Оглавление

[Лекция №1. Установка операционной системы Linux 2](#_Toc147175112)

[Лекция №2. Реализация многопользовательского режима в командном интерфейсе операционной системы Unix/Linux 8](#_Toc147175113)

[Лекция №3. Доступ к файловой системе через интерфейс системных вызовов 16](#_Toc147175114)

[Лекция №4. Работа с процессами в командной и программной средах операционной системы Linix 23](#_Toc147175115)

[Лекция №5. Ненадежные и надежные сигналы в системе Unix 28](#_Toc147175116)

[Лекция №6. Создание и использование программных потоков в системе Linux 33](#_Toc147175117)

[Лекция №7. Установка виртуального сервера Ubuntu Server 20.04 в среде Virtual Box 39](#_Toc147175118)

[Лекция №8. Создание рабочей среды в Arch Linux на базе файловых менеджеров окон 49](#_Toc147175119)

[Лекция №9. Создание неименованных каналов и приложений типа клиент-сервер в системе Linux 61](#_Toc147175120)

[Лекция 10. Создание простейшей командной оболочки ядра Linux на базе процессов 71](#_Toc147175121)

[Лекция №11. Семафоры стандарта Posix для синхронизации процессов и потоков в системе Linux 78](#_Toc147175122)

[Лекция №12. Создание командной оболочки ядра Linux на базе процессов и программных потоков 87](#_Toc147175123)

[Лекция №13. Создание командной оболочки ядра Linux на базе процессов, программных каналов и семафоров Posix 96](#_Toc147175124)

[Лекция №14. Приложения и компоненты графического интерфейса операционных систем семейства Linux 106](#_Toc147175125)

## Лекция №1. Установка операционной системы Linux

**Основные понятия и идеи стандарта POSIX**

Обеспечение мобильности (переносимости, портабельности) программного обеспечения (ПО) - задача исключительной важности и сложности; в наше время это обстоятельство едва ли нуждается в пространных обоснованиях. Один из общепринятых способов повышения мобильности ПО - стандартизация окружения приложений: предоставляемых программных интерфейсов, утилит и т.п. На уровне системных сервисов подобное окружение описывает стандарт POSIX (Portable Operating System Interface - мобильный интерфейс операционной системы); название предложено известным специалистом, основателем Фонда свободного программного обеспечения Ричардом Столмэном.

Стандарт POSIX описывает множество базовых, системных сервисов, необходимых для функционирования прикладных программ. Доступ к ним предоставляется посредством интерфейса, специфицированного для языка C, командного языка и общеупотребительных служебных программ.

У каждого интерфейса есть две стороны: вызывающая и вызываемая. Стандарт POSIX ориентирован в первую очередь на вызывающую. Его цель - сделать приложения мобильными на уровне исходного языка. Это значит, в частности, что при переносе C-программ на другую операционную платформу потребуется перекомпиляция. О мобильности выполнимых программ и/или объектных файлов речь не идет.

Стандарт POSIX отнюдь не ограничен рамками Unix-среды. Существуют операционные системы (ОС) "независимого происхождения" (например, системы реального времени), предоставляющие необходимые сервисы и тем самым поддерживающие выполнение POSIX-совместимых приложений. Можно утверждать, что следование стандарту POSIX облегчает перенос приложений практически на любую сколько-нибудь распространенную операционную платформу. Дополнительные усилия по повышению мобильности, прилагаемые на этапе разработки, безусловно, окупятся.

Определяя интерфейс к системным сервисам, POSIX оставляет за рамками рассмотрения их реализацию. В частности, не различаются системные вызовы и библиотечные функции. Не являются объектом стандартизации средства администрирования, аппаратные ограничения и функции, необходимые только суперпользователю, что еще раз подчеркивает направленность стандарта POSIX на приложения, а не на операционные системы.

POSIX нейтрален по отношению к системной архитектуре и разрядности процессора. Это очень важный аспект мобильности приложений.

Ориентация на международный стандарт языка C определила не только стиль описания функций, но и, до некоторой степени, направление развития спецификаций POSIX в плане синхронизации обоих стандартов.

Стандарт POSIX в редакции 2003-го года - весьма обширный, многогранный документ, где подробно рассматриваются следующие категории системных компонентов:

* средства разработки;
* сетевые средства;
* средства реального времени;
* потоки управления;
* математические интерфейсы;
* пакетные сервисы;
* заголовочные файлы;
* унаследованные интерфейсы.

Именно такой (на верхнем уровне, далеко не полный) репертуар должна предоставлять операционная система для работы приложения.

Важнейшим является понятие соответствия стандарту POSIX. Мы уже отмечали, что всякий интерфейс располагает двумя сторонами: вызывающей и вызываемой. Две стороны есть и у POSIX-соответствия: соответствие реализации (операционной системы) и приложения.

Реализация (операционная система), соответствующая стандарту POSIX, должна поддерживать все обязательные служебные программы, функции, заголовочные файлы с обеспечением специфицированного в стандарте поведения. Константа \_POSIX\_VERSION имеет значение 200112L.

ОС может предоставлять возможности, помеченные в стандарте в качестве дополнительных, а также содержать нестандартные функции. Если утверждается, что поддерживается некоторое расширение, это должно производиться непротиворечивым образом, для всех необходимых частей и так, как описано в стандарте.

Мы рассмотрим следующие основные понятия операционных систем, соответствующих стандарту POSIX:

* пользователь;
* файл;
* процесс;
* терминал;
* хост;
* узел сети;
* время;
* языково-культурная среда.

Это первичные понятия. Их нельзя строго определить, но можно пояснить с помощью других понятий и отношений. Для каждого из выделенных понятий будут описаны присущие им атрибуты и применимые к ним операции.

В тексте стандарта POSIX содержатся следующие пояснения основных понятий вместе со ссылками на атрибуты и операции.

1. У пользователя есть имя и числовой идентификатор.
2. Файл - объект, допускающий чтение и/или запись и имеющий такие атрибуты, как права доступа и тип. К числу последних относятся обычный файл, символьный и блочный специальные файлы, канал, символьная ссылка, сокет и каталог. Реализация может поддерживать и другие типы файлов.
3. Процесс - адресное пространство вместе с выполняемыми в нем потоками управления, а также системными ресурсами, которые этим потокам требуются.
4. Терминал (или терминальное устройство) - символьный специальный файл, подчиняющийся спецификациям общего терминального интерфейса.
5. Сеть - совокупность взаимосвязанных хостов.
6. Языково-культурная среда - часть пользовательского окружения, зависящая от языковых и культурных соглашений.

Для работы с большим числом сущностей всегда предоставляются механизмы группирования и построения иерархий. Существует иерархия файлов, группы пользователей и процессов, подсети и т.п.

Для написания программ, оперирующих с сущностями POSIX-совместимых систем, применяются командный интерпретатор (язык shell) и/или компилируемый язык C. В первом случае приложение может пользоваться служебными программами (утилитами), во втором - функциями. Функциональный интерфейс операционных систем естественно считать первичным, поскольку большинство служебных программ предназначены, по сути, для вызова той или иной функции. По этой причине далее мы будем рассматривать преимущественно уровень функций.

Основными операциями, применимыми к объектам ОС, являются чтение, запись и выполнение. Механизм прав доступа позволяет избирательно разрешать и запрещать осуществление подобных операций. Ранее в стандарте фигурировало понятие суперпользователя, не подверженного контролю прав доступа. В POSIX-2001 выбрана более гибкая формулировка - "имеющий соответствующие привилегии", что отражает прогресс в реализации ОС с расщеплением суперпользовательских возможностей.

В POSIX-совместимых ОС определены объекты, которые можно назвать вспомогательными; они помогают организовать взаимодействие между основными сущностями. Особенно широк спектр средств межпроцессного взаимодействия.

Процессы выполняются в определенном окружении, частью которого является языково-культурная среда (Locale), образованная такими категориями, как символы и их свойства, форматы сообщений, дата и время, числовые и денежные величины.

Как правило, с процессом ассоциированы по крайней мере три файла - стандартный ввод, стандартный вывод, стандартный протокол. Обычно стандартный ввод назначается на клавиатуру терминала, а стандартный вывод и стандартный протокол - на экран. Со стандартного ввода читаются команды и (иногда) исходные данные для них. На стандартный вывод поступают результаты выполнения команд. В стандартный протокол помещаются диагностические сообщения.

**Введение в операционные системы семейства Unix**

Первые вычислительные машины (на лампах) не имели операционных систем, а программы для них составлялись на машинных языках. В полупроводниковых электронных вычислительных машинах (ЭВМ) появились ассемблеры, макроязыки, библиотеки подпрограмм, а затем и языки высокого уровня. Операторы ЭВМ вручную запускали сначала трансляцию программы, а затем, посредством специальных программ–загрузчиков, загружали в оперативную память исполнимые коды программ и выполняли их.

Для автоматизации работы операторов и уменьшения простоев ЭВМ появились программы–мониторы для организации пакетной обработки программ (заданий). С увеличением быстродействия процессора появились системы мультипрограммирования – сначала для пакетной обработки, а затем для систем разделения времени (многотерминальных систем). Этот комплекс системных программ получил название операционной системы ЭВМ.

В рамках операционных систем были созданы файловая система, виртуальная память и аппаратная поддержка для программ операционной системы (система прерываний, защита памяти, привилегированный режим работы и др.). Появились сети ЭВМ, обеспечивающие удаленный доступ и удаленное выполнение заданий. И в этот период развития ЭВМ (в конце шестидесятых) была разработана операционная система Unix. Официально система Unix появилась в 1970 году.

Операционная система UNIX за 50 лет своего существования подтвердила правильность основных концепций, заложенных изначально в нее разработчиками. Она многократно модернизировалась и дополнялась, в результате чего в середине восьмидесятых появились две базовые версии системы UNIX: System V Realease 4 (SVR4) и 4BSD. На их основе затем было создано множество новых версий: FreeBSD, AIX, HP-UX, IRIX, Digital UNIX, SCO UNIX, Solaris, Minix, Linux и др. UNIX явилась первой действительно переносимой операционной системой, спецификация и реализация наиболее важных компонентов которой регламентируется стандартами Posix, XPG, System V, UNIX98 и др. Система UNIX послужила основой (прототипом) при создании операционных систем MS DOS и MS Windows. UNIX традиционно использовалась для изучения операционных систем в университетах, поскольку предоставляла исходные коды и свободно распространяемые версии своей системы. И сейчас такие версии системы UNIX, как, например, FreeBSD и Linux, предоставляют возможность свободного доступа к исходным кодам операционной системы.

**Ядро системы Linux**

Операционная система Linux имеет двухуровневую структуру, состоящую из ядра на нижнем уровне и приложений на верхнем уровне. Ядро (kernel) непосредственно взаимодействует с аппаратной частью компьютера, изолируя прикладные программы от особенностей ее архитектуры. Ядро имеет набор услуг, предоставляемых прикладным программам. К услугам ядра относятся операции ввода/вывода (открытия, чтения, записи и управления файлами), создания и управления потоками и процессами, их синхронизации и межпроцессного взаимодействия. Все приложения запрашивают услуги ядра посредством системных вызовов.

Приложения или задачи подразделяются на системные, определяющие функциональность операционной системы (регистрация пользователей, конфигурирование устройств, управление работой устройств, реализация консольного и графического интерфейса, интерпретация команд (скриптов) и компиляция программ), и прикладные, обеспечивающие пользовательский интерфейс системы Linux (например, текстовые редакторы и утилиты). Несмотря на внешнюю разнородность этих приложений, схемы их взаимодействия с ядром одинаковы. Важное место в этом взаимодействии занимает оболочка ядра, содержащая интерпретатор команд, реализующий командный интерфейс операционной системы.

Ядро ОС Linux монолитное, т.е. оно выполняется в одном адресном пространстве в режиме ядра. Тем не менее ядро Linux позаимствовало некоторые хорошие свойства микроядерной модели: в нем используется преемптивное (т.е. с вытеснением выполняющихся процессов более приоритетными) ядро, поддерживаются потоки пространства ядра и возможность динамической загрузки в ядро внешних бинарных файлов (модулей ядра). Ядро Linux не использует никаких функций микроядерной модели, которые приводят к снижению производительности: все выполняется в режиме ядра с непосредственным вызовом функций вместо передачи сообщений. Следовательно, операционная система Linux — модульная, многопоточная, а выполнение самого ядра можно планировать.

Ядро Linux поставляется в двух вариантах: стабильном (stable) и разрабатываемом (development). Версии стабильного ядра - это выпуски продукции промышленного уровня, которая готова для широкого использования. Новые стабильные версии ядра обычно выпускаются для исправления ошибок и для предоставления новых драйверов устройств. Разрабатываемые версии ядра, наоборот, подвержены быстрым изменениям, поскольку при их использовании разработчики экспериментируют с новыми решениями и часто вносят радикальные изменения в ядро.

Например, версия 2.6.10 определяет стабильное ядро, имеющее серию ядер 2.6 и редакцию 10. Если младшее (правое) число серии (число 6) четное, то версия ядра стабильная.

Архив исходного кода ядра в формате tar распространяется в сжатых форматах GNU zip (gzip) и bzip2. Формат bzip2 наиболее предпочтителен, так как обеспечивает больший коэффициент сжатия по сравнению с форматом gzip. Архив ядра в формате bzip2 имеет имя linux-x.у.z.tar.bz2, где х, у, z — это номер соответствующей версии исходного кода ядра. После загрузки исходного кода его можно декомпрессировать очень просто. Если tar-архив сжат с помощью GNU zip, то необходимо выполнить следующую команду.

$ tar xvzf linux-x.у.z.tar.gz

Если сжатие выполнено с помощью bzip2, то команда должна иметь следующий вид.

$ tar xvjf linux-x.у.z.tar.bz2

Обе эти команды позволяют декомпрессировать и развернуть дерево исходных кодов ядра в каталог с именем linux-x.y.z. Исходный код ядра обычно инсталлируется в каталог / u s r / s r c / l i n u x.

В сообществе разработчиков ядра Linux заплаты (patch) — это основной язык *общения.* Вы будете распространять ваши изменения исходного кода ядра в виде заплат и получать изменения кода от других разработчиков тоже в виде заплат. При данном рассмотрении наиболее важными являются *инкрементные заплаты* (incremental patch), которые позволяют перейти от одной версии ядра к другой. Вместо того чтобы загружать большой архив ядра, можно просто применить инкрементную заплату и перейти от имеющейся версии к следующей. Это позволяет сэкономить время и пропускную способность каналов связи. Для того чтобы применить инкрементную заплату, находясь в каталоге дерева исходных кодов ядра, нужно просто выполнить следующую команду.

$ patch -p1 < ../patch-х.у.z

Обычно заплата для перехода на некоторую версию ядра должна применяться к предыдущей версии ядра.

Как и все "уважающие себя" ядра Unix, ядро Linux написано на языке С. Однако ядро Linux написано не на чистом языке С в стандарте ANSI С. Наоборот, где это возможно, разработчики ядра используют различные расширения языка, которые доступны с помощью средств компиляции *gcc* (GNU Compiler Collection — коллекция компиляторов GNU, в которой содержится компилятор С, используемый для компиляции ядра).

Ядро Linux 5.8 является последней версией ядра. Однако в различных дистрибутивах системы Linux используются более ранние версии ядра.

**Дистрибутивы системы Linux**

Сначала версии Linux распространялись на обыкновенных дискетах - одна содержала ядро, а другая – корневую файловую систему и необходимые программы. Потом появились дистрибутивы, содержащие кроме этого программу установки операционной системы на компьютер. Компания Red Hat стала одной из первых, выпустивших свой дистрибутив системы Linux на лазерном компакт-диске. Отметим, что в настоящее время:

* основными дистрибутивами для настольных операционных систем являются Red Hat, Slackware и Debian, а все остальные – лишь производные от них. Так, Mandrake и ASPLinux произошли от Red Hat, дистрибутив ALT Linux взял за основу Mandrake, а Ubuntu основан на Debian. Потом на смену Red Hat пришел дистрибутив Fedora Core (сейчас просто Fedora), а на смену Mandrake – сначала Mandriva, а затем Mageia. Существует два типа версий Ubuntu: обычные и LTS (Long Term Support). Обычные версии выходят два раза в год, а LTS – только один раз. Однако техническая поддержка и обновление программ для LTS-дистрибутивов доступны на протяжении 5 лет;
* номер версии дистрибутива не совпадает с номером версии ядра – это принципиально разные вещи;
* самыми популярными дистрибутивами на сегодняшний день считаются Ubuntu и Fedora – для настольного применения (для дома и офиса), а также CentOS и Debian – для серверного. Также широко используется немецкий дистрибутив openSUSE, существующий в двух вариантах: Tumbleweed и Leap. Первый содержит все самое новое ПО и постоянно обновляется, а второй – только стабильное ПО;
* основными графическими интерфейсами (оболочками) ОС Linux являются GNOME и KDE. По умолчанию GNOME используется в Fedora, Ubuntu, Debian, а KDE – в openSUSE. Однако это можно изменить. Например, в Ubuntu для установки KDE следует выполнить в терминале следующую команду:

sudo apt install kubuntu-desktop

Эта команда установит интерфейс KDE со стандартным набором приложений. Чтобы получить KDE с полным набором приложений, вместо пакета kubuntu-desktop установите пакет kubuntu-full. В Fedora и в Ubuntu можно установить графические оболочки Cinnamom, MATE, XFCF, LXDE. Все эти интерфейсы отличаются ограниченной функциональностью и подходят для слабых компьютеров. Более или менее удобным из них считается LXDE.

GNU/Linux - мультинациональная ОС. И каждая страна создаёт свои дистрибутивы, которые используются как на рабочих станциях, так и на серверах. Россия не отстаёт, она создала несколько хороших (и не очень) дистрибутивов Linux:

* Rosa Linux - дистрибутив, основанный на ныне покойном Mandriva, и продолжающий его развитие. Данный дистрибутив имеет несколько редакций, расчитанных на разное применение. На базе проекта Rosa развивается дистрибутив OpenMandriva, который является "полигоном" для испытания нового ПО и технологий;
* Calculate Linux - это линейка корпоративных дистрибутивов, основанных на Gentoo, обладают простым и понятным установщиком, высоким качеством сборки и весьма широким набором предустановленного ПО;
* Runtu - это Российская сборка Ubuntu, ориентированная на русского пользователя. Система полностью русифицирована, весьма проста в установке, имеет хороший набор предустановленных приложений. Существует две редакции Runtu: Runtu XFCE - с легковесным графическим окружением Xfce, настроенным на привычный Windows-пользователю интерфейс; Runtu LITE - с оконным менеджером Openbox, ориентированная на старое и слабое оборудование;
* Russian Fedora Remix (или RFRemix) - сборка на основе дистрибутива Fedora;
* ALT Linux, отличительной особенностью которого является его пакетный менеджер: используются пакеты формата RPM, как в RedHat-производных дистрибутивах, но управляются они с помощью утилиты APT (Advanced Packaging Tool), которая является "родной" для Debian и его производных (таких как Ubuntu). ALT Linux также известен тем, что его рассылают во многие школы, а учебники информатики содержат задания именно для него. Дистрибутив имеет как общедоступные бесплатные редакции, так и сертифицированные версии для госструктур РФ. Дистрибутив Simply Linux является легковесной редакцией ALT Linux, содержащей большое количество обучающего и мультимедийного ПО, а также простой и удобный рабочий стол на базе Xfce. Разработка пакетов для ALT Linux ведётся в специальном репозитории Sysyphus;
* Astra Linux - операционная система специального назначения на базе Debian GNU/Linux, созданная для нужд российских силовых ведомств и спецслужб;
* Agilia Linux - дистрибутив, основанный на ныне не развиваемом MOPS Linux (который в свою очередь основан на Slackware). Основные принципы, которых придерживаются разработчики дистрибутива, заключаются в лёгкости установки и освоения системы, а также подборке наиболее стабильных программ. Формат пакетов — txz, в качестве менеджера пакетов используется mpkg;
* PupyRusLinux - это легковесный дистрибутив, созданный специально для слабого оборудования. Небольшой размер системы (около 120 мегабайт) позволяет ему полностью загрузиться в оперативную память, обеспечив максимальную производительность. PuppyRusLinux ориентирован на компьютеры с архитектурой х86, оптимизирован для обеспечения максимальной производительности и в силу низкой требовательности к оборудованию может вдохнуть «вторую» жизнь в устаревшие модели.

**Установка системы Linux в контейнер виртуальных машин**

Система Linux устанавливается либо в раздел файловой системы, либо в контейнер виртуальных машин. В первом случае Linux нужно устанавливать в раздел ранее установленной операционной системы (ОС) Windows.

При установке Linux из дистрибутива обычно необходимо зарезервировать следующую дисковую память:

для корневого раздела (/), где будут находиться файлы операционной системы и приложения, примерно 8 – 10 Гбайт;

## Лекция №2. Реализация многопользовательского режима в командном интерфейсе операционной системы Unix/Linux

**Файлы и каталоги**

В операционной системе Unix/Linux имеется два типа объектов: файлы и процессы (причем под файлами понимаются файлы, каталоги и т.д.).

Одно из существенных отличий Unix от Windows заключается в том, что в Windows для каждого устройства существует свое дерево каталогов, а в Unix существует единственное дерево, где каждое устройство является одной из ветвей этого дерева. Здесь / - это имя каталога - корня дерева.

Изображение выглядит как диаграмма, зарисовка, рисунок, линия

Автоматически созданное описание

Для каждого файла можно записать полный или относительный путь. Например, /usr/prog.c - это полный путь для файла prog.c, который начинается от корня дерева. Если же текущим каталогом является, например, каталог usr, то путь Ivanov/labs/lab2/add.cpp - это относительный путь для файла add.cpp.

В Unix системах имена основных каталогов являются стандартными, что существенно облегчает работу в операционной системе, ее админи­стрирование и переносимость. Упрощается инициализация и конфигурация системы, работа почтовых программ и печати. Нарушение структуры приводит к нарушению работы системы.

Рассмотрим назначение основных стандартных каталогов.

Корневой каталог "/" является основой любой файловой системы UNIX. Все остальные файлы и каталоги располагаются в рамках структуры, порожденной корневым каталогом, независимо от их физического местона­хождения

В каталоге /bin находятся наиболее часто употребляемые команды и ути­литы системы, как правило, общего пользования.

Каталог /dev содержит специальные файлы устройств, являющиеся интер­фейсом доступа к периферийным устройствам. Каталог /dev может содержать несколько подкаталогов, группирующих специальные файлы устройств одного типа. Например, каталог /dev/dsk содержит специальные файлы устройств для доступа к гибким и жестким дискам системы.

