов /sbin/init, /etc/init или /bin/init на диске. Программа init использует конфигурационный файл /etc/inittab для определения того, какие процессы еще должны быть запущены. Эти новые процессы в свою очередь могут запускать другие процессы и так далее.

**Команды работы с ядром:**

Dmesg – команда для вывода всех сообщений ядра

Команды для просмотра log’а ядра, то-есть событий которые произошли от начала загрузки до конца загрузки:

Journalctl –b (systemd)

/var/log/dmesg (init)

**Варианты работы родительского процесса init:**

* В стиле **SysV**
* В стиле **SystemD**
* В стиле **Upstart**

Для определения варианта работы инициализационного процесса рассмотрим команду **pstree**:

Pstree

Данная команда показывает дерево процессов, в котором можно посмотреть родительский процесс.

Также в папке корневой директории boot можно найти папку загрузчика ядра и initrd

**SysV**

В данном стиле **init** является родительским процессом инициализации системы на одном из заданных уровней инициализации системы (runlevel).

**Задачи:** Управление процессом загрузки, переключение между режимами загрузки, управление службами.

**Уровни управления по умолчанию (runlevel) -**  уровни функционирования по умолчанию операционной системы, в которых реализована система инициализации типа Unix’ового:

1. Выключение
2. Однопользовательский режим для отладки
3. Родные режимы для выполнения Debian’а или Ubuntu. Внутри них можно выбрать либо графический режим, либо режим командной строки
4. Режим для Red hat/Suse, по умолчанию текстовый режим.
5. WildCard (программируемый)
6. Графический режим для Red hat/Suse
7. Перезагрузка

**inittab**

Только в инициализации системы в режиме sysv существует файл инициализации inittab, который находится в директории /etc/

В нем можно увидеть уровни загрузки и изменить уровень по умолчанию.

Примечание: Запуск любого уровня – это определенный скрипт, который запускается из папки /etc/rc.d/rc

**Файлы «rc.»**

В директории /etc с помощью команды находим файлы скриптов, запускающийся при запуске системы в определенном уровне инициализации:

Ls –l | grep rc.

В данной директории перед файлами стоят аббревиатуры K (Kill process) и S (Start process). Из их названия следует какие запускают процесс, а какие его выключают.

**Основные команды:**

Init – управление уровнями инициализации загрузки

Tellinit – управление урованями инициализации загрузки, работает и в старых дистрибутивах в отличие от init

Wall – вывож сообщения пользователям системы

Halt – выключение компьютера, после чего питание нужно выключать в ручную

Reboot – перезагрузка компьютера

Shutdown – запланированное выключение. Данная команда может выполнять все, что и две предыдущие команды, но является более гибкой

Runlevel – просмотр текущего уровня инициализации

**Службы**

Все скрипты, запускаемы при старте системы лежат в папке /etc/init.d

Остановка службы:

./network stop

Service network stop

./ - запуск из текущей директории

Запуск службы:

./network start

Service network start

Service reload –перезапуск с учетом изменений конфигурационных файлов

**SystemD**

В данном стеле init является родительским процессом инициализации системы в ускоренном режиме, за счет параллельного запуска задач.

Стиль работы инициализации Systemd работает с такими понятиями как **юниты** и **таргеты.**

**Units**

**Юнит –** это текстовый ini-файл с описанием. Файл разделен на секции, внутри секции задаются параметры.

* **Service –** службы
* **mount –** точки монтирования
* **device –** устройства
* **socket –** сокеты

Юниты находятся в директории по умолчанию /ust/lib/system.

В данной директории есть две папки user и system, для пользователя и системы соответственно

Управляемые юниты находятся в директории /etc/system

**Targets**

Таргеты или цели – это некоторая группировка юнитов, последовательность вызова юнитов.

**Boot таргеты**

|  |  |  |
| --- | --- | --- |
| **Runlevel** | **Target** | **Описание** |
| 0 | Poweroff. Target | Выключение |
| 1 | Rescue.target | Однопользовательский режим |
| 2,4 |  | Настраиваемый режим |
| 3 | Multi-user.target | Многопользовательский режим |
| 5 | Graphical.target | Графика |
| 6 | Reboot.target | перезагрузка |

Система таргетов обратно совместима с классической системой инициализации SysV. Это означает, что работают команды telinit

Systemctl list-units --type=target – запущенные таргеты

Systemctl isolate name target – переключение на другой таргет name

Systemctl set-default –f name target – установить по умолчанию

Systemctl isolate reboot target – аналогия команды telinit 6 в sysv

Для изменения, запускаемого target по умолчанию, можно изменить настройки загрузчика boot.target, а также командой:

Systemctl set-default –f multiuser.target

**Журналирование**

Журналированием занимается служба **journald**, которая собирает информацию из разных источников событий и привязывает ее к конкретным юнитам

Journalctl –f – управление журнальным демоном

Journal –n 10 – показать последние 10 событий

Journal \_UID – вывод события по UID пользователя

**Упрвавление питанием компьютера**

Systemctl reboot/poweroff/suspend/hibernate/hybrid-sleep

**Управление юнитами**

Systemctl start/stop/reload/restart unit

**Управление демоном systemd**

Systemctl **–** отвечает за все действия system

Systemctl list-units **–** показывает все запущенные юниты

Systemctl --failed **–** показывает юниты, которые не запустились

Systemctl list-units --type=service – просмотр запущенных сервисов

Systemctl status name **–** просмотр статуса службы name

Systemctl start/stop/reload/restart name – старт, остановка перезагрузка с измененными параметрами, перезагрузка службы name

**Upstart**

В данном стиле init является родительски процессом инициализации системы основанный на отслеживании событий.

Upstart контролирует запуск демонов и служб при загрузке системы.

Задачи и службы запускаются по событиям. Сами события могут генерироваться задачами и служюами. Службы могут быть перезапущены в автоматическом режиме, если были завершены в аварийном.

Можно создавать свои события

**Upstart** полностью обратно совместим c SysV.

**Upstart** оперирует такими понятиями, как службы и задачи.

Примечание: отличие службы от задачи в том, что служба перезапускается если она аварийно завершена, а задача нет.

Каталог файлов заданий (jobs) - **/etc/init**

Каждый файл задания отвечает за запуск отдельного демона или службы. Имеет расширение. Conf

Утилита, управляющая upstart - **initctl**

Уровни инициализации остались как в SysV.

**Задания –** это скрипты запуска слуюжб, которые запускаются по условиям

Cat /etc/init/rc-sysinit.conf – изменение уровня инициализации по умолчанию

Init show-config – показывает какие есть задачи

Init check-config –проверяет файлы заданий на конфликт

**Процессы и демоны в Linux [1]**

Процесс, в самом первом приближении — это программа, выполняющаяся в оперативной памяти компьютера. Но такая формулировка как бы подразумевает, что речь идет только о наборе машинных инструкций, последовательно выполняемых процессором.

В Linux процессы делятся на три категории:

* фоновые процессы
* потоки
* пользовательские процессы

**Дескрипторы, контексты и структура данных типа task\_struct.**

Для хранения всех данных, которые необходимо запоминать в целях организации работы процессов, в памяти, выделенной для ядра, создается для каждого процесса особая структура данных типа **task\_struct** (структура задачи). В ней можно выделить следующие функциональные группы данных:

* идентификационная информация о процессе (PID, PPID и т. д.);
* статус процесса;
* информация для планировщика;
* информация для организации межпроцессорного взаимодействия;
* ссылки и связи процесса;
* информация о времени исполнения и таймеры;
* информация об используемых процессом ресурсах файловой системы;
* информация о выделенном процессу адресном пространстве;
* контекст процесса – информация о состоянии регистров процессора, стеке и т.д.

Элементы списка task\_struct можно разделить на две группы: дескрипторы процесса и его контекст. Дескрипторы содержат более оперативные данные для планирования процессов.

Под **контекстом** процесса понимают совокупность той информации, которая необходима для организации переключения между процессами, а именно:

* Указатели на адресное пространство процесса в режиме задачи. Сюда входят указатели на сегменты кода, данных и стека, а также указатели на области разделяемой памяти и динамических библиотек.
* Окружение процесса, т. е. перечень заданных для данного процесса переменных с их текущими значениями.
* Аппаратный контекст процесса, то есть значения общих и ряда системных регистров процессора. Сюда относятся состояния счетчика выполняемых команд (указатель на адрес очередной исполняемой инструкции), указатель стека и так далее.
* Указатели на каждый открытый процессом файл, а также указатели на два каталога – домашний (или корневой) каталог процесса и его текущий каталог. Счетчики числа обращений в индексных дескрипторах этих каталогов увеличиваются на единицу при создании процесса (при смене текущего каталога), в силу чего вы (или другой процесс) не можете удалить эти каталоги, пока процесс их не «освободит».