В каталоге /etc находятся системные конфигурационные файлы и многие утилиты администрирования. Среди наиболее важных файлов — скрипты инициализации системы. Эти скрипты хранятся в каталогах /etc/rcO.d, /etc/rcl.d, /etc/rc2.d и т. д, соответствующих уровням выполнения системы (run level), и управляются скриптами /etc/rcO, /etc/rcl, /etc/rc2 и т. д.

Во многих версиях BSD UNIX указанные каталоги отсутствуют, и загрузка сис­темы управляется скриптами /etc/rc.boot, /etc/re и /etc/re.local.В UNIX System V большинство исполняемых фай­лов перемещены в каталог /sbin или /usr/sbin.

В каталоге /lib находятся библиотечные файлы языка С и других языков программирования. Стандартные названия библиотечных файлов имеют вид libx.а (или libx.so), где х — это один или более символов, определяю­щих содержимое библиотеки.

В некоторых версиях Unix этот каталог разгружен и информация перенесена в /usr/lib.

*/lost+found -* этокаталог "потерянных" файлов. Ошибки целостности файловой системы, возникающие при неправильном останове UNIX или аппаратных сбоях, могут привести к появлению "безымянных" файлов, т.е. файлов структура и со­держимое которых являются правильными, однако отсутствует имя в каком-либо из каталогов.

Существуют программы проверки и восстановления файловых систем, которые находят данные файлы и помещают в этот каталог, давая им при этом номер вместо имени.

Стандартный каталог /*mnt* предназначен для временного связывания (монтирования) физиче­ских файловых систем к корневой для получения единого дерева логиче­ской файловой системы. Обычно содержимое каталога /mnt пусто, по­скольку при монтировании он перекрывается связанной файловой систе­мой.

Каталог */home* предназначен для размещения домашних каталогов пользо­вателей.

В каталоге /usr находятся подкаталоги различных сервисных подсистем:

* /usr/spool -системы печати, электронной почты и т.д.;
* /usr/bin -исполняемые фай­лы утилит UNIX;
* /usr/local- дополнительные программы, используемые на данном компьютере;
* /usr/inchide -файлы заголовков;
* /usr/man- элек­тронные справочники;
* /usr/man/shell- файлы оболочки.

В UNIX System V каталог */var* является заменителем каталога /usr/spool.

*/tmp* - это каталог хранения временных файлов, необходимых для работы различных подсистем UNIX. Обычно этот каталог открыт на запись для всех пользо­вателей системы.

**Управление доступом к файлам и каталогам**

Характерной особенностью Unix является то, что это многопользовательская система. Каждый файл имеет двух владельцев: владелец- пользователь (user-owner) и владелец-группа (group- owner). Важной особенностью является то, что владелец-пользователь может не являться членом группы, владеющей файлом. Третьей категорией пользователей являются все остальные (other) пользователи.

Значение прав доступа различно для разных типов файлов. Для файлов опе­рации, которые можно производить, следуют из самих названий прав досту­па. Например, чтобы просмотреть содержимое файла командой *cat,* поль­зователь должен иметь право на чтение (г). Редактирование файла, т. е. его изменение, предусматривает наличие права на запись (w). Наконец, для того чтобы запустить некоторую программу на выполнение, вы должны иметь соответствующее право (х). Исполняемый файл может быть как скомпили­рованной программой, так и скриптом командного интерпретатора shell. В последнем случае вам также понадобится право на чтение, поскольку при выполнении скрипта командный интерпретатор должен иметь возможность считывать команды из файла. Все сказанное, за исключением, пожалуй, права на выполнение, имеющего смысл лишь для обычных файлов и ката­логов, справедливо и для других типов файлов: специальных файлов уст­ройств, именованных каналов, и сокетов. Например, чтобы иметь возмож­ность распечатать документ, вы должны иметь право на запись в специаль­ный файл устройства, связанный с принтером. Для каталогов эти права имеют другой смысл, а для символических связей они вообще не использу­ются, поскольку контролируются целевым файлом.

Права доступа для каталогов не столь очевидны. Это в первую очередь связано с тем, что система трактует операции чтения и записи для ката­логов отлично от остальных файлов. Право чтения каталога позволяет вам получить имена (и только имена) файлов, находящихся в данном каталоге. Чтобы получить дополнительную информацию о файлах каталога (например, подробный листинг команды *Is* -/), системе придется "загля­нуть" в метаданные файлов^ что требует права на выполнения для катало­га. Право на выполнения также потребуется для каталога, в который вы захотите перейти (т. е. сделать его текущим) с помощью команды *cd.* Это же право нужно иметь для доступа ко всем каталогам на пути к ука­занному. Например, если вы установите право на выполнения для всех пользователей в одном из своих подкаталогов, он все равно останется не­доступным, пока ваш домашний каталог не будет иметь такого же права.

Права г и х действуют независимо, право х для каталога не требует нали­чия права г, и наоборот. Комбинацией этих двух прав можно добиться ин­тересных эффектов, например, создания "темных" каталогов, файлы кото­рых доступны только в случае, если пользователь заранее знает их имена, поскольку получение списка файлов таких каталогов запрещено. Данный прием, кстати, используется при создании общедоступных архивов в сети (FTP-серверов), когда некоторые разделы архива могут использоваться только "посвященными", знающими о наличии того или иного файла в каталоге.

Особого внимания требует право на запись для каталога. Создание и удале­ние файлов в каталоге требуют изменения его содержимого, и, следователь­но, права на запись в этот каталог. Самое важное, что при этом не учиты­ваются права доступа для самого файла. То есть для того, чтобы удалить не­который файл из каталога, не обязательно иметь какие-либо права доступа к этому файлу, важно лишь иметь право на запись для каталога, в котором находится этот файл. Этот порядок можно изменить, если использовать дополнительные атрибуты файлов.

Дополнительные атрибуты также устанавливаются утилитой *chmod()* Например, для установки атрибута SGID для файла filel необходимо выполнить команду

$ chmod *g+s* filel

Для компиляции и выполнения программы hello.c в консольном окне (из командной строки) используются следующие команды:

сс hello.c

a.out

Здесь по команде cc создается исполнимый файл a.out, который затем запускается на выполнение. Если в переменной PATH среды интерпретатора отсутствует текущий каталог, в команде запуска на выполнение необходимо указать полный или относительный путь к файлу a.out, например,

/home/virtualuser/lab2/a.out

или

./a.out.

Для добавления текущего каталога в список путей поиска файлов, находящийся в переменной PATH, достаточно ввести команду

PATH=$PATH:.

При создании многомодульных программ, т.е. программ, состоящих из нескольких исходных файлов, каждый из модулей отдельно компилируется в объектный модуль (файл с расширением .o), а затем выполняется компоновка объектных модулей в загрузочный модуль. В среде системы Linux используется флажок –с для указания компилятору (cc, gcc или g++) о том, что необходимо выполнить только компиляцию модуля. Например, для модуля add.cpp могут быть выполнены следующие команды

g++ -c add.cpp

g++ add.o

a.out

В команде cc или g++ можно с помощью флажка -o указать имя файла, в который необходимо записать полученный в результате компиляции и компоновки загрузочный модуль программы, например

сс -o prog hello.c

prog

Интерпретатор команд позволяет записывать в одной строке несколько команд, например

сс hello.c; a.out

В этом примере исполнимый файл a.out будет запускаться даже в том случае, если компиляция завершится неудачно и файл a.out не будет создан либо запустится загрузочный модуль, полученный когда-то ранее. Для предотвращения запуска следующей команды при неуспешном завершении предыдущей эти команды должны быть соединены связкой &&, например

сс hello.c&&./a.out

Файл a.out автоматически создается компилятором как исполнимый (с флажком *x*). Если же мы создаем командную процедуру (скрипт) обычным текстовым редактором, то она не будет иметь флажка *x* и для ее запуска необходимо использовать команду sh. Например, команда sh comf1 запускает выполнение скрипта comf1.

Аналогичный результат будет получен, если скрипт comf1 сделать исполнимым с помощью команды chmod, а затем запустить его выполнение без использования команды sh, например

chmod +x comf1

./comf1

Если comf1 будет содержать команды

file=hello.c

file1=""

str="Begin compile"

echo file;echo $file;echo $str

то при его выполнении будет получен следующий вывод в консольное окно:

file

hello.c

Begin compile

Внутри скрипта к фактическим параметрам обращаются, используя формальные позиционные параметры $1, $2, … , $9, а также специальные переменные $0, $#, $@, $\*. Переменная $0 содержит имя выполняемой shell-программы (скрипта), $# - число аргументов в команде, а $@ и $\* - список этих аргументов.

В качестве примера рассмотрим командную процедуру comf2 для трансляции и выполнения программ на языке С:

echo "Command = " $0

echo "List of arguments = " $@

echo "Number of arguments = " $#

for i in $@

do

if cc $i

then ./a.out

fi

done

В скрипте comf2 используется условный оператор if и оператор цикла с перечислением for. Командной переменной i присваиваются поочередно имена из списка аргументов - имена файлов с исходными текстами программ. Если трансляция и компоновка завершаются успешно (команда cc дает нулевое значение), полученный загрузочный модуль считывается из файла a.out и выполняется. Например:

$ sh comf2 hello.c add.c

Command = comf2

List of arguments = hello.c add.c

Number of arguments = 2

cc hello.c

./a.out

Hello, world!

cc add.c

./a.out

Input 2 numburs: 2 2

Summa = 4

Отметим, что вместо

for i in $@

в скриптах обычно пишут

for i

поскольку в обоих случаях i поочередно принимает значения позиционных параметров.

Скрипт comf3 в отличие от comf2 компилирует программы, написанные как на С, так и на С++:

for i

do

case $i in

\*.c)echo $i;cc $i;;

\*.cpp)echo $i;g++ $i;;

\*)echo "$i - bad file name";;

esac

a.out

done

Оператор case в зависимости от типа файла в списке аргументов выполняет команду cc или g++.

При компиляции программ (посредством команд cc или g++ с флаж­ком -c) в текущем каталоге появляются файлы объектных модулей. Некоторые компиляторы не создают новые объектные модули, если не удалены старые одноименные объектные модули. Для их удаления можно использовать командную процедуру comf4:

set - \*

for i

do

if test -f $i

then case $i in

\*.o)rm $i

echo "Remove file $i" ;;

esac

fi

done

Встроенная команда set присваивает позиционным параметрам $1, $2, … скрипта имена файлов текущего каталога. В каждом цикле командой test проверяется, является ли файл $i обычным файлом (не каталогом). Если команда test дает нулевое значение, то оператор case проверяет, содержит ли файл объектный модуль (тип файла - .o). Обнаруженный файл с объектным модулем уничтожается, а его имя выдается в консольное окно.

Значения командных переменных можно определить не только оператором присваивания, но и оператором чтения с клавиатуры read. При выполнении следующего скрипта (командной процедуры comf5) в каждом цикле выполняется программа, имя которой введено пользователем по команде read, или программа, находящаяся в файле a.out, если введено пустое имя (сразу введен символ <Enter>):

str2="Execute program and/or <Enter>?"

file1=""

while echo $str2; read file1 || exit

do

if [ "$file1" = "" ]

then echo a.out; a.out

else $file1

fi

done

Команда exit, используемая для завершения скрипта и соответственно цикла while, выполняется, если команда read дает ненулевой код завершения. Обычно команда read дает ненулевой код завершения, если введен символ конца файла <Ctrl-z>. Команда test (вместо слова test здесь используются квадратные скобки) проверяет наличие имени (цепочки символов) в переменной file1.

На рис. 1 приведена командная процедура clg для компиляции, компоновки и выполнения программ, написанных на языках С и С++.

menu="input kind of work:

1 - compile and link

2 - execute

3 - exit"

file1=""

str1="name of source file? "

str2="Execute program and/or Enter? "

while echo $menu; read rab || exit

do

case $rab in

3)exit ;;

1)echo $str1; read file || exit

echo $str2; read file1 || exit

test "$file1" || if test -f a.out ; then rm a.out ; fi

if test -f $file

then

case $file in

\*.c)if test -n "$file1"

then cc -o $file1 $file

else cc $file

fi;;

\*.cpp) if test -n "$file1"

then g++ -o $file1 $file

else g++ $file

fi ;;

\*)echo "bad name of source file " ;;

esac

else echo " source file is absend"

fi ;;

2)if [ $file1 ]

then file1=""

fi

echo $str2; read file1 || exit

if [ $file1 ]

then $file1

else

echo a.out; a.out

fi

;;

\*)echo "kind of work is bad";;

esac

done

Рис. 1. Командная процедура clg (compile/link/go)

В команде sh имеется несколько флагов, облегчающих отладку командных прцедур. Например, флаг -v задает вывод в консольное окно скрипта по мере его считывания из файла и выполнения синтаксического контроля команд и операторов.

При запуске скрипта с флагом -x, например

$ sh -x comf1

включается трассировка выполнения. Команды будут печататься по мере выполнения после подстановки в них значений переменных.

При помощи флага -u можно выявить случаи использования неинициализированных (пустых) командных переменных. Если такой флаг установлен, любая попытка подстановки переменной, которой не было присвоено значение, приведет к выдаче сообщения об ошибке и прекращению выполнения скрипта.

* для раздела с пользовательскими файлами (/home) память выделяется в соответствии с предполагаемыми объемами обрабатываемых данных;
* под область свопинга (swap) выделяется примерно 1 – 2 Гбайт (рекомендуется объем, равный двум объемам оперативной памяти гостевой машины).

В процессе установки пользователь вводит запрашиваемые данные. Поскольку современные компьютеры имеют большую оперативную память (8 и более Гбайт) и высокую производительность, установка системы Linux производится обычно в контейнер виртуальной машины. Предполагается использование эмулятора виртуальной машины VirtualBox, которая является бесплатной и распространяется по лицензии GPL. Отметим, что создаваемая виртуальная машина (ВМ) будет работать под управлением гостевой операционной системы (системы Linux), а основная операционная система (например, Windows), внутри которой создается ВМ, называется хостом.

Другими словами, операционная система хоста (ОС хоста, host OS) – это операционная система физического компьютера, в которой установлена программа для виртуализации (программа VirtualBox). Виртуальная машина (ВМ, VM) – это специальная среда, которую программа виртуализации создает для работы гостевой операционной системы. Таким образом, мы запускаем гостевую операционную систему на одной из созданных виртуальных машин.

Выбор дистрибутива VirtualBox производится в зависимости от операционной системы хоста и разрядности компьютера. 32-х разрядные версии VirtualBox уже не выпускаются. Версия 5.2.26 является последней из 32-х разрядных версий VirtualBox, она будет поддерживаться до июля 2021 года. Все последующие версии VirtualBox будут только 64-х разрядными. Текущая версия – 6.1.

Перед тем как выполнить установку VirtualBox (она выполняется с правами администратора), необходимо убедиться, что возможности виртуализация включены на вашем компьютере. Иначе, даже если VirtualBox вообще удастся запустить, его функции и возможности могут быть серьёзно ограничены. Например, если VirtualBox был установлен на 64-битной системе, он сможет поддерживать в качестве гостевых лишь 32-битные операционные системы. Поскольку виртуализация обычно отключена по умолчанию, нужно будет зайти в настройки BIOS компьютера, чтобы активировать её.

Ниже приведён пример включения виртуализации на ноутбуке HP с процессором Intel (для вашего компьютера процесс может немного отличаться):

1. Перезагрузите компьютер.

2. В процессе перезагрузке нажмите клавишу F10, чтобы войти в BIOS (в других компьютерах обычно нажимают клавишу Del или F2).

3. Используйте клавиши со стрелками, чтобы добраться до вкладки **System Configuration.**

4. Используйте клавиши со стрелками для выбора опции **Virtualisation Technology**, затем нажмите <Enter>.

5. Выделите опцию **Enabled** и нажмите <Enter>.

6. С помощью клавиш со стрелками перейдите на вкладку **Exit.**

7. Убедитесь, что выбрана опция **Exit Saving Changes** и нажмите <Enter>.

Отметим также, что если хост-системой является MS Windows, необходимо, чтобы в ее установленных компонентах отсутствовал Гипервизор.

Для установки VirtualBox необходимо запустить соответствующий установщик, например, VirtualBox-6.1.0-135406-Win.exe. Далее, выполняется несколько шагов по умолчанию, а в конце необходимо отказатся от создания на данном этапе виртуальной машины и нажать кнопку **Далее**.

Несмотря на то, что базовой установки VirtualBox может быть достаточно для некоторых пользователей, рекомендуется также загрузить и установить необязательный пакет расширений (**Extension Pack**) для него. Он добавит некоторые дополнительные функции, включая возможность удаленного доступа к ВМ, использования в виртуальной машине USB-устройств, таких как флешки и принтеры. Установка пакета расширений выполняется следующим образом:

1. После завершения установки VirtualBox запустите его с правами администратора.

2. Выберите в верхнем меню **Файл**, затем **Настройки.**

3. Перейдите на вкладку **Плагины** и щёлкните на значке **Добавить новый плагин**, расположенном справа.

4. Найдите загруженный Extension Pack, точно соответствующий установленной VirtualBox, например, Oracle\_VM\_VirtualBox\_Extension\_Pack-6.1.0.vbox-extpack, убедитесь, что он выделен, и нажмите кнопку **Открыть**, чтобы начать процесс установки.

5. Подтвердите, что вы желаете установить Extension Pack, затем подтвердите своё согласие с условиями лицензии (нужно хотя бы прокрутить текст лицензии вниз, чтобы активировать кнопку **Я согласен**).

6. Extension Pack должен установиться в течение нескольких секунд, после чего появится подтверждающее сообщение.

Итак среда виртуализации создана. Далее мы можем создать одну или несколько виртуальных машин. Создать виртуальную машину можно двумя способами — с использованием установочного диска (или его образа) и с использованием виртуального диска, содержащего ранее созданную ВМ.

При создании новой виртуальной машины необходимо запустить менеджер виртуальных машин и нажать кнопку **Создать**. При создании ВМ необходимо выделить оперативную память для гостевой ОС. Обычно рекомендуется отдать примерно половину объема оперативной памяти хоста, но на практике часто хватает 1 Гб. Однако некоторые современные дистрибутивы требуют не менее 2 - 4 Гб. Следующий шаг – это выбор файла образа жесткого диска. Выбираем опцию **Создать новый виртуальный жесткий диск**, а затем выбираем формат виртуального диска. Обычно используется формат VDI (VirtualBox Disk Image). Далее задается имя жесткого диска ВМ и метод резервирования места на диске хоста. Можно выбрать опцию **Динамический виртуальный жесткий диск** – в этом случае размер файла, эмулирующего жесткий диск ВМ, будет увеличиваться по мере необходимости, но не превысит заданного ранее размера.

После создания ВМ необходимо нажать кнопку **Настроить** для корректировки параметров ВМ. Появится список разделов для возможной настройки. Например, раздел **Носители** позволяет изменить образы загружаемых ОС. Поскольку по умолчанию предполагается использование образа ОС на CD/DVD, а обычно установочный образ диска гостевой ОС мы загружаем из Интернета и записываем в виде ISO-файла на винчестер, то необходимо выполнить следующее. Щелкаем по диску **Пусто** в дереве носителей и нажимем кнопку выбора диска, а из открывшегося меню выбираем команду **Выбрать образ оптического диска**. Откроется диалоговое окно выбора файлов, в котором нужно выбрать записанный на винчестер образ диска. В разделе **Сеть** Вы можете определить, как гостевая ОС будет взаимодействовать по сети с хостом.

Гостевую систему Linux можно создавать либо из дистрибутива, либо на основе виртуального диска, содержащего ранее созданную ОС.

## Лекция №3. Доступ к файловой системе через интерфейс системных вызовов

**Системные и библиотечные функции**

В среде программирования Linux существует два основных интерфейса для файлового ввода-вывода:

* интерфейс системных вызовов, предлагающий системные функции низкого уровня, непосредственно взаимодействующие с ядром операционной системы;
* стандартная библиотека ввода-вывода языка Си, предлагающая функции буферизованного ввода-вывода.

Второй интерфейс является «надстройкой» над интерфейсом системных вызовов, предлагающий более удобный способ работы с файлами. В стандартной библиотеке языка Си определены стандартные потоки ввода-вывода (stdin, stdout и stderr), которые используются функциями scanf(), printf(), getc(), putc(), gets(), puts(), fread(), fwrite(), fopen(), fclose(), fflush(), fseek(), fileno(). Согласно идеологии UNIX, стандартные потоки можно перенаправлять. Это позволяет образовывать цепочки программ, связанных посредством программных каналов.

Все три стандартных потока доступны и низкоуровневым функциям ввода-вывода (read(), write(), creat(), open(), close(), dup(), dup2(), lseek(), readv(), writev(), pipe(), fcntl()), входящим в интерфейс системных вызовов. В системе Linux этот интерфейс включает в себя около 200 системных вызовов.

Стандартная библиотека ввода-вывода языка Си реализует потоки посредством структуры данных FILE и файловых указателей. В системных функциях низкого уровня используется другой способ работы с потоками, основанный на доступе к файлам через файловые дескрипторы.

Библиотечные функции и системные вызовы также различаются по способу передачи процессу информации об ошибке, произошедшей во время выполнения. Так, в случае возникновения ошибки системные вызовы возвращают –1 и устанавливают значение глобальной переменной errno, указывающее причину возникновения ошибки. Файл заголовков <errno.h> содержит коды ошибок, значения которых может принимать переменная errno, с краткими пояснениями.

Следует иметь в виду, что значение errno не обнуляется следующим нормально завершившимся системным вызовом. Следовательно, errno хранит информацию о последней ошибке, которая возникла при выполнении программы.

**Проверка прав доступа к файлу**

В качестве примера рассмотрим идентификацию ошибок, возникающих при выполнении системного вызова access(). Эта функция определяет, имеет ли вызывающий ее процесс право доступа к заданному файлу. Функция способна проверить любую комбинацию прав доступа по чтению, записи и выполнению, а также факт существования файла.

Функция accsess() принимает два аргумента: путь к проверяемому файлу и битовое объединение флагов R\_OK, W\_OK и X\_OK, соответствующих правам чтения, записи и выполнения. При наличии у процесса проверяемой комбинации прав функция возвращает 0. Если файл существует, а нужные привилегии на доступ к нему у процесса отсутствуют, возвращается –1 и в переменную errno записывается код ошибки EACCSES (или EROFS, если проверяется право записи в файл, который расположен в файловой системе, смонтированной только для чтения).

Если второй аргумент равен F\_OK, функция accsess() проверяет лишь факт существования файла. Если файла нет, в переменную errno помещается код ошибки ENOENT). Когда один из каталогов на пути к файлу недоступен, в переменную errno будет помещен код EACCSES.

Программа, показанная в листинге 1, с помощью функции accsess() проверяет существование файла и определяет, разрешен ли к нему доступ на чтение/запись. Имя файла задается в командной строке.