Можно сказать, что контекст процесса и его виртуальное адресное пространство образуют как бы «виртуальный компьютер», в котором и исполняется процесс.

**Файлы дескрипторы с PID 0,1,2:**

В Linux библиотека libc открывает для каждого запущенного приложения(процесса) 3 файл дескриптора, с номерами 0,1,2.

* Файл дескриптор 0 называется STDIN и ассоциируется с вводом данных у приложения
* Файл дескриптор 1 называется STDOUT и используется приложениями для вывода данных, например, командами print
* Файл дескриптор 2 называется STDERR и используется приложениями для вывода данных, сообщающих об ошибке

Если в вашей программе вы откроете какой-либо файл на чтение или запись, то скорее всего вы получите первый свободный ID и это будет номер 3.

Список файл дескрипторов можно посмотреть у любого процесса, если вы знаете его PID.

Например, откроем консоль с bash и посмотрим PID нашего процесса:

echo $$

В выводе увидим PID нашего процесса. Далее можно посмотреть файлы дескрипторы данного процессора из директории /proc:

Ls –lah /proc/PID\_процесса/fd/

**Как рождаются процессы**

Новые процессы создаются в Linux методом «клонирования» какого-то уже существующего процесса, путем вызова системных функций clone(2) и fork(2). Процедура порождения нового процесса выполняется в режиме ядра и происходит следующим образом.

1. Создается новая структура task\_struct в таблице процессов ядра и содержание такой же структуры старого (или текущего) процесса копируется в новую структуру.
2. Ядро назначает идентификатор (PID) нового процесса. PID – это уникальное положительное число, которое присваивается каждому процессу при его рождении. Именно по этим идентификаторам система различает процессы.
3. Увеличиваются счетчики открытия файлов (порожденный процесс наследует все открытые файлы родительского процесса).
4. После того, как процесс создан, запускается выполняемая им программа с помощью одного из вариантов системного вызова exec. Далее происходит выполнение следующих системных функций fork, exec и так далее.

Приведенное описание позволяет сделать два важных вывода:

* во-первых, видно, что еще до того, как какая-то программа начинает исполняться, создается новый процесс, который после его создания и «выполняет программу»;
* во-вторых, каждый процесс порожден каким-то другим процессом, то есть для каждого процесса однозначно определен его "родитель" (или "предок"), для которого данный процесс является "дочерним" (или "потомком").

**Демоны, потоки и процессы.**

Демон:

**Системные процессы** являются частью ядра и всегда находятся в оперативной памяти. Такие процессы не имеют соответствующих им программ в виде исполняемых файлов и запускаются особым образом при инициализации ядра системы.

*Примерами* системных процессов являются планировщик процессов, диспетчер свопинга, диспетчер буферного кэша, диспетчер памяти ядра. Такие процессы являются фактически потоками ядра.

**Демоны** отличаются от обычных процессов только тем, что они работают в не интерактивном режиме. Если с обычным процессом всегда ассоциирован какой-то терминал или псевдотерминал, через который осуществляется взаимодействие процесса с пользователем, то демон такого терминала не имеет. Демоны обычно используются для выполнения сервисных функций, обслуживания запросов от других процессов, причем не обязательно выполняющихся на данном компьютере. Пользователь не может непосредственно управлять демонами, он может влиять на их работу, только посылая им какие-то задания, например, отправляя документ на печать.

**Прикладные процессы** – это все остальные процессы, выполняющиеся в системе. Как правило, эти процессы порождаются в рамках сеанса работы пользователя. В каждом таком сеансе работы вначале запускается оболочка (командный интерпретатор) shell.

Процессы могут запускать ("внутри себя") отдельные **потоки** или, в буквальном переводе с английского, – **нити (thread)**.

Потоки — это параллельно выполняемые части одной программы.