**Листинг 1. (*check-access.c*) Проверка прав доступа к файлу**

#include <errno.h>

#include <stdio.h>

#include <unistd.h>

int main (int argc, char\* argv[])

{

char\* path = argv[1];

int rval;

/\* Проверка существования файла. \*/

rval = access (path, F\_OK);

if (rval == 0)

printf ("%s exists\n", path);

else {

if (errno == ENOENT)

printf ("%s does not exist\n", path);

else if (errno == EACCES)

printf ("%s is not accessible\n", path);

return 0;

}

/\* Проверка права доступа. \*/

rval = access (path, R\_OK);

if (rval == 0)

printf ("%s is readable\n", path);

else

printf ("%s is not readable (access denied)\n", path);

/\* Проверка права записи. \*/

rval = access (path, W\_OK);

if (rval == 0)

printf ("%s is writable\n", path);

else if (errno == EACCES)

printf ("%s is not writable (access denied)\n", path);

else if (errno == EROFS)

printf ("%s is not writable (read-only filesystem)\n", path);

return 0;

}

В Linux имеется удобная библиотечная функция strerror(), возвращающая строковый эквивалент кода ошибки. Эти строки можно включать в формируемые программой сообщения об ошибках. Объявление функции находится в файле <string.h>.

Есть также функция perror() (объявлена в файле <stdio.h>), записывающая сообщение об ошибке непосредственно в поток stderr. Перед собственно сообщением следует размещать строковый префикс, содержащий имя функции или модуля, ставших причиной ошибки.

**Открытие или создание файла**

Программа, представленная в листинге 2, создает файл, имя которого задано в командной строке. В случае ошибки функция perror() выводит сообщение, которому будет предшествовать слово «open».

**Листинг 2. (*create-file.c*) Создание файла**

#include <fcntl.h>

#include <stdio.h>

#include <sys/stat.h>

#include <sys/types.h>

#include <unistd.h>

int main (int argc, char\* argv[])

{

/\* Путевое имя нового файла. \*/

char\* path = argv[1];

/\* Права доступа к файлу. \*/

mode\_t mode = S\_IRUSR | S\_IWUSR | S\_IRGRP | S\_IWGRP | S\_IROTH;

/\* Создание файла. \*/

int fd = open (path, O\_WRONLY | O\_EXCL | O\_CREAT, mode);

if (fd == -1) {

/\* Произошла ошибка. Выводим сообщение и завершаем работу. \*/

perror ("open");

return 1;

}

return 0;

}

В программе create-file вместо системного вызова creat() используется системный вызов open(). В системе Linux создание файла рассматривается как частный случай открытия файла. Вызов creat() эквивалентен вызову

open(path, O\_WRONLY | O\_CREAT | O\_TRUNC, mode);

если при создании файла разрешается удалять (замещать) существующий одноименный файл, или вызову

open(path, O\_WRONLY | O\_CREAT | O\_EXCL, mode);

если такая возможность запрещена.

Второй аргумент функции open() содержит битовое объединение флажков, определяющих способ открытия файла. Флажки O\_RDONLY и O\_WRONLY открывают файл только для чтения или только для записи. В случае флага O\_RDWR файл открывается и для чтения, и для записи.

Отметим, что не всякий файл можно открыть в любом из трех режимов. Например, CD-ROM нельзя открыть для записи. Обычно существующие права доступа к файлу могут не позволить конкретному процессу открывать файл для чтения или записи. Эти права доступа при создании файла передаются в функцию open() в качестве третьего аргумента.

В программе create-file устанавливаются права доступа к файлов посредством указания битового объединения флажков. Флажки S\_IRUSR и S\_IWUSR разрешают пользователю-владельцу операции чтения/записи. Аналогично, флажки S\_IRGRP и S\_IWGRP разрешают операции чтения/записи группе-владельцу файла. Для остальных категорий пользователей флажок S\_IROTH разрешает только чтение.

При создании файла с помощью функции open() некоторые из указываемых в аргументе mode битов режима могут отключаться. Это значит, что существующая для данного процесса маска прав доступа вводит свои ограничения. Например, функция

umask (S\_IWGRP | S\_IRWXO);

или соответствующая ей команда

$ umask 027

означают, что право записи для группы, а также права чтения, записи и выполнения для остальных пользователей будут всегда отниматься от указываемых в аргументе mode прав доступа. Для проверки текущей маски необходимо выполнить команду umask без аргументов.

**Текстовые файлы в Linux и в DOS/Windows**

Системная функция read() осуществляет чтение данных из файла. Она принимает дескриптор файла, указатель на буфер и счетчик числа считываемых данных. Функция возвращает число прочитанных байтов или –1 в случае ошибки. Иногда читается меньше байтов, чем требуется, например, если файл содержит меньший размер.

В Linux-программах нередко приходится читать файлы, созданные в DOS или Windows. Важно понимать разницу между тем, как структурируются текстовые файлы в Linux и в DOS/Windows.

В Linux каждая строка текстового файла оканчивается символом новой строки. Он представляется символьной константой ‘\n’, ASCII-код которой равен 10. В Windows строки разделяются двухсимвольной комбинацией: символ возврата каретки (константа ‘\r’, ASCII-код 13), за которым идет символ новой строки.

Некоторые текстовые редакторы Linux при отображении текстовых файлов Windows ставят в конце каждой строки обозначение ^M – символ возврата каретки. В редакторе Emacs такие файлы отображаются правильно, но в строке режима вместо прав доступа появляется запись (DOS). Многие Windows-редакторы, например, Notepad (Блокнот), показывают содержимое текстового файла Linux в виде одной длинной строки, поскольку для них строка всегда заканчивается символом возврата каретки.

Если программа читает текстовые файлы, сгенерированные Windows-программами, желательно заменять последовательность ‘\r\n’ на символ новой строки. Точно также при записи в Linux текстовых файлов, которые будут читаться Windows-программами, нужно менять одиночные символы новой строки на комбинации ‘\r\n’.

Программа, представленная в листинге 3, позволяет пользователю визуально определять операционную систему, в которой был создан файл. Программа отображает шестнадцатиричный дамп файла, заданного в командной строке. В каждой строке дампа выводится смещение от начала файла, а затем очередные 16 байтов.

**Листинг 3. (*hexdump.c*) Отображение шестнадцатиричного дампа файла**

#include <fcntl.h>

#include <stdio.h>

#include <sys/stat.h>

#include <sys/types.h>

#include <unistd.h>

int main (int argc, char\* argv[])

{

unsigned char buffer[16];

size\_t offset = 0;

size\_t bytes\_read;

int i;

/\* Открытие файла для чтения. \*/

int fd = open (argv[1], O\_RDONLY);

/\* Чтение данных из файла по одному блоку за раз. Чтение

продолжается до тех пор, пока размнр очередной порции

байтов не окажется меньше размера буфера. Это свидетель-

ствует о достижении конца буфера. \*/

do {

/\* Чтение следующей строки байтов. \*/

bytes\_read = read (fd, buffer, sizeof (buffer));

/\* Отображение смещения, а затем самих байтов. \*/

printf ("0x%06x : ", offset);

for (i = 0; i < bytes\_read; ++i)

printf ("%02x ", buffer[i]);

printf ("\n");

/\* Вычисление позиции в файле. \*/

offset += bytes\_read;

}

while (bytes\_read == sizeof (buffer));

/\* Конец работы. \*/

close (fd);

return 0;

}

**Метаданные файла**

При чтении файла иногда целесообразно использовать буфер, не меньший, чем размер файла. А для этого необходимо обратиться к метаданным файла, находящимся в индексном дескрипторе файла, и узнать размер файла.

Системная функция stat() позволяет получить сведения о файле, которые нам обычно предоставляет интерпретатор shell по команде *ls –l*. Этой функции необходимо передать путевое имя файла и указатель на структуру типа stat. В случае успешного завершения функция возвращает 0 и заполняет поля структуры данными о файле.

Перечислим наиболее полезные поля структуры stat:

* в поле st\_mode содержится код доступа к файлу. В старшем бите поля закодирован тип файла;
* в полях st\_uid и st\_gid содержатся идентификаторы пользователя и группы, которым принадлежит файл;
* в поле st\_size хранится размер файла в байтах;
* в поле st\_atime записано время последнего обращения к файлу (для чтения или записи);
* в поле st\_mtime записано время последней модификации файла;
* в поле st\_info содержится номер индексного дескриптора файла, определяющий порядковый номер файла в файловой системе.

Для проверки поля st\_mode используются макросы. Они, например, проверяют тип файла: блочное (S\_ISBLK) или символьное (S\_ISCHR) устройство, каталог (S\_ISDIR), FIFO-файл (S\_ISFIFO), символическая ссылка (S\_ISLNK), обычный файл (S\_ISREG) или сокет (S\_ISSOCK). Макросы возвращают ненулевое значения, если проверка подтвердила проверяемое предположение.

Если вызвать функцию stat() для символической ссылки, функция проследит, куда указывает ссылка, и вернет информацию о файле, а не о самой ссылке. Таким образом, в случае функции stat() макрос S\_ISLNK() всегда будет возвращать значение 0. Есть другая функция, lstat(), которая не пытается отслеживать символические ссылки. Во всем остальном она эквивалентна функции stat(). Отметим, что если вызвать функцию stat() для поврежденной ссылки (которая указывает на несуществующий или недоступный файл), возникнет ошибка, тогда как функция lstat() в подобной ситуации выполнится успешно.

Если файл открыт для чтения или записи, доступ к его метаданным лучше выполнять функцией fstat(). В качестве первого аргумента она принимает не путевое имя, а дескриптор файла.

В листинге 4 показана программа, которая создает буфер достаточного размера и загружает в него содержимое указанного файла. Размер файла определяется с помощью функции fstat(). Она же позволяет проверить, соответствует ли заданное имя обычному файлу.

**Листинг 4. (*read-file.c*) Загрузка файла в буфер**

#include <fcntl.h>

#include <stdio.h>

#include <stdlib.h>

#include <sys/stat.h>

#include <sys/types.h>

#include <unistd.h>

int main (int argc, char\* argv[])

{

int fd;

struct stat file\_info;

char\* buffer;

size\_t\* length;

/\* Открытие файла. \*/

fd = open (argv[1], O\_RDONLY);

/\* Получение информации о файле. \*/

fstat (fd, &file\_info);

\*length = file\_info.st\_size;

/\* Проверка того, что это обычный файл. \*/

if (!S\_ISREG (file\_info.st\_mode)) {

/\* Этот тип файла не поддерживается. \*/

close (fd);

return 0;

}

/\* Выделение буфера достаточного размера. \*/

buffer = (char\*) malloc (\*length);

/\* Загрузка файла в буфер. \*/

read (fd, buffer, \*length);

/\* Конец работы. \*/

close (fd);

return 0;

}

**Копирование файлов**

Системная функция sendfile() реализует механизм копирования данных из одного файла, представленного файловым дескриптором, в другой. Дескрипторы могут соответствовать дисковым файлам, сокетам или устройствам.

Функция sendfile() не требует создавать промежуточный буфер для копирования файлов. Ей передаются дескриптор для записи, дескриптор для чтения, указатель на переменную смещения и число копируемых байт. Переменная смещения определяет позицию входного файла, с которой начинается копирование (0 – это начало файла). После окончания копирования переменная будет содержать смещение конца блока.

Программа, представленная в листинге 5, представляет собой простую, но очень эффективную реализацию механизма файлового копирования. Она принимает в командной строке два имени файла и копирует содержимое первого файла во второй. Размер исходного файла определяется с помощью функции fstat().

**Листинг 5. (*copy.c*) Копирование файла с помощью функции sendfile()**

#include <fcntl.h>

#include <stdlib.h>

#include <stdio.h>

#include <sys/sendfile.h>

#include <sys/stat.h>

#include <sys/types.h>

#include <unistd.h>

int main (int argc, char\* argv[])

{

int read\_fd, write\_fd;

struct stat stat\_buf;

off\_t offset = 0;

/\* Открытие входного файла. \*/

read\_fd = open (argv[1], O\_RDONLY);

/\* Определение размера входного файла. \*/

fstat (read\_fd, &stat\_buf);

/\* Открытие выходного файла для записи. \*/

write\_fd = open (argv[2], O\_WRONLY | O\_CREAT, stat\_buf.st\_mode);

/\* Передача данных из одного файла в другой. \*/

sendfile (write\_fd, read\_fd, &offset, stat\_buf.st\_size);

/\* Закрытие файлов. \*/

close (read\_fd);

close (write\_fd);

return 0; }

**Работа с символическими ссылками**

При работе с символическими ссылками (связями) используется системная функция readlink(), которая определяет адресата символической ссылки. Функция принимает три аргумента: путь к символической ссылке, буфер для записи адресата и длину буфера. Отметим, что путевое имя, помещаемое в буфер, не завершается нулевым символом. Но поскольку в третьем аргументе возвращается длина строки-путевого имени, добавить этот символ несложно.

Если первый аргумент функции readlink() не является символической ссылкой, функция возвращает –1, а в переменную errno записывается константа EINVAL.

Программа, приведенная в листинге 6, выводит на экран адресата символической ссылки, заданной в командной строке.

**Листинг 6. (*print-symlink.c*) Отображение адресата символической ссылки**

#include <errno.h>

#include <stdio.h>

#include <unistd.h>

int main (int argc, char\* argv[])

{

char target\_path[256];

char\* link\_path = argv[1];

/\* Попытка чтения адресата символической ссылки. \*/

int len = readlink (link\_path, target\_path, sizeof (target\_path) - 1);

if (len == -1) {

/\* Функция завершилась ошибкой. \*/

if (errno == EINVAL)

/\* Это не символическая ссылка. \*/

fprintf (stderr, "%s is not a symbolic link\n", link\_path);

else

/\* Произошла какая-то другая ошибка. \*/

perror ("readlink");

return 1;

}

else {

/\* Завершаем путевое имя нулевым символом. \*/

target\_path[len] = '\0';

/\* Выводим результат. \*/

printf ("%s\n", target\_path);

return 0;

}

}

Для записи данных в файл предназначена системная функция write(). Она принимает дескриптор файла, указатель на буфер данных и число записываемых байтов. Файл должен быть открыт для записи. Функция write() работает не только с текстовыми, но и с произвольными байтами.

В Linux, как и в большинстве других операционных систем, при записи в файл данные не передаются на диск немедленно. Вместо этого операционная система помещает их в резидентный кэш-буфер с целью сокращения числа обращений к диску и повышения оперативности программы. Когда буфер заполнится или произойдет какое-нибудь другое событие (например, истечет определенный промежуток времени), система запишет содержимое буфера на диск в ходе одной непрерывной операции.

Данный тип кэширования делает ненадежными программы, зависящие от целостности дисковых данных. Если система внезапно выйдет из строя, например, вследствие сбоя ядра или отключения питания, любые данные, находящиеся в памяти и еще не записанные на диск, будут потеряны.

Поэтому такие программы могут сами управлять процессом записи данных на диск. Эту возможность предоставляет системная функция fsync(). Она принимает один аргумент – дескриптор записываемого файла – и принудительно переносит на диск все данные этого файла, находящиеся в кэш-буфере. Функция не завершается до тех пор, пока данные не окажутся на диске.

Программа, показанная на листинге 7, записывает вводимую в команде строку сразу на диск, используя функцию fsync().

**Листинг 7. (*write\_journal\_entry.c*) Использование синхронизации при записи в файл**

#include <fcntl.h>

#include <string.h>

#include <sys/stat.h>

#include <sys/types.h>

#include <unistd.h>

const char\* journal\_filename = "journal.log";

int main (int argc, char\* argv[])

{

int fd = open (journal\_filename, O\_WRONLY | O\_CREAT | O\_APPEND, 0660);

write (fd, argv[1], strlen (argv[1]));

write (fd, "\n", 1);

fsync (fd);

close (fd);

return 0;

}

## Лекция №4. Работа с процессами в командной и программной средах операционной системы Linix

**Мониторинг процессов**

Выполняющийся экземпляр программы называется процессом. Каждый процесс в Linux помечается уникальным идентификатором PID (process identifier). Идентификаторы – это 16-разрядные числа, назначаемые последовательно по мере создания процессов.

У всякого процесса имеется также родительский процесс, а идентификатор родительского процесса PPID (parent PID) относится к атрибутам процесса. Таким образом, все процессы Linux организованы в виде древовидной иерархии, на вершине которой находится процесс-демон init.

Работая с переменными-идентификаторами процессов в программах, написанных на языках С и С++, следует объявлять эти переменные как имеющие тип pid\_t (определен в файле <sys/types.h>). Программа может узнать идентификатор своего собственного процесса с помощью системного вызова getpid(), а идентификатор родительского процесса – с помощью вызова getppid(). В листинге 1 показано, как это сделать.

**Листинг 1. (*print-pid.c*) Вывод идентификатора процесса**

#include <stdio.h>

#include <unistd.h>

int main ()

{

printf ("The process id is %d\n", (int) getpid ());

printf ("The parent process id is %d\n", (int) getppid ());

return 0;

}

Обратите внимание, что при повторных запусках эта программа выдает разные значения pid, поскольку всякий раз запускается новый процесс. Однако родительский идентификатор у них оказывается одинаковым.

Команда ps отображает список процессов, работающих в данный момент в системе. С помощью опций можно указывать, о каких процессах и какую именно требуется получить информацию. GNU-версия этой команды, входящая в систему Linux, поддерживает опции в стиле трех разных типов UNIX. Опции в стиле Unix98 состоят из одного или нескольких символов, перед которым(и) должен стоять дефис. Опции в стиле BSD используются без дефиса. Опции, характерные только для GNU-версии, представляют собой слово, перед которым должно стоять два дефиса.

Будучи вызванной без аргументов, команда ps выводит список тех процессов, управляющим терминалом которых является ее собственный терминал. Более полный список можно получить с помощью команд

ps –e –o pid,ppid,command|more

ps –o pid,user,start\_time,command

Здесь опция -e означает вывод информации обо всех процессах в системе, иначе выводится информация о процессах, подключенных к данному терминалу. Опция -o указывает на атрибуты процессов, которые должны выводиться.

Полный список опций можно узнать на man-странице команды ps. Отметим, что имеются три предопределенных формата вывода: -f (полный листинг), -l (длинный листинг) и –j (вывод заданий). В зависимости от опций в выводе команды ps могут присутствовать следующие поля: имя владельца процесса (USER), идентификаторы процессов (PID и PPID), доля времени центрального процессора и доля реальной памяти (в процентах), использованные данным процессом (CPU и MEM), виртуальный размер процесса в килобайтах (VSZ), размер резидентного набора – количество 1К-страниц в памяти (RSS), время старта процесса (STIME), управляющий терминал процесса (TTY), статус процесса (S или STAT), приоритет планирования процесса (PRI) и значение nice – приращение приоритета (NI), сколько времени центрального процессора занял данный процесс (TIME), командная строка запуска программы, выполняемой данным процессом (CMD или COMMAND).

В поле статуса процесса могут стоять следующие значения:

* R – выполнимый процесс, ожидающий момента, когда планировщик задач выделит ему очередной квант времени;
* S – процесс "спит";
* D – процесс находится в состоянии подкачки на диске;
* T – остановленный процесс;
* Z – процесс-зомби.

Рядом с указателем статуса могут стоять дополнительные символы из следующего набора:

* + W – процесс не имеет резидентных страниц;
  + < - высокоприоритетный процесс;
  + N – низкоприоритетный процесс;
  + L – процесс имеет страницы, заблокированные в памяти.

Для того, чтобы увидеть "лес" деревьев "предок-потомок", можно воспользоваться командой pstree.

Команды ps и pstree позволяют сделать как бы "моментальный снимок" процессов, запущенных в системе. В отличие от них команда top отображает в окне состояние процессов и их активность в реальном режиме времени. Содержимое окна обновляется каждые 5 секунд. Список процессов в окне может быть отсортирован с помощью следующихзы команд программы top:

* N – сортировка по PID;
* A – сортировка процессов по возрасту;
* P – сортировка по использованию ЦП (по умолчанию);
* M – сортировка процессов по использованию памяти;
* T- сортировка по времени выполнения.

Кроме команд, определяющих режим сортировки, утилита top воспринимает еще ряд команд, которые позволяют управлять процессами в интерактивном режиме. С помощью команды-клавиши <K> можно завершить некоторый процесс ( его PID будет запрошен), а с помощью клавиши <R> можно переопределить значение nice для некоторого процесса. Таким образом, эти две команды утилиты top аналогичны командам kill и renice интерпретатора команд shell.

При работе в графической оболочке KDE можно использовать утилиту KDE Task Manager, запускаемую по команде kpm. Эта программа отображает рабочие процессы и строит графики загрузки процессора и использования памяти.

Система X Window содержит еще несколько инструментальных средств, позволяющих следить за использованием ресурсов процессами. Утилита xsysinfo показывает загрузку процессора, объем используемой физической памяти и объем используемой области подкачки. Еще одной утилитой, которая позволяет получить дополнительную информацию, является xosview.

**Создание процессов**

Существует два способа создания процессов. Первый из них относительно прост, но применяется редко, поскольку неэффективен и связан со значительным риском для безопасности системы. Этот способ основан на использовании библиотечной функции system(), которая позволяет выполнить команду интерпретатора shell в программной среде. Например, программа, представленная в листинге 2, выполняет команду ls –l /, отображающую содержимое корневого каталога.

**Листинг 2. (*system.c*) Использование библиотечной функции system()**

#include <stdlib.h>

int main ()

{

int return\_value;

return\_value = system ("ls -l /");

return return\_value;

}

При выполнении функции system() запускается стандартный интерпретатор Bourne shell (/bin/sh) и ему передается команда для исполнения. Если интерпретатор не может быть запущен, функция system() возвращает значение 127, а в случае возникновения других ошибок возвращает значение –1.

Поскольку функция system() запускает интерпретатор команд, могут возникнуть определенные проблемы. Так, в большинстве UNIX-систем программа /bin/sh представляет собой символическую ссылку на другой интерпретатор. В Linux – это bash. Выполнение команды может привести к различным последствиям, которые зависят от многих факторов. Поэтому более предпочтительным является создание процессов на низком уровне программирования - с помощью системных вызовов fork() и exec().

В DOS и Windows API имеется семейство функций spawn(). Они принимают в качестве аргумента имя программы, создают новый экземпляр ее процесса и запускают его. В Linux нет функции, которая делала все это за один заход. Вместо этого имеется функция fork(), создающая дочерний процесс, который является точной копией родительского процесса, и семейство функций exec(), заставляющих созданный процесс перестать быть экземпляром родительской программы и превратиться в экземпляр другой программы.

После создания дочернего процесса и родительский, и дочерний процессы начинают выполнять оператор, следующий за вызовом fork(). Функция fork() возвращает разные значения в родительском и дочернем процессах, что позволяет программам легко определить, кто родитель, а кто потомок.

Программа, представленная в листинге 3, имеет оператор if, в которой первую ветвь выполняет процесс-предок, а вторую – процесс-потомок.

**Листинг 3. (*fork.c*) Использование fork() для создания дочернего процесса**

#include <stdio.h>

#include <sys/types.h>

#include <unistd.h>

int main ()

{

pid\_t child\_pid;

printf("the main program process id is %d\n",(int)getpid ());

child\_pid = fork ();

if (child\_pid != 0) {

printf ("this is the parent process, with id %d\n",

(int) getpid ());

printf ("the child's process id is %d\n", (int) child\_pid);

}

else

printf ("this is the child process, with id %d\n",

(int) getpid ());

return 0;

}

Отметим, что родительский и дочерний процессы являются равноправными в борьбе за системные ресурсы и, в частности, за использование центрального процессора. Программа, представленная в листинге 4, демонстрирует это. В файле TEST мы будем от случая к случаю получать строки вида

aABbCcDdEe или AaBbcdCDEe

что говорит о том, что первым "проснуться" после fork() может любой из двух процессов. Если же опыт дает устойчиво строки, начинающиеся с одной и той же буквы – значит в данной реализации системы один из процессов все же запускается раньше. Но не стоит использовать этот эффект в своих программах - при переносе на другую систему его может не быть.

**Листинг 4. (*parent-child.c*) Работа процессов после завершения вызова fork()**

#include <stdio.h>

int pid, i, fd; char c;

main(){

fd = creat( "TEST", 0644);

if( !(pid = fork())){ /\* дочерний процесс \*/

c = 'a';

for(i=0; i < 5; i++){

write(fd, &c, 1); c++; sleep(1);

}

printf("child %d exit\n", getpid());

exit(0);

}

/\* else родительский процесс \*/

c = 'A';

for(i=0; i < 5; i++){

write(fd, &c, 1); c++; sleep(1);

}

printf("Parent %d process of %d exit\n",

getpid(), pid );

}

В данной программе используется следующее свойство системы UNIX: при системном вызове fork() порожденный процесс получает все открытые порождающим процессом файлы "в наследство" - это соответствует тому, что таблица открытых процессом файлов копируется в процесс-потомок. Именно так, в частности, передаются от родительского процесса дочернему стандартные каналы 0, 1, 2. Порожденному процессу не нужно открывать стандартные ввод, вывод и вывод ошибок явно. Изначально же они открываются специальной программой при входе в систему.

Следующие функции, входящие в семейство exec(), немного отличаются друг от друга по своим возможностям и способу вызова:

* Функции, в названии которых присутствует суффикс 'p' (execvp() и execlp()), принимают в качестве аргумента имя программы и ищут эту программу в каталогах, определяемых переменной среды PATH. Всем остальным функциям нужно передавать полное путевое имя программы.
* Функции, в названии которых присутствует суффикс 'v' (execv(), execvp() и execve()), принимают список аргументов программы, упакованный в массив строковых указателей, который оканчивается NULL-указателем.
* Функции с суффиксом 'l' (execl(), execlp() и execle()) принимают список аргументов, не упакованный в массив.
* Функции, в названии которых присутствует суффикс 'e' (execve() и execle()), в качестве дополнительного аргумента принимают массив переменных среды. Этот массив содержит строковые указатели и оканчивается пустым указателем. Каждая строка должна иметь вид "*ПЕРЕМЕННАЯ*=значение".

Поскольку функция exec() заменяет одну программу другой, она не возвращает значение, если завершается успешно.

Список аргументов, передаваемых программе, аналогичен аргументам командной строки, указываемых при запуске программы в интерактивном режиме. Их тоже можно получить с помощью параметров argc и argv функции main новой программы дочернего процесса.

В листинге 5 показана программа, которая отображает содержимое корневого каталога с помощью команды ls, как и программа в листинге 2. Но на этот раз команда ls запускается не из интерпретатора, а напрямую, и ей передаются аргументы -l и / .

**Листинг 5. (*fork-exec.c*) Совместное использование функций fork() и exec()**

#include <stdio.h>

#include <stdlib.h>

#include <sys/types.h>

#include <unistd.h>

int spawn (char\* program, char\*\* arg\_list)

{

pid\_t child\_pid;

/\* Создание копии текущего процесса. \*/

child\_pid = fork ();

if (child\_pid != 0)

/\* Это родительский процесс. \*/

return child\_pid;

else {

/\* Выполнение указанной программы. \*/

execvp (program, arg\_list);

/\* Функция execvp() возвращает значение только в случае

ошибки. \*/

fprintf (stderr, "an error occurred in execvp\n");

abort ();

}

}

int main ()

{

/\* Список аргументов, передаваемых команде "ls". \*/

char\* arg\_list[] = {

"ls", /\* argv[0] – имя программы. \*/

"-l",

"/",

NULL /\* Список аргументов должен заканчиваться

указателем NULL. \*/

};

spawn ("ls", arg\_list);

printf ("done with main program\n");

return 0;

}

## Лекция №5. Ненадежные и надежные сигналы в системе Unix

**Сигналы**

Процессы могут взаимодействовать между собой посредством сигналов. Пользователи могут управлять работой процессов также с помощью сигналов, которые в этом случае будет посылать процессу ядро системы Linux. Сигналы генерируются и при возникновении ошибок.

Сигнал представляет собой специальное сообщение, посылаемое процессу. Когда процесс принимает сигнал, он немедленно обрабатывает его, прерывая выполнение текущей функции и даже текущей строки (оператора) программы.

Сигналы распознаются по номерам, но в программах для ссылки на сигналы пользуются символическими константами. В Linux эти константы определены в файле /usr/include/bits/signum.h (его не надо включать в программу, для этого есть файл <signal.h>).

В ответ на полученный сигнал процесс выполняет ряд действий в зависимости от типа сигнала. У каждого сигнала есть стандартный обработчик, определяющий, что произойдет с процессом, если он попытается проигнорировать сигнал. Для большинства сигналов можно указать собственную функцию-обработчик сигнала. В этом случае при поступлении сигнала выполнение программы приостанавливается, выполняется обработчик, а потом программа возобновляет свою работу.

Система Linux посылает процессам сигналы в случае возникновения определенных ситуаций. Например, сигналы SIGBUS (ошибка на шине), SIGSEGV (нарушение сегментации) и SIGFPE (ошибка операции с плавающей запятой) могут быть посланы процессу, пытающемуся выполнить неправильную операцию. По умолчанию эти сигналы приводят к завершению процесса и созданию дампа оперативной памяти (файла core).

Процесс может сам послать сигнал другому процессу. Чаще всего возникает необходимость завершить требуемый процесс с помощью сигнала SIGTERM или SIGKILL. Сигнал SIGTERM является запросом на завершение; процесс может его проигнорировать и продолжить свое выполнение. Сигнал SIGKILL вызывает немедленное безусловное уничтожение процесса и не может быть проигнорирован или обработан.

С помощью сигналов можно передавать команды выполняющимся программам. Для этого существуют "пользовательские" сигналы SIGUSR1 и SIGUSR2. Иногда в аналогичных целях применяется сигнал SIGHUP, с помощью которого можно заставить программу повторно прочитать свои файлы конфигурации.

Системный вызов, находящийся в состоянии ожидания какого-либо события (read, ждущий нажатия кнопки на клавиатуре, или wait, ждущий окончания процесса-потомка), может быть прерван сигналом. При этом системный вызов вернет значение "ошибка" (-1) и errno станет равным EINTR. Это позволяет писать системные вызовы с выставлением тайм­а-ута: если событие не происходит в течение заданного времени, то завершить ожидание и прервать системный вызов. Для этой цели используется вызов alarm(sec), заказывающий посылку сигнала SIGALRM программе через целое число sec секунд (sec = 0, если отменяем заказ).

Программа, представленная в листинге 6, показывает использование сигнала SIGALRM для реализации электронных часов.

**Листинг 6. (*clock.c*) Простейшая реализация электронных часов**

#include <signal.h>

#include <time.h>

#include <stdio.h>

#include <string.h>

#include <unistd.h>

void tick(int nsig){

time\_t tim; char \*s;

signal (SIGALRM, tick);

alarm(3);

time(&tim);

s = ctime(&tim);

s[ strlen(s)-1 ] = '\0'; /\* обрубить '\n' \*/

fprintf(stderr, "\r%s", s);

}

int main(){

tick(0);

for(;;)

pause();

return();

}

При получении сигнала процесс может выбрать одно из трех возможных действий: игнорировать сигнал, перехватить и самостоятельно обработать сигнал, вызвать действие по умолчанию. Заданное действие при получении сигнала называется *диспозицией сигнала.*

Простейшим интерфейсом к сигналам UNIX является библиотечная функция signal(), использующая семантику устаревших или ненадежных сигналов. Процесс при этом имеет весьма слабые возможности управления сигналами. Во-первых, процесс не может заблокировать сигнал, т.е. отложить получение сигнала на период выполнения критического участка кода. Во-вторых, каждый раз при получении сигнала его диспозиция устанавливается на действие по умолчание. Поэтому в обработчике требуется восстанавливать диспозицию.

Библиотечная функция signal() имеет следующее определение на языке Си:

#include <signal.h>

void (\*signal (int sig, void (\*disp)(int)))(int);

Аргумент sig определяет сигнал, диспозицию которого нужно изменить. Аргумент disp определяет новую диспозицию сигнала, в качестве которой может быть указана определенная пользователем функция-обработчик или одно из следующих значений:

* SIG\_DFL – вызвать системный обработчик для данного сигнала, т.е. выполнить действие по умолчанию;
* SIG\_IGN – сигнал следует игнорировать.

В случае успешного завершения signal() возвращает предыдущую диспозицию – это может быть функция-обработчик сигнала типа

void (\*oldaction)();

или системные значения SIG\_DFL или SIG\_IGN. Возвращаемое значение может быть использовано для восстановления диспозиции в случае необходимости.

В программе, приведенной в листинге 7, определена диспозиция для трех сигналов: SIGINT, SIGUSR1 и SIGUSR2. При получении сигнала SIGINT вызывается обработчик sig\_hndlr(), при получении сигнала SIGUSR1 производится действие по умолчанию (процесс завершает работу), а сигнал SIGUSR2 игнорируется процессом. После установки диспозиции сигналов процесс запускает бесконечный цикл, в котором вызывается системная функция pause().

**Листинг 7. (*old-signal.c*) Установка и проверка диспозиции сигналов**

#include <signal.h>

/\* Функция-обработчик сигнала \*/

static void sig\_hndlr(int signo)

{

/\* Восстановим диспозицию \*/

signal (SIGINT, sig\_hndlr);

printf("Получен сигнал SIGINT\n");

}

main()

{

/\* Установим диспозицию \*/

signal (SIGINT, sig\_hndlr);

signal (SIGUSR1, SIG\_DFL);

signal (SIGUSR2, SIG\_IGN);

/\* Бесконечный цикл \*/

while(1)

pause();

}

Заметим, что каждый раз при получении сигнала SIGINT мы вынуждены восстанавливать требуемую диспозицию, в противном случае получение следующего сигнала этого типа вызвало бы завершение выполнения процесса (действие по умолчанию).

Запустим программу old-signal сначала как фоновый процесс, который не будет подключен к консоли для ввода-вывода данных. При запуске мы получим номер задания, идентификатор запущенного процесса и промптер - приглашение к вводу следующей команды. С помощью команды kill будем посылать этому процессу сигналы SIGINT, SIGUSR2, SIGUSR1. Последний сигнал вызовет завершение выполнение процесса:

Изображение выглядит как текст, Шрифт, снимок экрана

Автоматически созданное описание

Запустим теперь программу old-signal как интерактивный процесс, который захватит консоль и не позволит вводить команды, пока этот процесс не завершится. Мы можем послать процессу сигнал SIGINT, если нажмем Ctrl+C (на экране отобразится как ^C). Чтобы послать процессу сигнал SIGUSR2, а затем SIGUSR1, надо открыть еще одно консольное окно (окно эмулятора терминала), с помощью команды ps -e определить идентификатор нашего интерактивного процесса, а затем по этому PID послать сигналы:

Изображение выглядит как текст, снимок экрана, программное обеспечение, число

Автоматически созданное описание

**Надежные сигналы**

Стандарт POSIX.1 определил новый набор функций управления сигналами, основанный на понятии набора сигналов. Набор сигналов описывается в программе переменной типа sigset\_t, в которой каждый бит представляет один сигнал.

Вместо библиотечной функции signal() стандарт POSIX.1 определяет системную функцию sigaction(), позволяющую установить диспозицию сигналов и/или узнать ее текущее значение. Программа, представленная в листинге 8, является более современной реализацией программы, представленной в листинге 7.

Обычно процесс завершается одним из двух способов: либо выполняющаяся программа вызывает функцию exit(), либо функция main() заканчивается. Возможно также аварийное завершение процесса - в ответ на получение сигнала. Таковыми могут быть, например, сигналы SIGBUS, SIGSEGV и SIGFPE. Есть сигналы, явно запрашивающие прекращение работы процесса. Например, сигнал SIGINT посылается, когда пользователь нажимает <Ctrl+C>. Сигнал SIGTERM посылается процессу командой kill без аргумента-номера сигнала. Аналогичный сигнал посылается процессу child\_pid при выполнении функции kill(child\_pid, SIGTERM). Если программа вызывает функцию abort(), она посылает сама себе сигнал SIGABRT. Самый "могучий" из всех сигналов – SIGKILL: он приводит к безусловному уничтожению процесса и не может быть ни блокирован, ни обработан.

**Листинг 8. (*new-signal.c*) Использование надежных сигналов**

#include <signal.h>

#include <sys/types.h>

#include <sys/stat.h>

#include <fcntl.h>

#include <unistd.h>

/\* Вариант "надежной" функции signal() \*/

void (\*mysignal (int signo, void (\*hndlr)(int)))(int)

{

struct sigaction act, oact;

/\* Установим маску сигналов \*/

act.sa\_handler = hndlr;

sigemptyset(&act.sa\_mask);

act.sa\_flags = 0;

if(signo != SIGALRM)

act.sa\_flags |= SA\_RESTART;

/\* Установим диспозицию \*/

if(sigaction(signo, &act, &oact) < 0)

return (SIG\_ERR);

return (oact.sa\_handler);

}

/\* Функция-обработчик сигнала \*/

static void sig\_hndlr(int signo)

{

/\* Эта часть кода нам уже не нужна:

mysignal (SIGINT, sig\_hndlr); \*/

printf("Получен сигнал SIGINT\n");

}

main()

{

/\* Установим диспозицию \*/

mysignal (SIGINT, sig\_hndlr);

mysignal (SIGUSR1, SIG\_DFL);

mysignal (SIGUSR2, SIG\_IGN);

/\* Бесконечный цикл \*/

while(1)

pause();

}

Здесь функция mysignal() инициализирует поля структуры данных act типа sigaction и вызывает системную функцию sigaction(). Отметим, что функция sigemptyset() очищает набор сигналов в поле sa\_mask структуры типа sigaction.

**Завершение процессов**

Обычно нулевой код завершения процесса говорит об успешном завершении процесса. Необходимо знать правила формирования ненулевых кодов завершения. Несмотря на то, что тип параметра функции exit(), как и тип возвращаемого значения функции main(), равен int, система Linux записывает код завершения лишь в младший из четырех байтов. Если процесс завершается без получения сигнала, его код завершения находится в диапазоне от 0 до 127. Иначе код завершения равен 128 плюс номер сигнала.

После выполнения функции fork() родительский и дочерний процессы работают параллельно. В программах, приведенных в листингах 3 и 4, родительский процесс завершался, не дожидаясь завершения порожденного им процесса. Однако в системных программах родительскому процессу часто необходимо знать, как завершился дочерний процесс. Это можно сделать с помощью функций семейства wait().

Например, в функцию main() файла fork-exec.c можно добавить следующий фрагмент программы:

/\*Wait for the child process to complete.\*/

wait (&child\_status);

if (WIFEXITED(child\_status))

printf('the child process exited normally,with exit code %d \n',

WEXITSTATUS (child\_status));

else {

printf('the child process exited abnormally,with exit code %d \n',

WTERMSIG (child\_status));

Здесь функция wait() блокирует вызывающий процесс до тех пор, один из его дочерних процессов не завершится. Код состояния потомка возвращается через аргумент, являющийся указателем на целое число. В этом коде зашифрована различная информация о потомке, для извлечения которой используются макросы. Например, макрос WEXITSTATUS() возвращает код завершения дочернего процесса. Макрос WIFEXITED() позволяет узнать, как именно завершился процесс: обычным образом (с помощью функции exit() или оператора return функции main()) либо аварийно вследствие получения сигнала. В последнем случае макрос WTERMSIG() извлекает из кода завершения номер сигнала.

Наряду с функцией wait() используются и другие функции этого семейства. Функция waitpid() позволяет дождаться не любого, а конкретного дочернего процесса. Функция wait3() возвращает информацию о статистике использования центрального процессора завершившимся дочерним процессом. Функция wait4() позволяет задать дополнительную информацию о том, завершения каких процессов следует дождаться.

Функция wait() не только возвращает код завершения потомка, но и удаляет информацию о потомке из системных таблиц. Если же дочерний процесс завершается в отсутствии ожидающего его системного вызова wait(), он не удаляется из системы и переходит в состояние зомби. Удаление процесса-зомби (с помощью функции wait()) возлагается на родительский процесс. Если же родительский процесс завершится, не удалив из системы своих потомков, эту функцию выполнит процесс-демон init.

Именно этот вариант работы процессов представлен в программе на листинге 9. После компиляции и запуска программы можно перейти в другое консольное окно и посмотреть список активных процессов с помощью команды

ps –e –o pid,ppid,stat,cmd

Мы обнаружим в списке активных процессов и родительский, и дочерний процессы, причем последний будет обозначен как <defunct> (несуществующий), а его код состояния равен Z, т.е. zombie.

**Листинг 9. (*zombie.c*) Создание процесса-зомби**

#include <stdlib.h>

#include <sys/types.h>

#include <unistd.h>

int main ()

{

pid\_t child\_pid;

/\* Создание дочернего процесса. \*/

child\_pid = fork ();

if (child\_pid > 0) {

/\* Это родительский процесс – засыпает на 1 минуту. \*/

sleep (60);

}

else {

/\* Это дочерний процесс – немедленно завершает работу. \*/

exit (0);

}

return 0; /\* Завершение родительского процесса \*/

}

## Лекция №6. Создание и использование программных потоков в системе Linux

**Создание потока**

Современные операционные системы предоставляют возможность разбить процесс на ряд подпроцессов, называемых потоками, которые выполняются параллельно. Cистема Linux при создании процесса по умолчанию создает в нем единственный (основной) поток, последовательно выполняющий программный код. Этот поток может создавать дополнительные потоки. Все потоки процесса будут иметь доступ к общему виртуальному пространству, общим дескрипторам файлов и другим системным ресурсам. Если, к примеру, один поток меняет значение переменной, это изменение отразится на других потоках. Точно так же, когда один поток закрывает файл, остальные потоки теряют возможность работать с этим файлом. В связи с тем, что процесс и все его потоки могут выполнять лишь одну программу одновременно, как только один из потоков вызывает функцию семейства exec(), все остальные потоки завершаются (естественно, новая программа может создавать собственные потоки).

В Linux реализована библиотека API-функций работы с потоками, соответствующая стандарту POSIX (она называется Pthreads). Все функции и типы данных библиотеки объявлены в файле <pthread.h>. Эти функции не входят в стандартную библиотеку языка С, поэтому при компоновке программы нужно указывать опцию –lpthread в командной строке.

Для создания программного потока используется функция pthread\_create(). При создании поток получает идентификатор – тип данных pthread\_t. После создания поток начинает выполнять потоковую функцию - соответствующую часть программы. По завершении этой функции поток уничтожается. В Linux потоковая функция имеет следующий тип:

void\* (\*) (void\*)

Она принимает единственный параметр типа void\*, называемый аргументом потока, и возвращает значение аналогичного типа.

Программа, представленная в листинге 1, создает поток, который непрерывно записывает символы 'x' в стандартный поток ошибок. После вызова функции pthread\_create() основной поток начинает делать то же самое, но вместо символов 'x' печатаются символы 'o'.

Компиляция и компоновка программы осуществляется следующим образом:

gcc –o thread\_create thread\_create.c –lpthread

или

gcc –o thread\_create thread\_create.c –pthread

**Листинг 1. (*thread-create.c*) Создание дополнительного потока**

#include <pthread.h>

#include <stdio.h>

void\* print\_xs (void\* unused)

{

while (1)

fputc ('x', stderr);

return NULL;

}

int main ()

{

pthread\_t thread\_id;

pthread\_create (&thread\_id, NULL, &print\_xs, NULL);

while (1)

fputc ('o', stderr);

return 0;

}

После запуска программы можно увидеть, что символы 'x' и 'o' чередуются самым непредсказуемым образом.

**Передача данных программному потоку**

Потоковый аргумент – удобное средство передачи данных потокам. Но поскольку его тип void\*, данные содержатся не в самом аргументе. Он лишь должен указывать на какую-то структуру или массив. Лучше всего создать для каждой потоковой функции собственную структуру, в которой определялись бы "параметры", ожидаемые потоковой функцией.

В программе, приведенной в листинге 2, создаются два дополнительных потока: один отображает символы 'x', а другой – символы 'o'. Чтобы вывод на экран не длился бесконечно, потокам передается дополнительный аргумент, определяющий, сколько раз отобразить символ.

**Листинг 2. (*thread-create2.c*) Создание двух потоков**

#include <pthread.h>

#include <stdio.h>

struct char\_print\_parms

{

char character;

int count;

};

void\* char\_print (void\* parameters)

{

struct char\_print\_parms\* p = (struct char\_print\_parms\*) parameters;

int i;

for (i = 0; i < p->count; ++i)

fputc (p->character, stderr);

return NULL;

}

int main ()

{

pthread\_t thread1\_id;

pthread\_t thread2\_id;

struct char\_print\_parms thread1\_args;

struct char\_print\_parms thread2\_args;

/\* Создание потока для отображения 30000 символов 'x'. \*/

thread1\_args.character = 'x';

thread1\_args.count = 30000;

pthread\_create (&thread1\_id, NULL, &char\_print, &thread1\_args);

/\* Создание потока для отображения 20000 символов 'o'. \*/

thread2\_args.character = 'o';

thread2\_args.count = 20000;

pthread\_create (&thread2\_id, NULL, &char\_print, &thread2\_args);

/\* Ожидание завершения первого дополнительного потока \*/

pthread\_join (thread1\_id, NULL);

/\* Ожидание завершения второго дополнительного потока \*/

pthread\_join (thread2\_id, NULL);

return 0;

}

Здесь одна и та же функция char\_print() эксплуатируется обоими потоками, но каждый из них конфигурируется независимо с помощью структуры char\_print\_parms.