Потоки появились как логическое продолжение понятия процесса. Во-первых, появились некоторые задачи внутри ядра, которые требовалось выполнять параллельно с остальными работающими процессами. Во-вторых, ядро современной операционной системы должно уметь работать одновременно на нескольких процессорах. Эти две проблемы решаются посредством разделения некоторых частей ядра на отдельные потоки управления. Потоки ядра не имеют своего собственного адресного пространства и работают внутри пространства ядра. Фактически, они представлены лишь стеком данных и набором регистров, что позволяет ядру очень быстро переключаться между ними. Потоки ядра порождаются с помощью функции kernel\_thread(), которая делает системный вызов **clone(2)** в режиме ядра. Потоки ядра обычно имеют высокий приоритет и не имеют пользовательского адресного пространства, они получают прямой доступ к адресному пространству ядра. Идентификаторы этих потоков находятся в начале диапазона числовых значений идентификаторов.

**Идентификационные данные процесса**

Идентификаторы используются, в частности, для организации доступа к структурам ядра, хранящим информацию о процессах. А именно, отдельные записи типа task\_struct, соответствующие процессам, взаимосвязаны двумя способами:

* как **упорядоченный массив**, индексированный по идентификаторам процессов;
* как **кольцевой двухсвязный список**, в котором элементы ссылаются друг на друга посредством указателей next\_task (предыдущий) и prev\_task (последующий).

Здесь имеется в виду то, что список дескриптора процесса task\_struct содержит ссылки на PID родительского процесса PPID (Parent Process ID) процесса, описанного в task\_struct, и процесс рожденный этим процессом.

Для каждого процесса запоминаются также реальный и эффективный идентификаторы пользователя и реальный и эффективный идентификаторы группы. Реальный идентификатор пользователя для процесса определяется идентификатором пользователя, запустившего процесс. Эффективный идентификатор процесса обычно совпадает с реальным. Однако в некоторых случаях процессу требуются дополнительные права для получения доступа к некоторым системным ресурсам (в первую очередь к ресурсам файловой системы). В таком случае права процесса могут быть расширены. Примером такого расширения прав служит случай запуска на выполнение исполняемого файла, для которого установлен атрибут SUID. В таком случае эффективный идентификатор запускаемого процесса будет равен идентификатору владельца исполняемого файла (которым может быть, например, администратор). Реальный и эффективный идентификаторы группы для процесса задаются аналогичным образом.

**Состояния процессов**

Каждый запущенный процесс в любой момент времени находится в одном из следующих состояний (которое называют еще статусом процесса)

* **Активен (R=Running)** – процесс находится в очереди на выполнение, то есть либо выполняется в данный момент, либо ожидает выделения ему очередного кванта времени центрального процессора.
* **«Спит» (S=Sleeping)** – процесс находится в состоянии прерываемого ожидания, то есть ожидает какого-то события, сигнала или освобождения нужного ресурса.
* Находится в состоянии непрерываемого ожидания **(D=Direct)** – процесс ожидает определенного («прямого») сигнала от аппаратной части и не реагирует на другие сигналы;
* **Приостановлен (T)** – процесс находится в режиме трассировки (обычно такое состояние возникает при отладке программ).
* **«Зомби» (Z=Zombie)** – это процесс, выполнение которого завершилось, но относящиеся к нему структуры ядра по каким-то причинам не освобождены. Одной из причин их появления в системе может быть следующая ситуация. Обычно освобождение структур ядра, относящихся к процессу, выполняет процесс-родитель после получения от потомка сигнала о завершении. Но бывают случаи, когда родительский процесс завершается раньше дочернего. Процессы, не имеющие родителя, называются "сиротами". "Сироты" автоматически усыновляются процессом init, который и принимает сигналы об их завершении. Если процесс-родитель или init по каким-то причинам не может принять сигнал о завершении дочернего процесса, то процесс-потомок превращается в "зомби" и получает статус Z. Процессы-зомби не занимают процессорного времени (т. е. их выполнение прекращается), но соответствующие им структуры ядра не освобождаются. В некотором смысле это «мертвые» процессы. Уничтожение таких процессов — одна из обязанностей системного администратора.

Кроме перечисленных выше данных в структуре типа task\_struct хранятся и другие параметры (или атрибуты) процесса.