**Значения, возвращаемые программными потоками**

Функция pthread\_joint() ожидает завершения потока так же как функция wait() ожидает завершения процесса. Если второй аргумент функции pthread\_joint() не равен NULL, то в него помещается значение, возвращаемое потоком. Поскольку тип второго аргумента – указатель void\*, то возвращаемое функцией значение (например, типа int) сначала приводится к типу void\*, а затем уже вне функции pthread\_joint() обратно преобразуется в тип int.

Программа, представленная в листинге 3, в отдельном потоке вычисляет *n*-е простое число и возвращает его в программу. Тем временем функция main() может продолжать свои собственные вычисления. Отметим, что алгоритм последовательного деления, используемый в потоковой функции compute\_prime(), весьма неэффективен. Более мощным алгоритмом является, например, "решето Эратосфена".

**Листинг 3. (*primes.c*) Вычисление простых чисел в дополнительном потоке**

#include <pthread.h>

#include <stdio.h>

/\* Находим простое число с порядковым номером n, где n - это

значение, на которое указывает параметр arg. \*/

void\* compute\_prime (void\* arg)

{

int candidate = 2;

int n = \*((int\*) arg);

while (1) {

int factor;

int is\_prime = 1;

/\* Проверяем, является ли значение candidate простым числом. \*/

for (factor = 2; factor < candidate; ++factor)

if (candidate % factor == 0) {

is\_prime = 0;

break;

}

if (is\_prime) {

if (--n == 0)

/\* Возвращаем найденное число в основной поток. \*/

return (void\*) candidate;

}

++candidate;

}

return NULL;

}

int main ()

{

pthread\_t thread;

int which\_prime = 5000;

int prime;

/\* Запускаем поток, вычисляющий 5000-е простое число. \*/

pthread\_create (&thread, NULL, &compute\_prime, &which\_prime);

/\* Выполняем здесь другие действия ... \*/

pthread\_join (thread, (void\*) &prime);

printf("The %dth prime number is %d.\n", which\_prime, prime);

return 0;

}

Каждый поток имеет набор атрибутов. Потоковые атрибуты – это механизм настройки поведения отдельных потоков. При создании потока через второй аргумент функция pthread\_create() принимает указатель на объект атрибутов потока. Если этот указатель равен NULL, поток конфигурируется на основании стандартных атрибутов.

Для большинства Linux-приложений интерес представляет один единственный атрибут (остальные используются в приложениях реального времени): *статус отсоединения потока*. Поток может быть создан как *ожидаемый* (по умолчанию) или *отсоединенный*. Ожидаемый поток, подобно процессу, после своего завершения не удаляется автоматически операционной системой. Код его завершения хранится где-то в системе (как у процесса-зомби), пока какой-нибудь другой поток не вызовет функцию pthread\_joint(), чтобы запросить это значение. Только тогда ресурсы потока считаются освобожденными. С другой стороны, отсоединенный поток, завершившись, сразу уничтожается. Другие потоки не могут вызвать по отношению к нему функцию pthread\_joint(), чтобы получить возвращаемое значение.

**Отмена потока**

Обычно поток завершается при выходе из потоковой функции или вследствие вызова функции pthread\_exit(). Существует также возможность принудительного завершения (отмены) потока из другого потока с помощью функции pthread\_cancel(). При этом необходимо дождаться завершения потока (если он не отсоединенный) с целью освобождения ресурсов. Отмененный поток возвращает значение PTHREAD\_CANCELED.

Во многих случаях поток выполняет код, который нельзя просто взять и прервать. Например, поток может получить какие-то ресурсы, поработать с ними, а затем удалить. Если отмена потока произойдет где-то посередине, освободить занятые ресурсы станет невозможно, вследствие чего они окажутся потерянными для системы. Чтобы учесть эту ситуацию, поток должен определить, где и когда он может быть отменен.

С точки зрения возможности отмены поток находится в одном из трех состояний:

* *Асинхронно отменяемый*. Такой поток можно отменить в любой точке его выполнения.
* *Синхронно отменяемый*. Поток можно отменить в определенных местах, называемых точками отмены. Запрос на отмену помещается в очередь, и поток отменяется только при достижении очередной точки отмены.
* *Неотменяемый*. Попытки отменить поток игнорируются.

Первоначально поток является синхронно отменяемым. Чтобы сделать поток асинхронно отменяемым, используется функция pthread\_setcanceltype(). Эта функция влияет на тот поток, в котором она была вызвана. Поток может отказаться удаляться или переключаться, вызвав функцию pthread\_setcancelstate().

Эта функция позволяет организовывать критические секции. Критической секцией называется участок программы, который должен быть либо выполнен целиком, либо вообще не выполнен. Другими словами, если поток входит в критическую секцию, он во что бы ни стало должен дойти до ее конца.

Предположим, к примеру, что для банковской программы требуется написать функцию, осуществляющую перевод денег с одного счета на другой. Для этого нужно добавить заданную сумму на баланс одного счета и вычесть аналогичную сумму с баланса другого счета. Если между этими двумя операциями произойдет отмена потока, выполняющего функцию, программа ложно увеличит суммарный депозит банка вследствие незавершенной транзакции. Чтобы этого не случилось, обе операции должны выполняться в критической секции.

В листинге 4 показан пример функции process\_transaction(), осуществляющей данную задумку. Функция запрещает отмену потока до тех пор, пока баланс обоих счетов не будет изменен.

Отметим, что по окончании критической секции восстанавливается предыдущее состояние потока, а не режим PTHREAD\_CANCEL\_ENABLE, разрешающий отмену потока.

**Листинг 4. (*critical-section.c*) Защита транзакции с помощью критической секции**

#include <pthread.h>

#include <stdio.h>

#include <string.h>

/\* Массив балансов счетов, упорядоченных по номеру счета. \*/

float\* account\_balances;

int process\_transaction (int from\_acct, int to\_acct, float dollars)

{

int old\_cancel\_state;

/\* Проверяем баланс на счету from\_acct. \*/

if (account\_balances[from\_acct] < dollars)

return 1;

/\* Начало критической секции. \*/

pthread\_setcancelstate (PTHREAD\_CANCEL\_DISABLE, &old\_cancel\_state);

/\* Переводим деньги. \*/

account\_balances[to\_acct] += dollars;

account\_balances[from\_acct] -= dollars;

/\* Конец критической секции. \*/

pthread\_setcancelstate (old\_cancel\_state, NULL);

return 0;

}

**Область потоковых данных**

Все потоки программы делят общее адресное пространство. Тем не менее у каждого потока – свой собственный стек вызова. Это позволяет всем потокам выполнять разный код, а также вызывать функции традиционным способом. При каждом вызове функции в любом потоке создается отдельный набор локальных переменных, которые сохраняются в стеке этого потока.

Иногда все же требуется продублировать определенную переменную, чтобы у каждого потока была ее собственная копия. С этой целью операционная система Linux предоставляет потокам область потоковых данных. Переменные, сохраняемые в этой области, дублируются для каждого потока, что позволяет потокам свободно работать с ними, не мешая друг другу. Имеются специальные функции для чтения и записи значений, хранящихся в области потоковых данных.

Потоковые переменные должны иметь тип void\*. Ссылка на каждую переменную осуществляется по ключу. Для создания нового ключа, т.е. новой переменной, предназначена функция pthread\_key\_create(). Первым ее аргументом является указатель на переменную типа pthread\_key\_t. В нее будет записано значение ключа, посредством которого любой поток сможет обращаться к своей копии данных. Второй аргумент – это указатель на функцию очистки ключа. Она будет автоматически вызываться при уничтожении потока.

После того как ключ создан, каждый поток может назначать ему собственное значение, вызывая функцию pthread\_setspecific(). Ее первый аргумент – это ключ, а второй – требуемое значение типа void\*. Для чтения потоковых переменных предназначена функция pthread\_getspecific(), единственным аргументом которой является ключ.

Предположим, имеется приложение, распределяющее вычисления между несколькими потоками. В целях аудита за каждым потоком закреплен отдельный журнальный файл, куда записываются сообщения о ходе выполнения поставленной задачи. Область потоковых данных – удобное место для хранения указателя на журнальный файл каждого потока.

В листинге 5 показано, как это сделать. Для хранения файлового указателя в функции main() создается ключ, запоминаемый в переменной thread\_log\_key. Эта переменная является глобальной, поэтому она доступна всем потокам. Когда поток начинает выполнять свою потоковую функцию, он открывает журнальный файл и сохраняет указатель на него в своем ключе. Позднее любой поток может вызвать функцию write\_to\_thread\_log(), чтобы записать сообщение в свой журнальный файл. Эта функция извлекает из области потоковых данных указатель на журнальный файл и помещает в файл требуемое сообщение.

**Листинг 5. (*tsd.c*) Создание журнального файла для каждого потока**

#include <malloc.h>

#include <pthread.h>

#include <stdio.h>

/\* Ключ, связывающий указатель журнального файла с каждым потоком. \*/

static pthread\_key\_t thread\_log\_key;

/\* Запись параметра message в журнальный файл текущего потока. \*/

void write\_to\_thread\_log (const char\* message)

{

FILE\* thread\_log = (FILE\*) pthread\_getspecific (thread\_log\_key);

fprintf (thread\_log, "%s\n", message);

}

/\* Закрытие журнального файла, на который указывает

параметр thread\_log. \*/

void close\_thread\_log (void\* thread\_log)

{

fclose ((FILE\*) thread\_log);

}

void\* thread\_function (void\* args)

{

char thread\_log\_filename[20];

FILE\* thread\_log;

/\* Создание имени журнального файла для текущего потока. \*/

sprintf (thread\_log\_filename, "thread%d.log", (int) pthread\_self ());

/\* Открытие журнального файла. \*/

thread\_log = fopen (thread\_log\_filename, "w");

/\* Сохранение указателя файла в области потоковых данных

под ключом thread\_log\_key. \*/

pthread\_setspecific (thread\_log\_key, thread\_log);

write\_to\_thread\_log ("Thread starting.");

/\* Далее идет основное тело потока... \*/

return NULL;

}

int main ()

{

int i;

pthread\_t threads[5];

/\* Создание ключа, который будет связывать указатели

журнальных файлов с областью потоковых данных. Функция

close\_thread\_log() закрывает все журнальные файлы \*/

pthread\_key\_create (&thread\_log\_key, close\_thread\_log);

/\* Создание потоков. \*/

for (i = 0; i < 5; ++i)

pthread\_create (&(threads[i]), NULL, thread\_function, NULL);

/\* Ожидание завершения всех потоков. \*/

for (i = 0; i < 5; ++i)

pthread\_join (threads[i], NULL);

return 0;

}

Обратите внимание на то, что в функции thread\_function() не нужно закрывать журнальный файл. Просто когда создавался ключ, функция close\_thread\_log() была назначена функцией очистки данного ключа. Поэтому при любом завершении потока система Linux вызовет эту функцию, передав ей значение ключа, соответствующее данному потоку. В функции close\_thread\_log() и происходит закрытие файла.

## Лекция №7. Установка виртуального сервера Ubuntu Server 20.04 в среде Virtual Box

Ubuntu 20.04 LTS Server (Focal Fossa) имеет LTS-ядро версии 5.4, что означает появление ряда нововведений, улучшений безопасности, повышение производительности и добавление новых возможностей. Например: применён алгоритм сжатия ядра без потерь LZ4, что заметно сократило загрузку ОС; появилась поддержка по умолчанию файловой системы от Microsoft exFAT; задействован режим блокировки ядра Linux Lockdown Kernel, предотвращающий выполнение произвольного кода от UID даже с правами root. Кроме того, реализована полнофункциональная поддержка Кubernetes 1.18. Продолжается внедрение нового формата установочных пактов SNAP. Полностью прекращена поддержка 32-разрядных версий. Даже если у вас сейчас 32-разрядная Ubuntu 18.04, обновиться до версии 20.04 уже не получится.

До начала создания виртуальной машины-сервера рекомендуется выбрать виды создаваемых сетей на базе сервера и ввести эту информацию в настройки среды виртуализации VirtualBox, используя графический интерфейс, предоставляемый программой VirtualBox.

**Начальная настройка сети в программе VirtualBox**

Существует несколько способов создания сети в VirtualBox.

**NAT** - этот способ используется по умолчанию. Для каждой виртуальной машины создается отдельная внутренняя локальная сеть. Машина может связаться с интернетом, используя технологию NAT, и может обратиться к другой виртуальной машине сети, используя проброс портов. Однако хост доступа к виртуальным машинам не имеет.

**Виртуальный адаптер хоста** - создается виртуальный сетевой адаптер (виртуальный аналог сетевой карты), к которому можно подключить несколько виртуальных машин, тем самым объединив их в локальную сеть. Доступа к интернету нет, но зато машины находятся в одной сети и каждая имеет свой ip адрес, и они могут взаимодействовать между собой. Основная система (хост) тоже доступна по ip 192.168.56.1. Машины доступны не только между собой, но и из основной системы.

**Сетевой мост** - при таком подключении виртуальная машина становится полноценным членом локальной физической сети (внешней сети), к которой подключена основная система (хост). Машина использует сетевой интерфейс, чтобы получить адрес у роутера, и становится доступна для других устройств, как и основной компьютер по своему ip адресу.

**Внутренняя сеть** - почти то же самое, что и виртуальный адаптер хоста, только без возможности доступа к виртуальной сети из основной системы, доступа к интернету нет.

Итак, остановим сервер Ubuntu (если он уже установлен и запущен), закроем окно с терминалом, и уже в главном окне VirtualBox идем в Файл > Настройки > Сеть > Виртуальные сети хоста — справа три маленькие иконки — жмем на нижнюю с изображением отвертки:

Изображение выглядит как текст, программное обеспечение, Значок на компьютере, веб-страница

Автоматически созданное описание

Видим IP-адрес: 192.168.56.1 — это адрес, по которому Windows (наш хост) обращается к VirtualBox:

Изображение выглядит как текст, программное обеспечение, Значок на компьютере, веб-страница

Автоматически созданное описание

Рядом вкладка «DHCP сервер» в нашем случае она пуста (чекбокс не выбран). И пусть пока таковой и остается:

Изображение выглядит как текст, программное обеспечение, Значок на компьютере, веб-страница

Автоматически созданное описание

**Создание виртуальной машины для сервера**

Минимальные требования Ubuntu 20.04 LTS Server к аппаратуре (по данным с официальной страницы Canonical):

* процессор: одноядерный, 2 ГГц;
* ОЗУ: 2 Гб;
* жёсткий диск: 10 Гб.

Рекомендуемые характеристики для получения среды с приемлемым уровнем комфорта по производительности:

* процессор: двухъядерный, 2 ГГц;
* ОЗУ: 4 Гб;
* жёсткий диск: 25 Гб.

Откройте VirtualBox и нажмите на кнопку Создать. Укажите имя виртуальной машины, тип и версию (рис. 1).

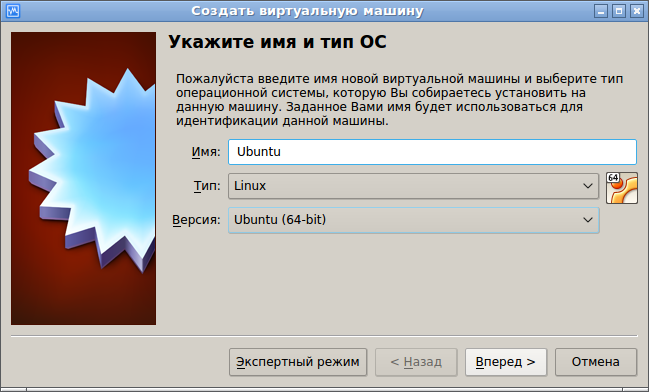


Рис. 1. Указание имени и типа виртуальной машины для сервера

Укажите размер оперативной памяти (рис. 2).

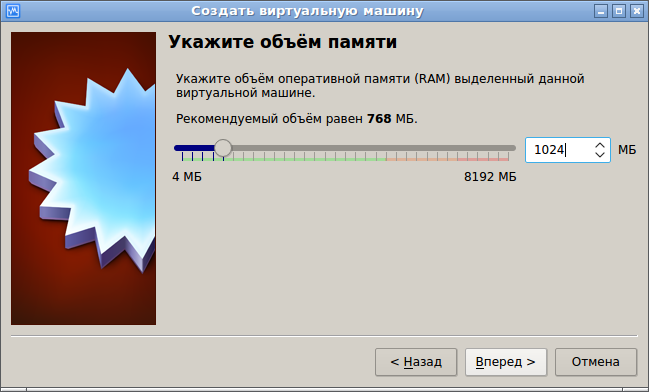


Рис. 2. Указание объема оперативной памяти

Создайте новый виртуальный диск (рис. 3).

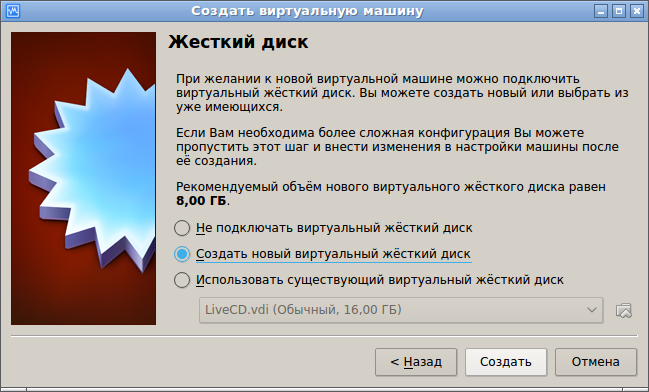


Рис. 3. Указание вида виртуального диска

Укажите тип жесткого диска VDI (рис. 4).

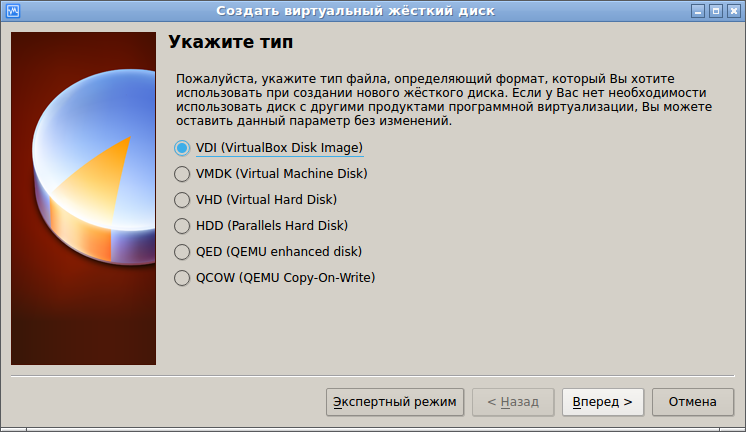


Рис. 4. Указание типа виртуального диска

Укажите формат хранения: динамический виртуальный жесткий диск (рис. 5).

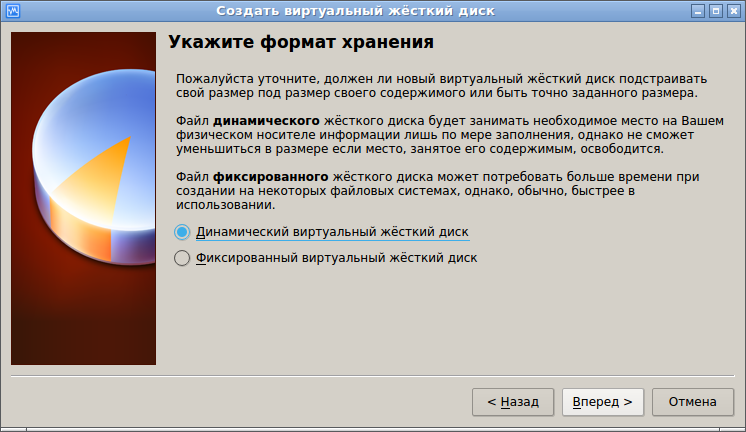


Рис. 5. Указание формата хранения на виртуальном диске

Укажите имя и размер виртуального диска (рис. 6).

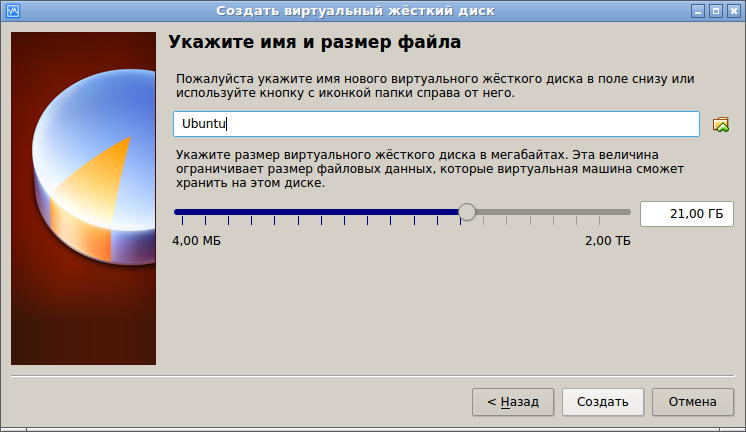


Рис. 6. Указание имени, размера и места хранения виртуального диска

После создания виртуальной машины, откройте ее настройки, перейдите на вкладку Носители, и укажите скачанный ISO диск (рис. 7).

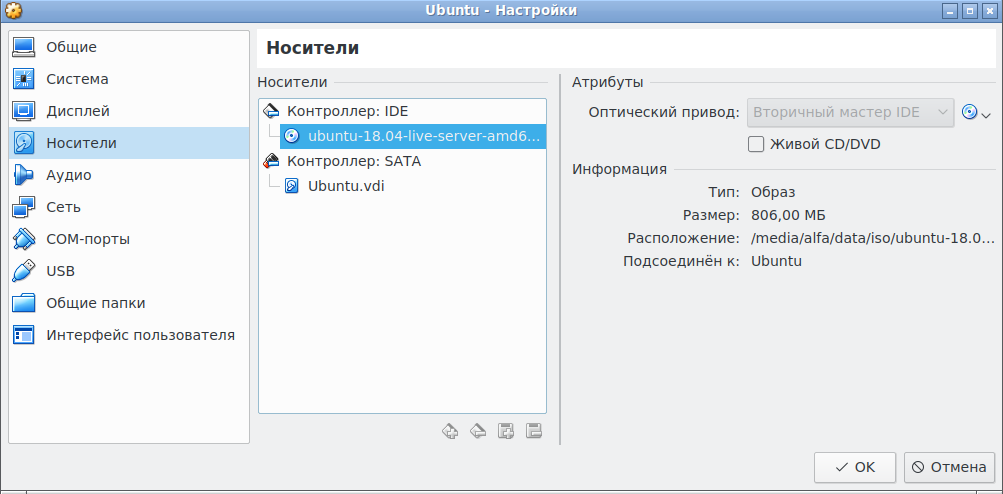


Рис. 7. Дистрибутив сервера установлен в виде ISO-диска в оптическом приводе

Перейдите в раздел Сеть. Выберите NAT или сетевой мост с именем virtualbox (рис. 8).

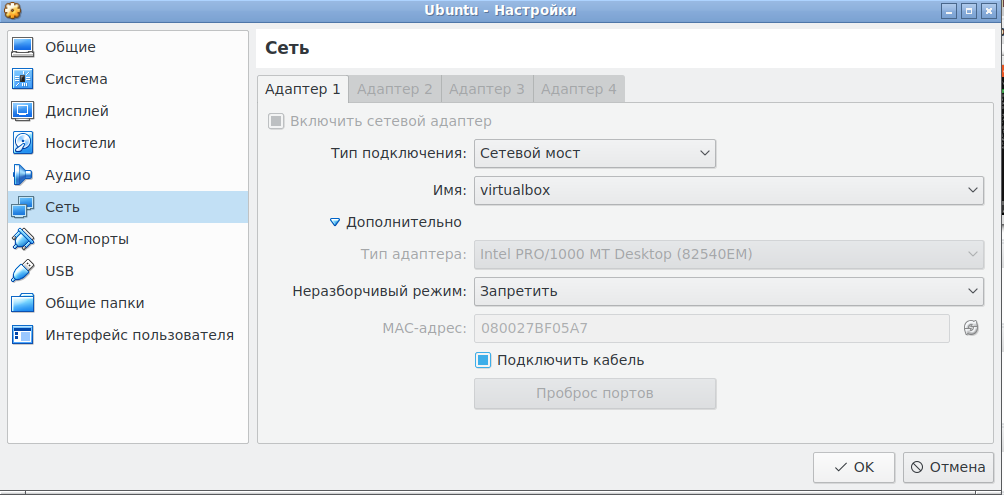


Рис. 8. Подключение сетевого адаптера (виртуальной сетевой карты сервера)

Если вы выбрали подключение NAT, то, возможно, вам нужно пробросить порты (рис. 9).

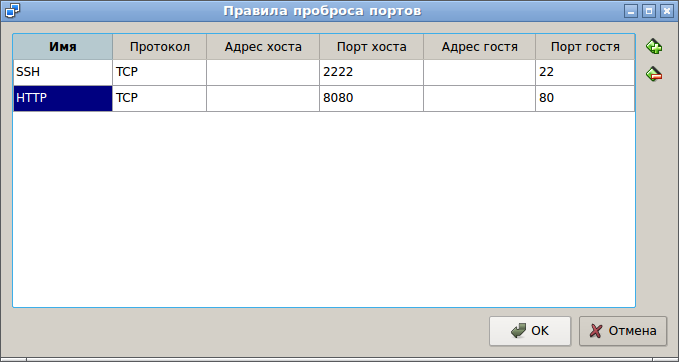


Рис. 9. Задание правил проброски портов

**Дополнительные сетевые настройки гостевой системы**

После установки идем в настройки гостевой системы (в нашем случае — Ubuntu 16.04.2 Server), щелкаем правой кнопкой на вкладке с системой и выбираем самый верхний пункт «Настроить...»:

Изображение выглядит как текст, снимок экрана, программное обеспечение, Значок на компьютере

Автоматически созданное описание

Здесь снова идем в «Сеть». Здесь (если мы раньше не выполняли сетевых настроек для сервера) мы видим, что для первого адаптера отмечена NAT. Не меняем эту настройку:

Изображение выглядит как текст, снимок экрана, дисплей, программное обеспечение

Автоматически созданное описание

Открываем вкладку «Адаптер 2», ставим галку на «Включить сетевой адаптер» и в списке выбираем тип подключения: «Виртуальный адаптер хоста». Жмем ОК. Это действие добавит нам в Ubuntu сетевой адаптер, через который мы и прорубим окно во внешний мир, и в которое можно будет посмотреть на Ubuntu из Windows через браузер или SSH-клиент:

Изображение выглядит как текст, снимок экрана, программное обеспечение, дисплей

Автоматически созданное описание

Далее идем в окно терминала Ubuntu, логинимся (если еще нет), смотрим какие сетевые интерфейсы нам доступны:

Изображение выглядит как текст, снимок экрана, Шрифт

Автоматически созданное описание

Здесь мы видим, что есть три интерфейса: lo — локальная петля, enp0s3 — это «Адаптер 1» который у нас настроен как NAT (через него идет интернет-трафик в Ubuntu) и, наконец, enp0s8 — это и есть «Адаптер 2» («Виртуальный адаптер хоста»). Теперь осталось его настроить.

Запускаем mc от имени суперпользователя:

sudo mc

И открываем для правки файл /etc/network/interfaces (переводим на него курсор и жмем F4), либо просто командой

mcedit /etc/network/interfaces

И добавляем туда в конец текста такие строки:

auto enp0s8

iface enp0s8 inet static

address 192.168.56.107

netmask 255.255.255.0

Получается что-то вроде как показано на скриншоте:

Изображение выглядит как текст, снимок экрана, Шрифт

Автоматически созданное описание

Здесь мы нашему «Адаптеру 2» дали статический IP-адрес: 192.168.56.107 (можете поменять его на 192.168.56.101 например, или 192.168.56.110, без разницы). И теперь через этот адрес Ubuntu будет видна из Windows-хоста.

Сохраняем изменения — F2 и выходим из редактора — F10. Далее перезапускаем интерфейсы, например так:

sudo service networking restart

или можете просто перезагрузить Ubuntu Server.

**Утилита NetPlan для управления настройками сети**

Настройка сети в Ubuntu Server 20.04 LTS осуществляется через утилиту **Netplan**.

**NetPlan** — это инструмент для управления настройками сети, представленный в Ubuntu начиная с версии 17.10 и выше.

Этот инструмент заменяет файл статических интерфейсов /etc/network/interfaces, который ранее использовался для настройки сетевых интерфейсов в Ubuntu. Теперь нужно использовать /etc/netplan/\*.yaml для ваших изменений в настройках сетевых интерфейсов.

Предположим, что необходимо заменить IP адрес 192.169.0.150 на IP адрес 192.169.0.105. Отредактируем файл конфигурации **netplan,** который находится в директории /etc/netplan/. Вместо mcedit можно использовать редактор nano:

sudo nano /etc/netplan/00-installer-config.yaml

Файл после редактирования:

network:

ethernets:

enp0s3:

addresses:

- 192.168.0.105/24

gateway4: 192.168.0.1

nameservers:

addresses: [192.168.0.1, 8.8.4.4]

optional: true

version: 2

renderer: networkd

Обратите внимание на пробелы! Именно пробелы должны быть, а не табуляция. Если у вас после сохранения файла не появилась ошибка типа: **Error while loading /etc/netplan/00-installer-config.yaml**, то файл отредактирован правильно с точки зрения его синтаксиса.

Использование **netplan** для генерации необходимой конфигурации:

sudo netplan generate

Для подробного вывода информации при генерации, используйте опцию --debug:

sudo netplan --debug generate

Применение конфигурации **netplan**:

sudo netplan apply

Для подробного вывода информации при применении, используйте опцию --debug.

Далее рекомендуется перезапустить интерфейсы или перезагрузить сервер:

sudo reboot

## Лекция №8. Создание рабочей среды в Arch Linux на базе файловых менеджеров окон

**Оконные менеджеры**

**Оконный менеджер** (window manager, WM) - это программа, которая управляет окнами приложений в системе, а именно: осуществляет позиционирование окон на экране, отвечает за изменение их размера, фокусировку и прочее. При этом оконный менеджер работает поверх существующей оконной системы. В простейшем случае, оконный менеджер просто управляет окнами и взаимодействием с ним оборудования ввода, и не делает с ними ничего другого. При этом нагрузка за отрисовку ложится на центральный процессор (так называемая **программная отрисовка**). Некоторые оконные менеджеры, помимо управления окном, могут отрисовывать тени на его гранях, добавлять различные анимации, плавности, полупрозрачности и так далее. В этом случае, оконный менеджер является **композитным**. Одной из важнейших (но необязательных) функций композитного оконного менеджера является задействование для отрисовки окна возможностей видеокарты, как правило посредством **OpenGL**. Тем самым увеличивая качество отрисовки и значительно снижая нагрузку на центральный процессор. Функционал композитного менеджера может быть как встроенным в оконный менеджер, так и являться отдельной программой. Часто отрисовку окна через композитный оконный менеджер называют просто - **композитингом**. Пример из Windows 7: когда в ней отключены эффекты **Aero**, отрисовка ведётся силами центрального процессора. Нагрузка на видеокарту меньше, однако при воспроизведении видео на экране появляются артефакты, известные как тиринг (когда кадры меняются слишком быстро и посередине видно прозрачную мерцающую полосу).

Когда эффекты Aero включены - отрисовка ведётся силами видеокарты. Что становится очевидным, так как появляются анимации появления и сворачивания окна, полупрозрачности и так далее.

Однако, композитинг имеет и обратную сторону. При отрисовке рабочего стола силами видеокарты частота кадров в секунду синхронизируется с частотой монитора (как правило 60 кадров в секунду, что соответствует стандартным 60 герцовым мониторам), поэтому в играх частота будет несколько ниже, так как будет идти двойная синхронизация кадра. В тяжёлых случаях - производительность игры может упасть вдвое. Поэтому часто можно встретить рекомендации отключать графические эффекты при запуске игр.

В данный момент доминирующей графической подсистемой в Линуксе является Xorg (иксы). И работа оконного менеджера в ней точно такая, как описана выше. При этом функционал композитинга не был изначально в Xorg, и его реализовали значительно позже, потому композитный менеджер работает там как-бы сбоку. В общем случае, получается весьма толстый бутерброд различных слоёв, через который происходит отрисовка картинки.

Первым широко известным оконным менеджером композитинга для X был Xfwm. Менеджер окон Compiz предоставляет ускоренный 3D-композитинг для платформы Linux. KDE KWin также поддерживает композитинг.

Сейчас на смену Xorg идут две графические подсистемы - Wayland и Mir. Первый не привязан к какому-либо дистрибутиву или графической оболочке, в то время как второй разрабатывается для Ubuntu и её графической оболочки Unity, и уже работает в мобильной редакции Ubuntu. Что касается Wayland - в нём отсутствуют привычные понятия оконного и композитного менеджера. В нём есть только композитор, который и производит все операции над окнами, без лишних прослоек. Отрисовка приложения при этом ложится на программный инструмент (тулкит), на котором оно написано, например на Qt или GTK. Это так называемая отрисовка на стороне клиента. В случае же если отрисовка окна идёт непосредственно в Wayland-композиторе, то это называется отрисовкой на стороне сервера. В случае отрисовки на стороне клиента, заголовок окна приложения, его внешний вид и прочее целиком ложится на разработчика приложения.

Wayland уже работает в автомобильных ОС, в мобильных операционках Tizen и SailfishOS, и много где ещё. Однако Wayland пока не поддерживается фирменными проприетарными драйверами Nvidia и AMD, что сильно затрудняет его введение в дистрибутивах Linux.

Наряду с композитными существуют стековые, фреймовые и динамические менеджеры окон:

* **стековые** (также *плавающие*) оконные менеджеры следуют традиционной метафоре рабочего стола, которая используется в коммерческих операционных системах Windows и OS X. Окна отображаются подобно листкам бумаги на столе, накладываясь и перекрывая друг друга.
* **фреймовые** (тайловые) оконные менеджеры располагают окна на экране в виде плиток (фреймов) так, что они не перекрывают друг друга. Как правило, фреймовые оконные менеджеры подразумевают активное использование клавиатуры для управления окнами, и имеют слабую поддержку мыши (либо не имеют ее вовсе). Фреймовые оконные менеджеры могут предлагать набор стандартных расположений фреймов или позволять задавать их вручную.
* **динамические** оконные менеджеры позволяют динамически переключаться между двумя режимами — стековым и фреймовым.

**Основные оконные менеджеры системы Linux**

**Стековые менеджеры**

В Linux оконные менеджеры считаются легкими, более быстрыми альтернативами более раздутым и модным окружениям рабочего стола, таким как Gnome Shell, KDE Plasma 5, Cinnamon и т. д. Перечислим наиболее известные стековые менеджеры (в круглых скобках указано, в какой среде рабочего стола менеджер используется по умолчанию):

* mutter (GNOME3),
* marco (Mate),
* metacity (GNOME2),
* muffin (Cinnamon),
* kwin (KDE),
* xfwm (XFCE),
* twm.

Более подробно рассмотрим некоторые тайловые менеджеры окон.

**Основные тайловые менеджеры**

**Awesome** — это настраиваемая среда оконного менеджера «следующего поколения» для графического сервера Xorg / X11. Работает очень быстро и поддерживает множество режимов работы с окнами, включая мозаичное размещение окон, плавающее окно (то, что делают большинство современных оконных менеджеров среды рабочего стола Linux) и многие другие стили. Он поддерживает одновременно мозаичный, плавающий и полноэкранный режим. В Awesome есть панель с несколькими предварительно настроенными тегами, на которые легко переключаться, и пользователи могут использовать их для организации и сортировки программ для конкретных целей.Отличная поддержка дисплеев с несколькими экранами (двойные мониторы) с помощью инструментов XR и R.Оконный менеджер Awesome написан на скриптовом языке Lua. Благодаря этому ядро ​​оконного менеджера легко настраивается, и существует огромное количество плагинов Lua, которые доступны.

Пакет awesome обычнопредоставляется дистрибутивом.

**Xmonad** — это динамический оконный менеджер для Linux. Его основная цель — уменьшить необходимость для пользователей масштабировать и выравнивать свои окна в определенных плитках, делая это автоматически.

Программы в Xmonad выровнены совсем иначе, чем в других менеджерах листов в Linux. Вместо того, чтобы иметь промежутки между плитками, WM полностью удаляет их, чтобы максимизировать пространство экрана пользователя.

Xmonad имеет «яркую» библиотеку расширений, и пользователи могут настраивать свои настройки в соответствии с потребностями своего рабочего стола. Оконный менеджер Xmonad может автоматически сортировать и мозаично размещать ваши окна, экономя время. Предоставляет пользователям экранные рабочие пространства и поддерживает оформление окон. Xmonad поддерживает большинство основных панелей рабочего стола, а также работает с Gnome и KDE.

**Ratpoison** — это чрезвычайно легкий оконный менеджер, который претендует на то, чтобы «не иметь толстых библиотечных зависимостей», что делает его идеальным дополнением к ПК с Linux, у которого не так много свободного места на диске и ресурсов.

Пользователи перемещаются по Ratpoison так же, как и по любому другому тайловому оконному менеджеру. Диспетчер окон поддерживает как режим максимизации, так и настройку листов, которая есть в большинстве оконных менеджеров на платформе Linux.

Ratpoison запускает все окна в полноэкранном режиме, когда используется одна программа, и старается максимально увеличить площадь экрана. Чрезвычайно легкий и, пожалуй, один из лучших оконных менеджеров для использования на компьютере почти без ресурсов. Комбинации клавиш в оконном менеджере Ratpoison специально разработаны, чтобы не мешать другим программам, поэтому он не мешает вашему рабочему процессу.

Менеджер **i3** - это бесплатный и полностью настраиваемый оконный менеджер с открытым исходным кодом, предназначенный для опытных пользователей Linux и BSD, а также разработчиков. Он имеет древовидную структуру данных, которая обеспечивает более гибкую компоновку, чем его альтернативы, и не требует использования Haskell или LUA.

i3является одним из самых популярных приложений для управления окнами благодаря своим функциям, таким как текстовые настройки, настраиваемые сочетания клавиш и изменения конфигурации на лету, без необходимости перезагружать базовую систему.

Пакет i3предоставляется дистрибутивом, просто воспользуйтесь менеджером пакетов как показано ниже:

$ sudo apt install i3 [On Debian/Ubuntu]

**tmux** - это терминальный мультиплексор с открытым исходным кодом, который позволяет пользователям создавать несколько сеансов терминала, к которым они могут обращаться и управлять ими с одного экрана, что делает его идеальным для одновременного запуска нескольких программ командной строки.

tmuxпускает в дело все доступное пространство, и его легко использовать благодаря поддержке комбинаций клавиш, с помощью которых можно разделять окна или создавать большее количество панелей. Также вы можете сохранить экземпляр оболочки для использования в других сессиях и расшарить его для разных пользователей.

**Установка и настройка i3wm в Arch Linux**

Профессиональные программисты, работающие на Linux, часто отказываются от использования громоздкого графического интерфейса на базе рабочих столов (Desktop Environment, DE), таких как GNOME, KDE или Xfce. Более того, им не нужен и классический менеджер окон, ибо двигать окна мышкой и накладывать их друг на друга - это контрпродуктивно. Вместо этого они установливают тайловый менеджер окон, который не разрешает перемещать окна по экрану, а вместо этого либо всегда открывает все окна на полный экран, либо делит экран между окнами пропорционально (или непропорционально), не позволяя им накладываться друг на друга.

Считается, что лучший тайловый менеджер окон из доступных на сегодняшний день — это i3. Отметим, что i3 не единственная в своем классе, есть и другие вроде awersome, sway и т.д. Менеджер i3 рекомедуется использовать по следующим причинам.

Во-первых, i3 создан для работы с клавиатурой, и в работе с ним волей не волей пользуешься горячими клавишами. Горячие клавиши тут играют важную роль. И профессионалам это обычно нравится.

Во-вторых, i3 благоволит минималистичности. Минимум UI нагроможденностей. Все настройки системы производятся из файловых конфигов. Можно использовать любые программы, но тут очень хорошо подходят именно минималистичные, консольные утилиты вроде ranger как файловый менеджер, и тому подобное.

В третьих, это минимальные системные требования. Сам i3 легковесен и не прожорлив. Потребляет минимум оперативной памяти. Если у Вас слабое железо, то это несомненно большой плюс.

В четвертых, это, наверное, уникальность. И это не может не радовать. И то, как быстро и удобно можно решать с помощью нее свои задачи, часто восхищает.

i3 работает на графическом сервере x-org, поэтому нам потребуется установить его. Также мы установим не сам оригинальный i3wm, а его немного модернизированный форк i3-gaps. Еще сразу нам понадобится программа i3status для отображения системного бара, и программа dmenu для запуска установленных программ (которую в последствии заменим на аналог, rofi):

sudo pacman -S xorg-server xorg-xdm i3-gaps i3status dmenu ttf-dejavu

После установки создадим и отредактируем файл ~/.xsession, записав в него текст “exec i3”:

echo 'exec i3' >> ~/.xsession

chmod +x ~/.xsession

Эта инструкция при запуске графического сервера xorg запустит наш i3wm.

В качестве эмулятора терминала выберем mate-terminal:

sudo pacman -S mate-terminal

Далее включим службу XDM:

$ sudo systemctl enable xdm.service

Установим polkit для общесистемных привилегий:

$ sudo pacman -S polkit

Сервису Polkit нужен агент аутентификации, которого i3 не предоставляет по умолчанию. Можно использовать lxsession:

$ sudo pacman -S lxsession

Добавьте lxsession & в ваш файл .xsession перед строкой exec i3.

И, наконец, выполняем перезагрузку:

reboot

Появляется менеджер входа. После авторизации появляется окно менеджера i3:

Изображение выглядит как текст, снимок экрана, программное обеспечение, Мультимедийное программное обеспечение

Автоматически созданное описание

При самом первом запуске i3 нас спросят о том, какую клавишу использовать по умолчанию для модификатора команд (клавиши **mod**). Оставим клавишу Win. Далее i3 спросит о создании конфигурационного файла в домашней директории, согласимся, нажав Enter. Путь к файлу нам понадобится неоднократно для кастомизации (настройки параметров конфигурации i3).

После этого диалоговое окно исчезнет и перед нами останется черный экран менеджера i3, поскольку в нем не установлены обои.

Новичку потребуется выучить горячие клавиши для работы. Самое базовое, что нужно знать сейчас, это:

* **mod + Enter** - запустить терминал;
* **mod + d** - запустить dmenu для запуска любой программы;
* **mod + Shift + q** - закрыть активное окно программы;
* **mod + 1, mod + 2, mod + …n** - переключение рабочих столов от 1 до 9;
* **mod + Shift + 1, mod + Shift + … n** - перемещение активного окна на другой рабочий стол;
* **mod + Shift + … ←, →, ↑, ↓** - изменение положения окон в рамках рабочего стола;
* **mod + r** - изменение размеров активного окна;
* **mod + Shift + c** - перечитать конфиг;
* **mod + Shift + r** - перезапуск i3;
* **mod + Shift + e** - выход из i3 с подтверждением;
* **mod + Shift + space** - перевод окна в плавающий режим.

Это далеко не все хоткеи (горячие клавиши), все остальное можно увидеть на официальном сайте.

Любые биндинги (определения) горячих клавиш и остальные настройки i3 хранятся в файле ~./config/i3/config.

**Основные операции i3wm**

Ключевое слово $mod мы будем использовать для обозначения настроенного выше модификатора. Ранее использовалась клавиша Alt (Mod1), однако сейчас клавиша Windows (Mod4) стала популярной альтернативой, которая в значительной степени предотвращает конфликты с горячими клавишами, используемыми различными приложениями. Мы будем использовать клавишу Win (Mod4) в качестве $ mod.

**Открытие терминалов и перемещение**

Одна из самых простых операций - это открытие нового терминала. По умолчанию привязка клавиш для этого - $ mod + Enter , то есть Win + Enter ( Mod4 + Enter ). При нажатии $ mod + Enter откроется новый терминал. Он заполнит все пространство на вашем экране:

Изображение выглядит как текст, снимок экрана, дисплей, программное обеспечение

Автоматически созданное описание

Если вы сейчас откроете другой терминал, i3 поместит его рядом с текущим, разделив размер экрана пополам. В зависимости от вашего монитора, i3 поместит созданное окно рядом с существующим окном (на широких дисплеях) или под существующим окном:

Изображение выглядит как текст, снимок экрана, Шрифт, программное обеспечение

Автоматически созданное описание

Чтобы переместить фокус между двумя терминалами, вы можете использовать клавиши направления, которые вы, возможно, знаете из редактора vi . Итак, чтобы переключаться между терминалами, используйте $ mod + k или $ mod + l . Конечно, вы также можете использовать клавиши со стрелками.