**Средства межпроцессорного взаимодействия**

Хотя процессы изолированы друг от друга, они могут обмениваться данными с помощью предоставляемых системой средств межпроцессорного взаимодействия. К таким средствам относятся **каналы (pipes)**, **именованные каналы (FIFO)**, **сообщения (messages)**, **разделяемая память (shared memory)**, **семафоры (semaphores)**, **сигналы (signals)** и **сокеты (sockets)**.

**Каналы**

Канал обеспечивает однонаправленную передачу данных между двумя процессами, причем только между «родственными» процессами.

*Например*, когда выполняется команда

cat myfile | wc

оба процесса cat и wc создаются процессом shell, и являются родственными. Поэтому каналы не могут использоваться в качестве средства межпроцессорного взаимодействия между независимыми процессами.

**FIFO**

FIFO тоже являются средством однонаправленной передачи данных, но, в отличие от программных каналов, имеют имена (поэтому и называются именованными каналами), которые позволяют независимым процессам получить к этим объектам доступ. FIFO является отдельным типом файла в файловой системе Linux.

**Сообщения**

Очереди сообщений размещаются в адресном пространстве ядра и являются разделяемым системным ресурсом. Каждая очередь сообщений имеет свой уникальный идентификатор. Процессы могут записывать сообщения в очередь и читать сообщения из очереди. При этом процесс, пославший сообщение в очередь, не обязан ожидать приема этого сообщения другим процессом (или процессами). Он может даже завершиться, а его сообщение будет прочитано позже.

**Семафоры**

Семафоры, собственно говоря, не являются средством передачи данных. Они выполняют в межпроцессорном взаимодействии вспомогательную роль и служат для организации одновременного использования разделяемых данных несколькими процессами. Если два процесса читают один набор данных и выполнение процессов — последовательное, то это, очевидно, не создает проблем. Если же два процесса пытаются изменить один и тот же набор данных, то результат уже будет зависеть от того, в какой последовательности выполняется считывание и запись этих данных. Для управления такими процессами и вводятся семафоры.

В простейшем случае семафор представляет собой просто счетчик, содержащий 0 или единицу. Значение счетчика, равное 1, означает доступность соответствующего ресурса (например, файла или страницы виртуальной памяти). Если же в счетчике 0, значит, ресурс занят, и операция недопустима.

Семафоры должны быть доступны различным процессам, поэтому они размещаются в адресном пространстве ядра и операции с ними осуществляются через интерфейс системных вызовов.

**Разделяемая память**

Интенсивное использование таких механизмов межпроцессорного взаимодействия, как каналы, FIFO и очереди сообщений, может привести к значительному падению производительности системы. Это связано с тем, что данные, передаваемые с помощью этих механизмов, копируются из буфера передающего процесса в буфер ядра, а затем в буфер принимающего процесса. Механизм разделяемой памяти позволяет избавиться от излишних пересылок, данных, предоставляя двум и более процессам доступ к одной и той же области оперативной памяти. Проблема совместного обращения двух процессов к одним и тем же данным решается с помощью семафоров.

**Сокеты**

Сокет (от англ. “socket” – разъем, гнездо) – это унифицированный интерфейс взаимодействия процессов с использованием стека сетевых протоколов в ядре. Сокет представляет собой виртуальный объект, аналогичный в некотором смысле сетевому интерфейсу. Такой объект характеризуется семейством протоколов и типом сокета, который представляет собой не что иное, как тип передаваемого через сокет потока данных. В настоящее время сокеты в Linux допускают использование примерно тридцати семейств протоколов и 6 типов сокетов (все они описываются в файле /usr/include/bits/socket.h)

Сокеты имеют соответствующий интерфейс доступа в файловой системе (имя, подобное имени файла или, точнее, устройства) и обращение к ним, так же как к обычным файлам, осуществляется через дескрипторы. Однако, в отличие от обычных файлов, сокеты представляют собой виртуальный объект, который существует, пока на него ссылается хотя бы один из процессов.

Чтобы использовать сокет, какой-то процесс должен его создать, а другие процессы – установить соединение с данным сокетом. Для создания сокетов используется системный вызов soket(2). Затем сокету присваивается имя путем вызова системного вызова bind(2). Другие процессы для установления соединения с сокетом используют системный вызов connect(2). По всем этим вызовам имеются man-странички, из которых вы можете получить дополнительную информацию о сокетах.

**Сигналы**

Сигналы — это средство, с помощью которого процессам можно передать сообщения о некоторых событиях в системе. С помощью сигналов можно осуществлять такие акции управления процессами, как приостановка процесса, запуск приостановленного процесса, завершение работы процесса. Пользователи тоже могут "общаться" с процессами путем посылки им сигналов. Когда мы нажимаем комбинацию клавиш <Ctrl>+<C>, чтобы завершить выполнение какой-то программы, мы фактически посылаем соответствующему процессу сигнал "Завершить работу". Завершаясь, процесс посылает родительскому процессу сигнал о своем завершении. Таким образом, сигналы напоминают программные прерывания, являясь средством, с помощью которого нормальное выполнение процесса может быть прервано. Сами процессы тоже могут генерировать сигналы, с помощью которых они передают определенные сообщения ядру и другим процессам.

Сигналы принято обозначать номерами или символическими именами. Все имена начинаются на SIG, но эту приставку иногда опускают: например, сигнал с номером 1 обозначают или как SIGHUP, или просто как HUP. Всего в Linux существует 63 разных сигнала, их перечень можно посмотреть по команде

kill –l

Если после этой опции указать номер сигнала, то будет выдано его символическое имя, а если указать имя, то получим соответствующий номер.

Существуют несколько причин генерации сигналов или ситуаций, в которых отправляются сигналы.

* Терминальные прерывания. Нажатие пользователем некоторых комбинаций клавиш вызывает отправку определенного сигнала процессу, связанному с текущим терминалом.
* Особые ситуации. Когда выполнение процесса вызывает особую ситуацию, например, деление на ноль, процесс получает от ядра соответствующий сигнал.
* Межпроцессорное взаимодействие. Процесс может отправить сигнал другому процессу с помощью системного вызова kill(2).
* Управление заданиями. Командные интерпретаторы, поддерживающие управление заданиями, используют сигналы для манипулирования текущим и фоновыми заданиями.
* Квоты. Когда процесс превышает установленную для него квоту использования тех или иных ресурсов, ему отправляется соответствующий сигнал.
* Уведомления о готовности устройств или наступлении других событий тоже посылаются процессам с помощью сигналов.
* Если процесс установил таймер, то ему будет послан сигнал, когда значение таймера станет равным нулю.

Когда процесс получает сигнал, то возможен один из двух вариантов развития событий. Если в исполняемой процессом программе для данного сигнала определена подпрограмма обработки, то вызывается эта подпрограмма. В противном случае ядро выполняет от имени процесса действие, определенное по умолчанию для данного сигнала. Вызов подпрограммы обработки называется *перехватом* сигнала. Когда завершается выполнение подпрограммы обработки, процесс возобновляется с той точки, где был получен сигнал.

Можно заставить процесс игнорировать или блокировать некоторые сигналы. Игнорируемый сигнал просто отбрасывается процессом и не оказывает на него никакого влияния. Блокированный сигнал ставится в очередь на выдачу, но ядро не требует от процесса никаких действий до разблокирования сигнала. После разблокирования сигнала программа его обработки вызывается только один раз, даже если в течение периода блокировки данный сигнал поступал несколько раз.

Следует заметить, что любая обработка сигналов процессом производится только тогда, когда процесс выполняется. То есть процесс не получит сигнал, пока не будет выбран планировщиком и ему не будут предоставлено время центрального процессора.

**Команды:**

Маршруты:

Route –n (n-numeric)

Netstat –n –r

Pwd – вывод текущего каталога

Ls (каталог) – вывод файлов в каталоге

Ключи: (-lh)

Установка новых пакетов:

Apt-get install

Apt install

Yum install

Решение проблемы Sub-process /usr/bin/dpkg returned an error code (1)

Способ 1: переконфигурировать базу данных пакетов

Первый метод, который вы можете попробовать, — перенастроить базу данных пакетов.

Вероятно, база данных повреждена при установке пакета.

Реконфигурация часто устраняет проблему.

sudo dpkg --configure -a

Способ 2: Использовать принудительную установку

Если установка ранее прервалась, вы можете попробовать установить пакет принудительно:

sudo apt-get install -f

Способ 3. Попробуйте удалить проблемный пакет.

sudo apt remove

Способ 4. Удалите информационные файлы с информацией о проблемном пакете

Это должно быть вашим последним прибежищем.