На данный момент ваше рабочее пространство разделено (оно содержит два терминала) в определенном направлении (по умолчанию горизонтально). Каждое окно можно снова разделить по горизонтали или вертикали, как и рабочее пространство (рабочий стол).

Чтобы разделить окно по вертикали, нажмите $ mod + v перед созданием нового окна. Чтобы разделить его по горизонтали, нажмите $ mod + h .

**Изменение макета контейнера**

Разделенный контейнер может иметь один из следующих макетов: разделение (splith/splitv), наложение (stacking), с вкладками (tabbed). Рассмотрим их:

splith/splitv

Размер окон такой, что каждое окно занимает в контейнере равное количество места. splith распределяет окна по горизонтали (окна расположены рядом друг с другом), splitv распределяет их по вертикали (окна расположены друг над другом).

stacking

Отображается только сфокусированное (имеющее фокус) окно в контейнере. Вы увидите список окон вверху контейнера.

tabbed

Тот же принцип, что и наложение, но список окон вверху представляет собой только одну строку, разделенную по вертикали.

Чтобы переключать режимы, нажмите $ mod + e для splith/splitv (он переключается), $ mod + s для наложения и $ mod + w для вкладок:

Изображение выглядит как текст, снимок экрана, программное обеспечение, Мультимедийное программное обеспечение

Автоматически созданное описание

**Переключение полноэкранного режима для окна**

Чтобы отобразить окно в полноэкранном режиме или снова выйти из полноэкранного режима, нажмите $ mod + f .

**Открытие других приложений**

Помимо открытия приложений с терминала, вы также можете использовать приложение dmenu, которое по умолчанию открывается нажатием $ mod + d . Просто введите имя (или его часть) приложения, которое вы хотите открыть. Соответствующее приложение должно быть в вашем $ PATH, чтобы это работало.

Кроме того, если у вас есть приложения, которые вы открываете очень часто, вы можете создать привязку клавиш для непосредственного запуска приложения.

**Закрытие окон**

Если приложение не предоставляет механизм для закрытия (в большинстве приложений есть меню, клавиша выхода или комбинация клавиш, например Control + с для закрытия), вы можете нажать $ mod + Shift + q, чтобы закрыть любое окно. Для приложений, поддерживающих протокол WM\_DELETE, это приведет к правильному закрытию приложения (с сохранением любых изменений или выполнением другой очистки). Если приложение не поддерживает протокол WM\_DELETE, ваш X-сервер закроет окно, и его поведение будет зависеть от приложения.

**Использование рабочих областей**

Рабочие области (рабочие столы) - это простой способ сгруппировать набор окон. По умолчанию вы находитесь в первой рабочей области, как показывает метка в нижнем левом углу. Чтобы переключиться на другое рабочее пространство, нажмите $ mod + num, где num - номер рабочего пространства, которое вы хотите использовать. Если рабочее пространство еще не существует, оно будет создано.

Распространенная парадигма - разместить веб-браузер в одной рабочей области, коммуникационные приложения ( mutt , irssi ,…) в другой, а те, с которыми вы работаете, в третьей. Конечно, нет необходимости следовать этому подходу.

Если у вас несколько экранов, при запуске на каждом экране будет создано рабочее пространство. Если вы откроете новое рабочее пространство, оно будет привязано к экрану, на котором вы его создали. Когда вы переключаетесь на рабочее пространство на другом экране, i3 установит фокус на этот экран.

**Перемещение окон в рабочие области**

Чтобы переместить окно в другое рабочее пространство, просто нажмите $ mod + Shift + num, где num - это (например, при переключении рабочих пространств) номер целевого рабочего пространства. Подобно переключению рабочих пространств, целевая рабочая область будет создана, если она еще не существует.

**Изменение размера**

Самый простой способ изменить размер контейнера - использовать мышь: возьмите границу и переместите ее до нужного размера.

**Перезапуск i3 на месте**

Чтобы перезапустить i3 на месте (и, таким образом, перейти в чистое состояние, если есть ошибка, или для обновления до более новой версии i3), вы можете использовать $ mod + Shift + r .

**Выход из i3**

Чтобы полностью выйти из i3 без остановки вашего X-сервера, вы можете использовать $ mod + Shift + e . По умолчанию в диалоговом окне вас попросят подтвердить, действительно ли вы хотите выйти.

**Плавающий режим**

Плавающий режим противоположен режиму мозаичного (тайлового) изображения. I3 не управляет положением и размером окна автоматически, а управляется вручную вами. Использование этого режима нарушает парадигму мозаики, но может быть полезно для некоторых случаев, таких как диалоговые окна «Сохранить как» или окна панели инструментов (GIMP или аналогичные). Эти окна обычно устанавливают соответствующую подсказку и по умолчанию открываются в плавающем режиме.

Вы можете переключить плавающий режим для окна, нажав $ mod + Shift + Space. Перетаскивая мышью строку заголовка окна, вы можете перемещать окно. Захватывая границы и перемещая их, вы можете изменить размер окна. Другой способ изменить размер плавающих окон с помощью мыши - щелкнуть правой кнопкой мыши по строке заголовка и перетащить.

Плавающие окна всегда находятся поверх окон мозаики.

**Деревья из рабочих столов, контейнеров и окон терминалов**

**Дерево состоит из контейнеров**

Строительными блоками нашего дерева являются так называемые контейнеры . В контейнере можно разместить одно или несколько окон. В качестве альтернативы контейнер может содержать один или несколько контейнеров . Таким контейнером является рабочая область (можно использовать термин "рабочий стол"): когда вы запускаете i3 с одним монитором, одной рабочей областью и открываете два окна терминала, вы получите такое дерево:

Изображение выглядит как текст, Мультимедийное программное обеспечение, программное обеспечение, Графическое программное обеспечение

Автоматически созданное описание

**Ориентация и разделение контейнеров**

Естественно использовать так называемые разделенные контейнеры для построения макета при использовании дерева в качестве структуры данных. В i3 каждый контейнер имеет ориентацию (горизонтальную, вертикальную или неопределенную), и ориентация зависит от макета, в котором находится контейнер (вертикальный для разделения и стекирования, горизонтальный для разделения и с вкладками). Итак, в нашем примере с рабочей областью макет контейнера рабочей области по умолчанию - разделенный (в настоящее время большинство мониторов широкоэкранные). Если вы измените макет на splitv ( $ mod + v в конфигурации по умолчанию), а **затем** откроете два терминала, i3 настроит ваши окна следующим образом:

Изображение выглядит как текст, снимок экрана, программное обеспечение, Мультимедийное программное обеспечение

Автоматически созданное описание

Интересная новая функция i3, начиная с версии 4, - это возможность разделить что угодно: предположим, что у вас есть два терминала в рабочем пространстве (с разделенным макетом, то есть горизонтальной ориентацией), фокус находится на правом терминале. Теперь вы хотите открыть другое окно терминала под текущим. Если бы вы просто открыли новое окно терминала, оно бы появилось справа из-за макета разделения. Вместо этого нажмите $ mod + v, чтобы разделить контейнер с макетом splitv (чтобы открыть контейнер с горизонтальным разделением , используйте $ mod + h ). Теперь вы можете открыть новый терминал, и он откроется под текущим:

Изображение выглядит как текст, снимок экрана, Мультимедийное программное обеспечение, программное обеспечение

Автоматически созданное описание

Вы, наверное, уже догадались: нет никаких ограничений на то, насколько глубокой может быть ваша иерархия разбиений.

**Родительский фокус**

Остановимся на нашем примере сверху. У нас есть терминал слева и два вертикально разделенных терминала справа, фокус находится на правом нижнем. Когда вы открываете новый терминал, он откроется под текущим.

Итак, как вы можете открыть новое окно терминала **справа** от текущего? Решение состоит в том, чтобы использовать родителя фокуса, который будет фокусировать родительский контейнер текущего контейнера. В конфигурации по умолчанию используйте $ mod + a для перемещения по одному контейнеру вверх по дереву (вы можете повторять это несколько раз, пока не дойдете до контейнера рабочей области ). В этом случае вы должны сфокусировать контейнер вертикального разделения, который находится **внутри** горизонтально ориентированного рабочего пространства. Таким образом, теперь справа от контейнера вертикального разделения будут открываться новые окна :

Изображение выглядит как текст, снимок экрана, программное обеспечение, Мультимедийное программное обеспечение

Автоматически созданное описание

**Классификация схем контейнеров**

Итак, i3 группирует окна в контейнеры. Всего есть 4 типа контейнеров.

1. Горизонтально разделённый контейнер (horizontal-split):

Изображение выглядит как снимок экрана, диаграмма, Прямоугольник, текст

Автоматически созданное описание

2. Вертикально разделённый контейнер (vertical-split):

Изображение выглядит как снимок экрана, Прямоугольник, диаграмма, линия

Автоматически созданное описание

3. Контейнер с табами (tabbed):

Изображение выглядит как текст, диаграмма, Прямоугольник, дизайн

Автоматически созданное описание

4. Стековый контейнер (stacked):

Изображение выглядит как Прямоугольник, диаграмма, дизайн, снимок экрана

Автоматически созданное описание

Горячие клавиши для преобразования схем:

Win + E переводит горизонтальный контейнер в вертикальный и наоборот,

Win + W переводит контейнер в табовый,

Win + S переводит контейнер в стековый.

Мы можем создать новый контейнер на месте активного окна:

Win + V создаёт вертикальный контейнер,

Win + H создаёт горизонтальный контейнер.

Пусть контейнеры организованы следующим образом, активным является окно 3 и мы с помощью Win + H создаём горизонтальный контейнер:

Изображение выглядит как текст, диаграмма, снимок экрана, дизайн

Автоматически созданное описание

Новые окна будут открываться внутри нового контейнера:

Изображение выглядит как текст, диаграмма, снимок экрана, дизайн

Автоматически созданное описание

И команды Win + E, Win + W, Win + S будут менять компоновку внутреннего контейнера, а не внешнего:

Изображение выглядит как диаграмма, линия, Прямоугольник, дизайн

Автоматически созданное описание

Изображение выглядит как текст, диаграмма, снимок экрана, дизайн

Автоматически созданное описание

Изображение выглядит как текст, диаграмма, дизайн, снимок экрана

Автоматически созданное описание

Мы видим, что поменять компоновку второго контейнера легко, а как поменять компоновку первого? Для этого надо подняться на уровень выше. Есть два способа:

* использовать мышь или Win + стрелка, чтобы переключиться на окно, принадлежащее первому контейнеру;
* выделить *второй контейнер целиком* сочетанием Win + A:

Изображение выглядит как текст, диаграмма, линия, снимок экрана

Автоматически созданное описание

Жмём Win + E, чтобы превратить стековый контейнер в горизонтально разделённый:

Изображение выглядит как снимок экрана, диаграмма, Прямоугольник, линия

Автоматически созданное описание

Благодаря рекурсивным контейнерам в i3 можно создавать сколь угодно сложную мозайку вообще без напряга. Вот пример рабочей компоновки для большого монитора:

Изображение выглядит как диаграмма, снимок экрана, Прямоугольник, линия

Автоматически созданное описание

## Лекция №9. Создание неименованных каналов и приложений типа клиент-сервер в системе Linux

Неименованные каналы используются для взаимодействия между родственными процессами как в командной, так и в программной средах. Например, в среде интерпретатора shell можно выполнить командную строку:

cat myfile | wc

при выполнении которой стандартный вывод программы (утилиты) cat через канал передается на стандартный ввод программы wc. Для организации такого взаимодействия процессов интерпретатор shell создает неименованный канал, который затем наследуется его дочерними процессами, выполняющими команды cat и wc соответственно. Этот общедоступный канал и используется дочерними процессами для обмена данными.

**Взаимодействие родительского и дочернего процессов**

Для создания неименованного канала в программной среде используется системный вызов pipe():

int pipe(int \*fildes);

который возвращает два файловых дескриптора – fildes[1] для записи в канал и fildes[0] для чтения из канала. Эти дескрипторы наследуются потомком, созданным в результате выполнения системного вызова fork(), и в результате предок и потомок могут обмениваться данными через этот общий канал. Например, если предок закроет выход канала - дескриптор fildes[0], а потомок закроет вход канала - дескриптор fildes[0], то предок после этого сможет записывать данные на вход канала, а потомок – читать эти данные с выхода канала.

В программе, представленной в листинге 1, родительский процесс записывает в канал строку, а дочерний процесс читает ее. С помощью функции fdopen() файловые дескрипторы приводятся к типу FILE\*. Благодаря этому появляется возможность наряду с системными вызовами использовать высокоуровневые (библиотечные) функции ввода-вывода, такие как printf() и fgets().

**Листинг 1. (*pipe.c*) Общение с дочерним процессом посредством канала**

#include <stdlib.h>

#include <stdio.h>

#include <unistd.h>

/\* Запись указанного числа копий (count) сообщения (message) в поток (stream) с паузой между каждой операцией \*/

void writer (const char\* message, int count, FILE\* stream)

{

for (; count > 0; --count) {

/\* Запись сообщения в поток с немедленным "выталкиванием" из буфера\*/

fprintf (stream, "%s\n", message);

fflush (stream);

/\* Небольшая пауза. \*/

sleep (1);

}

}

/\* Чтение строк из потока, пока он не опустеет. \*/

void reader (FILE\* stream)

{

char buffer[1024];

/\* Чтение данных, пока не будет обнаружен конец потока.

Функция fgets() завершается, когда встречает символ

новой строки или признак конца файла. \*/

while (!feof (stream)

&& !ferror (stream)

&& fgets (buffer, sizeof (buffer), stream) != NULL)

fputs (buffer, stdout);

}

int main ()

{

int fds[2];

pid\_t pid;

/\* Создание канала. Дескрипторы входа и выхода канала помещаются

в массив fds. \*/

pipe (fds);

/\* Порождение дочернего процесса. \*/

pid = fork ();

if (pid == (pid\_t) 0) {

FILE\* stream;

/\* Это дочерний процесс. Закрываем копию входа канала. \*/

close (fds[1]);

/\* Приводим дескриптор выхода канала к типу FILE\*

и читаем данные из канала. \*/

stream = fdopen (fds[0], "r");

reader (stream);

close (fds[0]);

}

else {

/\* Это родительский процесс. \*/

FILE\* stream;

/\* Закрываем копию выхода канала. \*/

close (fds[0]);

/\* Приводим дескриптор выхода канала к типу FILE\*

и записываем данные в канал. \*/

stream = fdopen (fds[1], "w");

writer ("Hello, world.", 5, stream);

close (fds[1]);

}

return 0;

}

Отметим, что в функции writer() родительский процесс принудительно "выталкивает" буфер канала, вызывая функцию fflush(). Без этого строка могла бы "застрять" в буфере и отправиться в канал только после завершения родительского процесса.

**Перенаправление стандартных потоков**

Через неименованный канал легко реализовать перенаправление стандартных потоков ввода, вывода и ошибок. В этом случае используется системный вызов dup2(). Напомним, что системные вызовы dup() и dup2() позволяют создать еще один файловый дескриптор, который будет ссылаться на тот же файл. Например, вызов

dup2(fd, 100);

создает файловый дескриптор 100 как вторую ссылку на файл с дескриптором fd. Аналогично, чтобы связать стандартный поток ввода с файлом, имеющим дескриптор fd, выполняется вызов

dup2(fd, STDIN\_FILENO);

Символическая константа STDIN\_FILENO представляет дескриптор файла, соответствующий стандартному потоку ввода (значение этого дескриптора равно 0). Функция dup2() в этом случае закрывает входной поток, а потом открывает его перенаправленным на файл с дескриптором fd. Оба дескриптора (0 и fd) будут указывать на одну и ту же позицию в файле и иметь одинаковый набор флагов состояния, т.е. дескрипторы станут взаимозаменяемыми.

Программа, представленная в листинге 2, с помощью функции dup2() соединяет выход канала со входом команды sort(). Предварительно родительский процесс создает канал и порождает дочерний процесс для выполнения команды sort(). После этого родительский процесс записывает в канал различные строки, а дочерний процесс сортирует их и выдает на стандартный вывод.

**Листинг 2. (*dup2.c*) Перенаправление выхода канала на стандартный ввод**

#include <stdio.h>

#include <sys/types.h>

#include <sys/wait.h>

#include <unistd.h>

int main ()

{

int fds[2];

pid\_t pid;

/\* Создание канала. Дескрипторы входа и выхода канала

помещаются в массив fds. \*/

pipe (fds);

/\* Создание дочернего процесса. \*/

pid = fork ();

if (pid == (pid\_t) 0) {

/\* Это дочерний процесс. Закрываем копию дескриптора

входа канала. \*/

close (fds[1]);

/\* Соединяем выход канала со стандартным потоком ввода. \*/

dup2 (fds[0], STDIN\_FILENO);

/\* Загружаем в дочерний процесс программу sort. \*/

execlp ("sort", "sort", 0);

}

else {

/\* Это родительский процесс. \*/

FILE\* stream;

/\* Закрываем копию дескриптора выхода канала. \*/

close (fds[0]);

/\* Приводим дескриптор входа канала к типу FILE\* и

записываем данные в канал. \*/

stream = fdopen (fds[1], "w");

fprintf (stream, "This is a test.\n");

fprintf (stream, "Hello, world.\n");

fprintf (stream, "My dog has fleas.\n");

fprintf (stream, "This program is great.\n");

fprintf (stream, "One fish, two fish.\n");

fflush (stream);

close (fds[1]);

/\* Дожидаемся завершения дочернего процесса. \*/

waitpid (pid, NULL, 0);

}

return 0;

}

**Средства создания программных каналов типа "предок-потомок"**

Каналы часто используются для передачи данных программе, выполняющейся как подпроцесс (или приема данных от нее). Специально для этих целей предназначены функции popen() и pclose(), устраняющие необходимость в вызове функций pipe(), dup2(), exec() и fdopen().

Программа, приведенная в листинге 3, является более простым вариантом предыдущей программы.

**Листинг 3. (*popen.c*) Использование функций popen() и pclose()**

#include <stdio.h>

#include <unistd.h>

int main ()

{

FILE\* stream = popen ("sort", "w");

fprintf (stream, "This is a test.\n");

fprintf (stream, "Hello, world.\n");

fprintf (stream, "My dog has fleas.\n");

fprintf (stream, "This program is great.\n");

fprintf (stream, "One fish, two fish.\n");

return pclose (stream);

}

Функция popen() создает дочерний процесс, в котором выполняется команда sort(). Один этот вызов заменяет вызовы функций pipe(), fork(), dup2() и execlp(). Второй аргумент, "w", указывает на то, что текущий процесс хочет осуществлять запись в дочерний процесс. Функция popen() возвращает указатель на вход канала, а выход канала соединяется со стандартным входным потоком дочернего процесса.

Функция pclose() закрывает поток, указатель на который был возвращен функцией popen(), и дожидается завершения дочернего процесса.

**Создание приложений типа клиент-сервер**

Одним из наиболее распространенных подходов к созданию как системного, так и прикладного программного обеспечения является подход, в соответствии с которым программы делятся на два вида: программы-серверы, реализующие определенную функциональность, и программы-клиенты, обращающиеся к программам-серверам для обслуживания. Созданные в соответствии с моделью клиент-сервер приложения (программы) называются приложениями типа клиент-сервер.

Рассмотрим особенности создания приложения типа клиент-сервер на примере простейшего сервера файлов. Предположим, что программа-клиент (клиент в дальнейшем) считывает полное имя файла из стандартного потока ввода и записывает его в канал IPC. Сервер считывает это имя из канала IPC и производит попытку открытия указанного файла на чтение. Если попытка оказывается успешной, сервер считывает файл и записывает его в канал IPC. В противном случае сервер возвращает клиенту сообщение об ошибке. Клиент считывает данные из канала IPC и записывает их в стандартный поток вывода.

Программа, приведенная в листинге 4, использует два неименованных канала для организации взаимодействия между клиентом и сервером файлов.

**Листинг 4. (*mainpipe.c*) Функция main() для приложения типа клиент-сервер**

#include "unpipc.h"

int main(int argc, char \*\*argv)

{

int pipe1[2], pipe2[2];

pid\_t childpid;

pipe(pipe1); /\* создание двух каналов \*/

pipe(pipe2);

if ( (childpid = fork()) == 0) { /\* потомок \*/

close(pipe1[1]);

close(pipe2[0]);

server(pipe1[0], pipe2[1]);

exit(0);

}

/\* родитель \*/

close(pipe1[0]);

close(pipe2[1]);

client(pipe2[0], pipe1[1]);

waitpid(childpid, NULL, 0); /\* ожидание завершения потомка \*/

exit(0);

}

Функция main() создает два канала и дочерний процесс. Родительский процесс становится клиентом, а дочерний – сервером. Первый канал (канал pipe1) используется для передачи полного имени файла от клиента серверу, а второй – для передачи содержимого файла (или сообщения об ошибке) от сервера клиенту.

Поскольку неименованные каналы реализуются ядром операционной системы, то каждый передаваемый байт между клиентом и сервером пересекает интерфейс ядра дважды: при записи в канал и при считывании из него.

После создания каналов родительский процесс вызывает функцию client (листинг 5), а дочерний – функцию server(листинг 6).

Процесс-сервер (дочерний процесс) завершает свою работу первым, вызывая функцию exit() после завершения записи в канал. После этого он становится процессом-зомби, поскольку процесс-родитель не ожидает его завершения, а посланный родителю ядром сигнал SIGCHLD не обрабатывается и по умолчанию игнорируется. После этого функция client() родительского процесса завершает считывание данных и возвращает управление функции main(). Затем родительский процесс вызывает waitpid() для получения информации о завершении дочернего процесса, находящегося в состоянии зомби. Если родительский процесс не вызовет waitpid(), а просто завершит работу, сервер будет унаследован процессом init, которому будет послан еще один сигнал SIGCHLD.

**Листинг 5. (*mainpipe.c*) Функция client() для приложения типа клиент-сервер**

void client(int readfd, int writefd)

{

size\_t len;

ssize\_t n;

char buff[MAXLINE];

/\* получение полного имени файла \*/

fgets(buff, MAXLINE, stdin);

len = strlen(buff); /\* fgets() гарантирует нулевой байт в конце \*/

if (buff[len-1] == '\n')

len--; /\* удаление перевода строки из fgets() \*/

/\* Запись полного имени файла в канал IPC \*/

write(writefd, buff, len);

/\* считывание из канала IPC, вывод в stdout \*/

while ( (n = read(readfd, buff, MAXLINE)) > 0)

write(STDOUT\_FILENO, buff, n);

}

**Листинг 6. (*mainpipe.c*) Функция server() для приложения типа клиент-сервер**

void server(int readfd, int writefd)

{

int fd;

ssize\_t n;

char buff[MAXLINE+1];

/\* получение полного имени файла из канала IPC \*/

if ( (n = read(readfd, buff, MAXLINE)) == 0)

err\_quit("end-of-file while reading pathname");

buff[n] = '\0'; /\* полное имя завершается нулем \*/

if ( (fd = open(buff, O\_RDONLY)) < 0) {

/\* ошибка: сообщить клиенту \*/

snprintf(buff + n, sizeof(buff) - n, ": can't open, %s\n",

strerror(errno));

n = strlen(buff);

write(writefd, buff, n);

} else {

/\* файл успешно открыт и копируется в канал \*/

while ( (n = read(fd, buff, MAXLINE)) > 0)

write(writefd, buff, n);

close(fd);

}

}

Отметим, что в функции server() используется функция snprintf() вместо функции sprintf(), поскольку sprintf() не производит проверки переполнения приемного буфера. В отличие от нее snprintf() получает в качестве второго аргумента размер приемного буфера и впоследствии предотвращает его переполнение. Умышленное переполнение буфера программы, использующей sprintf(), в течение многих лет использовалось хакерами для взлома систем.