Вы можете попробовать удалить файлы, связанные с пакетом, из **/var/lib/dpkg/info.**

Вам нужно немного узнать о базовых командах Linux, чтобы выяснить, что происходит, и как использовать их с вашей проблемой.

В моем случае у меня была проблема с polar-bookhelof.

Поэтому я искал файлы, связанные с ним:

ls -l /var/lib/dpkg/info | grep -i polar-bookshelf

-rw-r--r-- 1 root root 2324811 Aug 14 19:29 polar-bookshelf.list

-rw-r--r-- 1 root root 2822824 Aug 10 04:28 polar-bookshelf.md5sums

-rwxr-xr-x 1 root root 113 Aug 10 04:28 polar-bookshelf.postinst

-rwxr-xr-x 1 root root 84 Aug 10 04:28 polar-bookshelf.postrm

Теперь мне нужно удалить эти файлы:

sudo mv /var/lib/dpkg/info/polar-bookshelf.\* /tmp

Приложения:

Приложение 1. **Уникальные идентификаторы.**

* **PID, идентификатор процесса ([англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA" \o "Английский язык) *process identifier*)** — уникальный номер (идентификатор) [процесса](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D1%86%D0%B5%D1%81%D1%81_(%D0%B8%D0%BD%D1%84%D0%BE%D1%80%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0)" \o "Процесс (информатика)) в [многозадачной](https://ru.wikipedia.org/wiki/%D0%9C%D0%BD%D0%BE%D0%B3%D0%BE%D0%B7%D0%B0%D0%B4%D0%B0%D1%87%D0%BD%D0%BE%D1%81%D1%82%D1%8C" \o "Многозадачность) [операционной системе (ОС)](https://ru.wikipedia.org/wiki/%D0%9E%D0%BF%D0%B5%D1%80%D0%B0%D1%86%D0%B8%D0%BE%D0%BD%D0%BD%D0%B0%D1%8F_%D1%81%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B0" \o "Операционная система). В ОС семейства [Windows](https://ru.wikipedia.org/wiki/Windows" \o "Windows) pid хранится в [переменной](https://ru.wikipedia.org/wiki/%D0%9F%D0%B5%D1%80%D0%B5%D0%BC%D0%B5%D0%BD%D0%BD%D0%B0%D1%8F_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)" \o "Переменная (программирование)) [целочисленного типа](https://ru.wikipedia.org/wiki/%D0%A6%D0%B5%D0%BB%D1%8B%D0%B9_%D1%82%D0%B8%D0%BF" \o "Целый тип). В ОС, соответствующих стандарту [POSIX](https://ru.wikipedia.org/wiki/POSIX" \o "POSIX), [тип](https://ru.wikipedia.org/wiki/%D0%A2%D0%B8%D0%BF_%D0%B4%D0%B0%D0%BD%D0%BD%D1%8B%D1%85" \o "Тип данных) переменной, хранящей pid, определяется каждой ОС индивидуально. Например, в ОС [Linux](https://ru.wikipedia.org/wiki/Linux" \o "Linux) pid хранится в переменной целочисленного типа (int).

Когда создается новый процесс, ядро присваивает ему идентификатор – PID. Когда выполнение процесса завершилось, идентификатор освобождается и может быть присвоен другому процессу. Присвоение идентификаторов происходит последовательно по возрастанию, начиная с единицы. Идентификатор нового процесса обычно больше, чем идентификатор процесса, созданного ранее. Если идентификатор уже достиг максимально возможного значения, следующий процесс получит минимальный из свободных идентификаторов и цикл продолжается. В системе не может быть двух процессов с одинаковыми идентификаторами и на время жизни процесса его идентификатор не меняется.

Приложение 2. Определения:

API (application programming interface) – программный интерфейс приложения. Описание способов, которыми одна программа может взаимодействовать с другой программой.

Библиографический список:

1. rus-linux.net/kos.php?name=/papers/proc/proc\_lin.html

1. это уникальное положительное число, которое присваивается каждому процессу при его рождении. Регистрируется в ядре. см. Приложение 1. Уникальные идентификаторы [↑](#footnote-ref-1)
2. Все изменения в директории /proc/sys не сохраняются после перезагрузки компьютера [↑](#footnote-ref-2)