**Именованные каналы**

Операционная система UNIX в своей основе состоит из множества процессов, которые в процессе функционирования системы создаются, взаимодействуют между собой и завершаются. Взаимодействие процессов (обмен сообщениями, синхронизация, взаимное исключение при совместном использовании ресурсов) реализуется посредством различных механизмов межпроцессного взаимодействия (Inter-Process Communication, IPC). Простейшими средствами IPC, предназначенными для передачи данных между процессами, являются именованные и неименованные каналы.

Файл FIFO (First-In, First-Out – первым пришел, первым обслужен) – это канал, у которого есть имя в файловой системе. Любой процесс может открыть и закрыть такой файл. Процессы, находящиеся на противоположных концах канала, не обязаны быть родственными или каким-либо образом связанными процессами. FIFO-файлы называют именованными каналами.

FIFO-файл может быть создан как в командной, так и в программной средах. В среде интерпретатора shell FIFO-файл создается с помощью команды mkfifo. Путь к файлу указывается в командной строке, например:

mkfifo /tmp/fifo

ls –l /tmp/fifo

prw-rw-rw- 1 ovp ovp 0 Jan 16 14 : 04 /tmp/fifo

Первый символ в строке режима доступа (p) указывает на то, что файл имеет тип FIFO (именованный) канал.Теперь в одном терминальном окне можно осуществлять чтение из файла с помощью команды

cat < /tmp/fifo

а в другом окне можно выполнять запись в файл

cat > /tmp/fifo

В результате текст, введенный во втором окне, отобразится в первом окне. Канал закрывается нажатием клавиш <Ctrl+D> во втором окне. FIFO-файл удаляется как обычный файл, т.е. с помощью команды rm:

rm /tmp/fifo

FIFO-файл можно создать программным путем с помощью функции mkfifo(). Первым аргументом является полное имя файла, знание которого позволит двум неродственным процессам обратиться к одному и тому же FIFO. Второй аргумент задает права доступа к каналу со стороны его владельца, группы и остальных пользователей – каждой из этих категорий пользователей может быть разрешен доступ "r" и/или "w".

Функция mkfifo() действует как open(), вызванная с аргументом O\_CREAT | O\_EXCL. Это означает, что создается новый канал FIFO или возвращается ошибка EEXIST в случае, если канал с заданным полным именем уже существует. Если не требуется создавать новый канал, целесообразно использовать open() вместо mkfifo(). Для открытия существующего канала или создания нового в том случае, если его еще не существует, вызовите mkfifo(), проверьте, не возвращена ли ошибка EEXIST, и если такое случится, вызовите функцию open().

К FIFO-файлу можно обращаться как к обычному файлу. При организации межзадачного взаимодействия одна программа должна открыть файл для записи, а другая – для чтения. Над файлом можно выполнять как низкоуровневые (open(), write(), read(), close() и др.), так и высокоуровневые (fopen(), fprintf(), fscanf(), fclose() и др.) функции.

Например, на низком уровне запись блока данных в FIFO-файл осуществляется следующим образом:

int fd = open (fifo\_path, O\_WRONLY);

write (fd, data, data\_length);

close (fd);

А так выполняется чтение строки из FIFO-файла на высоком уровне:

FILE\* fifo = fopen (fifo\_ path, "r");

fscanf (fifo, "%s", buffer);

fclose (fifo);

У FIFO-файла одновременно может быть несколько читающих и записывающих программ. Входные потоки разбиваются на атомарные блоки, размер которых определяется константой PIPE\_BUF (4 Кбайт в Linux). Если несколько программ параллельно друг другу осуществляют запись в файл, их блоки будут чередоваться. То же самое относится к программам, одновременно читающим данные из файла. Нельзя открывать FIFO-файл на чтение и запись одновременно, поскольку именованные каналы могут быть только односторонними.

При записи в программный (неименованный) канал или канал FIFO вызовом write() данные всегда добавляются к уже имеющимся, а вызов read() считывает данные, помещенные в программный канал или канал FIFO первыми. При вызове функции lseek() для программного канала или FIFO будет возвращена ошибка ESPIPE.

*2.2. Создание приложений типа клиент-сервер*

Одним из наиболее распространенных подходов к созданию как системного, так и прикладного программного обеспечения является подход, в соответствии с которым программы делятся на два вида: программы-серверы, реализующие определенную функциональность, и программы-клиенты, обращающиеся к программам-серверам для обслуживания. Созданные в соответствии с данной моделью приложения (программы) называются приложениями типа клиент-сервер.

Рассмотрим особенности создания приложения типа клиент-сервер на примере простейшего сервера файлов. Предположим, что программа-клиент (клиент в дальнейшем) считывает полное имя файла из стандартного потока ввода и записывает его в канал IPC. Сервер считывает это имя из канала IPC и производит попытку открытия указанного файла на чтение. Если попытка оказывается успешной, сервер считывает файл и записывает его в канал IPC. В противном случае сервер возвращает клиенту сообщение об ошибке. Клиент считывает данные из канала IPC и записывает их в стандартный поток вывода.

Программа, приведенная в листинге 1, использует два именованных канала для организации взаимодействия между клиентом и сервером файлов.

**Листинг 1. (*pipe/mainfifo.c*) Функция main() для приложения типа клиент-сервер**

#define FIFO1 "/tmp/fifo.1"

#define FIFO2 "/tmp/fifo.2"

#define FILE\_MODE (S\_IRUSR | S\_IWUSR | S\_IRGRP | S\_IROTH)

void client(int, int), server(int, int);

int main(int argc, char \*\*argv)

{

int readfd, writefd;

pid\_t childpid;

/\* создание двух fifo, если уже существуют - OK \*/

if ((mkfifo(FIFO1, FILE\_MODE) < 0) && (errno != EEXIST))

err\_sys("can't create %s", FIFO1);

if ((mkfifo(FIFO2, FILE\_MODE) < 0) && (errno != EEXIST)) {

unlink(FIFO1);

err\_sys("can't create %s", FIFO2);

}

if ( (childpid = fork()) == 0) { /\* потомок \*/

readfd = open(FIFO1, O\_RDONLY, 0);

writefd = open(FIFO2, O\_WRONLY, 0);

server(readfd, writefd);

exit(0);

}

/\* родитель \*/

writefd = open(FIFO1, O\_WRONLY, 0);

readfd = open(FIFO2, O\_RDONLY, 0);

client(readfd, writefd);

waitpid(childpid, NULL, 0); /\* ожидание завершения потомка \*/

close(readfd);

close(writefd);

unlink(FIFO1);

unlink(FIFO2);

exit(0);

}

Функция main() создает два канала FIFO и дочерний процесс. Родительский процесс вызывает функцию client(), а дочерний – функцию server(). Первый канал (канал FIFO1) используется для передачи полного имени файла от клиента серверу, а второй – для передачи содержимого файла (или сообщения об ошибке) от сервера клиенту.

Отметим, что использование в данной программе именованных каналов вместо неименованных определило ряд отличий в программе:

* Для создания и открытия программного канала требуется только один вызов – pipe(). Для создания и открытия FIFO требуется вызов mkfifo и последующий вызов open().
* Программный канал автоматически исчезает после того, как будет закрыт последним использующим его процессом. Канал FIFO удаляется из файловой системы только после вызова unlink().

После создания каналов родительский процесс вызывает функцию client (листинг 2), а дочерний – функцию server(листинг 3).

**Листинг 2. (*pipe/client.c*) Функция client() для приложения типа клиент-сервер**

void client(int readfd, int writefd)

{

size\_t len;

ssize\_t n;

char buff[MAXLINE];

/\* получение полного имени файла \*/

fgets(buff, MAXLINE, stdin);

len = strlen(buff); /\* fgets() гарантирует нулевой байт в конце \*/

if (buff[len-1] == '\n')

len--; /\* удаление перевода строки из fgets() \*/

/\* Запись полного имени файла в канал IPC \*/

write(writefd, buff, len);

/\* считывание из канала IPC, вывод в stdout \*/

while ( (n = read(readfd, buff, MAXLINE)) > 0)

write(STDOUT\_FILENO, buff, n);

}

**Листинг 3. (*pipe/server.c*) Функция server() для приложения типа клиент-сервер**

void server(int readfd, int writefd)

{

int fd;

ssize\_t n;

char buff[MAXLINE+1];

/\* получение полного имени файла из канала IPC \*/

if ( (n = read(readfd, buff, MAXLINE)) == 0)

err\_quit("end-of-file while reading pathname");

buff[n] = '\0'; /\* полное имя завершается нулем \*/

if ( (fd = open(buff, O\_RDONLY)) < 0) {

/\* ошибка: сообщить клиенту \*/

snprintf(buff + n, sizeof(buff) - n, ": can't open, %s\n",

strerror(errno));

n = strlen(buff);

write(writefd, buff, n);

} else {

/\* файл успешно открыт и копируется в канал \*/

while ( (n = read(fd, buff, MAXLINE)) > 0)

write(writefd, buff, n);

close(fd);

}

}

Отметим, что в функции server() используется функция snprintf() вместо функции sprintf(), поскольку sprintf() не производит проверки переполнения приемного буфера. В отличие от нее snprintf() получает в качестве второго аргумента размер приемного буфера и впоследствии предотвращает его переполнение. Умышленное переполнение буфера программы, использующей sprintf(), в течение многих лет использовалось хакерами для взлома систем.

*2.3. Неродственные клиент и сервер*

В листингах 1 – 3 клиент и сервер являются родственными процессами. Переделаем эту программу так, чтобы родство между ними отсутствовало. В листинге 4 приведен текст программы-сервера. Текст практически идентичен той части программы из листинга 1, которая относилась к серверу.

**Листинг 4. (*pipe/server\_main.c*) Функция main() независимого сервера**

#include "fifo.h"

void server(int, int);

int main(int argc, char \*\*argv)

{

int readfd, writefd;

/\* создание двух FIFO, ОК, если они существуют \*/

if ((mkfifo(FIFO1, FILE\_MODE) < 0) && (errno != EEXIST))

err\_sys("can't create %s", FIFO1);

if ((mkfifo(FIFO2, FILE\_MODE) < 0) && (errno != EEXIST)) {

unlink(FIFO1);

err\_sys("can't create %s", FIFO2);

}

readfd = open(FIFO1, O\_RDONLY, 0);

writefd = open(FIFO2, O\_WRONLY, 0);

server(readfd, writefd);

exit(0);

}

Содержимое заголовка fifo.h приведено в листинге 5. Этот файл определяет имена двух FIFO, которые должны быть известны как клиенту, так и серверу.

**Листинг 5. (*pipe/fifo.h*) Заголовочный файл для клиента и сервера**

#define FIFO1 "/tmp/fifo.1"

#define FIFO2 "/tmp/fifo.2"

#define FILE\_MODE (S\_IRUSR | S\_IWUSR | S\_IRGRP | S\_IROTH)

В листинге 6 приведен текст программы-клиента, которая не слишком отличается от части программы из листинга 1, относящейся к клиенту. Отметим, что клиент, а не сервер удаляет канал FIFO по завершении работы, потому что последние операции с этим каналом выполняются им.

**Листинг 6. (*pipe/client\_main.c*) Функция main() независимого клиента**

#include "fifo.h"

void client(int, int);

int main(int argc, char \*\*argv)

{

int readfd, writefd;

writefd = open(FIFO1, O\_WRONLY, 0);

readfd = open(FIFO2, O\_RDONLY, 0);

client(readfd, writefd);

close(readfd);

close(writefd);

unlink(FIFO1);

unlink(FIFO2);

exit(0);

}

Для запуска клиента и сервера запустите сервер в фоновом режиме:

server\_fifo &

а затем запустите клиент. Можно было сделать и по другому: запускать только программу-клиент, которая запускала бы сервер с помощью fork и exec. Клиент мог бы передавать серверу имена FIFO в качестве аргументов командной строки в команде exec, вместо того чтобы обе программы считывали их из заголовка. Но в этом случае сервер являлся бы дочерним процессом и проще было бы обойтись программным каналом.

## Лекция 10. Создание простейшей командной оболочки ядра Linux на базе процессов

**Функция main простейшей оболочки ядра операционной системы Linux**

В листинге 1 представлена функция main программы, которая может быть использована в качестве простейшей оболочки ядра операционной системы Linux.

**Листинг 1. (*myshell.c*) Функция main() простейшей оболочки ядра**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <unistd.h>

#include <sys/stat.h>

#include <sys/ipc.h>

#include <sys/types.h>

#include <sys/time.h>

#include <time.h>

#include <errno.h>

#include <fcntl.h>

#include <limits.h>

#include <signal.h>

#include <sys/wait.h>

#include <sys/param.h>

#include <sys/resource.h>

#include <syslog.h>

#include <semaphore.h>

#include <sys/mman.h>

// Размерности массивов

#define MAXLINE 4096

#define MAXBUFF 256

#define MAXQUEUE 10

#define MAXPATH 120

#define MAXLEN 256

/\*Разрешения по умолчанию для вновь создаваемых файлов\*/

#define FILE\_MODE (S\_IRUSR | S\_IWUSR | S\_IRGRP | S\_IROTH | O\_NDELAY)

char\* arg[MAXLEN]; // массив аргументов команды

char\* s\_path[MAXPATH]; // массив команд с путями из PATH

char\* queue[MAXQUEUE]; // массив для хранения неинтерактивных команд

char\* queue0[MAXQUEUE]; // массив для хранения интерактивных команд

int errno; // код ошибки

int outfl; // флаг вида перенаправления вывода (замещение/дополнение)

void preparse(char\* cmd);

int str\_parse(char \*cmd);

int input(char \*name);

int output(char \*name);

void cmdExec(char \*\*env);

int execat(char \*search\_path, char \*path);

int main(int argc, char \*\*argv, char \*\*env)

{

int in\_out, p;

char cmd[MAXLEN]; //строка, считанная с терминала

int fd\_1, fd\_0; //для сохранения дискрипторов потоков стандартного ввода/вывода

setbuf(stdout, NULL); //отмена буфферизации

for(;;) {

write(1, "$> ", 3); // вывод приглашения к вводу командной строки

gets(cmd); // чтение командной строки в массив cmd

preparse(cmd); // разбиение командной строки cmd на отдельные команды

p=0;

// цикл по интерактивным командам

while (queue0[p]!=NULL) {

//сохранение текущих дискрипторов потоков стандартного ввода/вывода

fd\_1=dup(1); fd\_0=dup(0);

// запись очередной команды в массив cmd

snprintf(cmd, sizeof(cmd), "%s", queue0[p]);

in\_out = str\_parse(cmd); //разбор строки команды

p++;

// Выполнение встроенных в оболочку команд

if (!strcmp(arg[0], "showenv")) {

int i;

printf("\nEnvironment:\n");

for (i=0; env[i]; i++)

printf("\n%s", env[i]);

printf("\n");

}

if (!strcmp(arg[0], "echo"))

prinf("\n%s\n", arg[1]);

// Выполнение команд – исполнимых файлов

cmdExec(env);

//восстановление дискрипторов stdout/stdin

dup2(fd\_1, 1);

close(fd\_1);

dup2(fd\_0, 0);

close(fd\_0);

}

}

}

Функция main организует цикл для ввода и выполнения командных строк. Для каждой командной строки выдается промптер (приглашение к вводу), состоящее из трех символов. Пример командной строки:

date;pwd;ls;ls -al > f1&;echo "Метаданные" >> f1&;/bin/ls -l add.c >> f1&;cat add.c

Данная командная строка содержит четыре интерактивных и три неинтерактивных команды. После чтения и записи этой строки в массив cmd функции main, содержимое этого массива передается (копируется через аргумент функции) в массив cmd другой функции - функции preparse.

**Разбиение командной строки на отдельные команды**

Первый этап обработки командной строки, который выполняется функцией preparse (листинг 2), - это разбиение командной строки на отдельные команды и запись этих команд в массивы интерактивных (массив queue0) и неинтерактивных (массив queue) команд. Для приведенного выше примера командной строки будут сформированы следующие два массива:

|  |  |
| --- | --- |
| **Массив queue0** | **Массив queue** |
| date | ls -al > f1& |
| pwd | echo "Метаданные" >> f1& |
| ls | /bin/ls -l a.c >> f1& |
| cat add.c |  |
|  |  |

**Листинг 2. (*myshell.c*) Функция preparse() простейшей оболочки ядра**

void preparse(char\* cmd)

{

int i=0, j=0, k=0, cntr=0, cntr0=0;

char tmp[80];

char\* ptmp = tmp;

for (k=0; k<=79; k++) tmp[k]='\0'; // очистка массива tmp

// Очистка массивов команд

for (i=0; i<MAXQUEUE; i++) free(queue0[i]), queue0[i]=NULL; // очистка массива queue0

for (i=0; i<MAXQUEUE; i++) free(queue[i]), queue[i] = NULL; // очистка массива queue

i=0;

strcat(cmd, ";");

// разбиение командной строки cmd на отдельные команды

do {

if ((\*(cmd+i)!=';')&&(\*(cmd+i)!='\0')) {

\*(tmp+j)=\*(cmd+i); j++; //запись в tmp очередного символа

}

else { // сохранить очередную команду

if (strchr(ptmp, '&')) {

queue[cntr]=strdup(tmp); // сохранение неинтерактивной команды

cntr++;

}

else { // сохранение интерактивной команды

queue0[cntr0]=strdup(tmp);

cntr0++;

}

k=0; for(k=0; k<=79; k++) { tmp[k]='\0'; } // очистка массива tmp

j = 0;

}

i++;

}

while(\*(cmd+i)!='\0');

return;

}

При выполнении функции preparse массив tmp используется как буфер, в который посимвольно переписывается очередная команда. Символ ; является символом разделения команд и, соответственно, при его обнаружении завершится цикл записи символов в массив tmp:

if ((\*(cmd+i)!=';')&&(\*(cmd+i)!='\0')) {

\*(tmp+j)=\*(cmd+i); j++; //запись в tmp очередного символа

}

В данном фрагменте используются указатели и адресная арифметика языка Си. Этот фрагмент можно переписать с использованием массивов с индексами следующим образом:

if ((cmd[i])!=';')&&(cmd[i]!='\0')) {

tmp[j]= cmd[i]; j++; //запись в tmp очередного символа

}

Цикл do ..... while (\*(cmd+i)!='\0'); завершится, когда будет обнаружен символ конца командной строки (символ \0).

Отметим, что для сохранения очередной команды функция strdup дублирует строку, на которую указывает аргумент. Память под дубликат строки выделяется из кучи с помощью функции malloc и по окончанию работы с дубликатом должна быть очищена с помощью функции free.

**Синтаксический анализ строк каждой команды**

Данная версия оболочки ядра предназначена для выполнения только интерактивных команд. Другими словами, оболочка не будет выполнять команды из массива queue. Чтобы усложнить синтаксический анализ команд сделаем все команды примера командной строки интерактивными:

date;pwd;ls;ls -al > f1;echo "Метаданные" >> f1; /bin/ls -l add.c >> f1;cat add.c

После выполнения функции preparse эта строка будет преобразована в следующие массивы:

|  |  |
| --- | --- |
| **Массив queue0** | **Массив queue** |
| date |  |
| pwd |  |
| ls |  |
| ls -al > f1 |  |
| echo "Метаданные" >> f1 |  |
| /bin/ls -l add.c >> f1 |  |
| cat add.c |  |

В функции main организован цикл while (queue0[p]!=NULL) { ..... }, в котором каждая очередная интерактивная команда вначале переписывается в массив cmd, который затем передается в функцию str\_parse (листинг 3) для разбиения ее на строки аргументов и записи их в массив arg. Отметим, что в строке arg[0] будет находиться имя команды, включая путь или без него.

Пример массива arg, полученного для предпоследней команды из массива queue0:

|  |  |
| --- | --- |
| **Массив arg** | **Элементы массива arg** |
| /bin/ls | arg[0] |
| -l | arg[1] |
| add.c | arg[2] |
| >> | arg[3] |
| f1 | arg[4] |
|  |  |

**Листинг 3. (*myshell.c*) Функция str\_parse () простейшей оболочки ядра**

/\*Разбор строки команды. Символы перенаправления ввода/вывода воспринимаются как

отдельные аргументы - пишутся через пробел. \*/

int str\_parse(char \*cmd)

{

char tmp[80];

int i=0,j=0,k,CNTR\_arg=0;

char \*name;

int in\_out = -1; //0 - поток перенаправлен

// Очистка массива-буфера для очередного аргумента

k=0; for (k=0; k<=79; k++) tmp[k]='\0'; // очистка массива tmp

// Очистка массива аргументов

for (i=0; i<MAXLEN; i++) free(arg[i]), arg[i] = NULL; // очистка массива arg

strcat(cmd, " "); //добавление пробела в конце команды

i=0;

// разбиение команды на аргументы

do {

if ((\*(cmd+i)!=' ')&&(\*(cmd+i)!='\0')) {

\*(tmp+j)=\*(cmd+i); j++; //запись в tmp очередного символа

}

else { // сохраняем очередной аргумент

arg[CNTR\_arg]=strdup(tmp);

CNTR\_arg++;

for(k=0; k<=79; k++) { tmp[k]='\0'; } // очистка массива tmp

j = 0;

}

i++;

}

while(\*(cmd+i)!='\0');

// Поиск символов перенаправления, выборка имени файла и

// модификация массива аркументов

for (i=0; arg[i] != NULL; i++) {

if ((strcmp(arg[i], ">")) ==0) {

arg[i]=NULL; name = arg[i+1];

arg[i+1]=NULL; in\_out = 0; outfl=0; output(name); break;}

else if ((strcmp(arg[i], ">>")) ==0) {

arg[i]=NULL;name = arg[i+1];

arg[i+1]=NULL; in\_out = 0; outfl=1; output(name); break;}

}

for (i=0; arg[i] != NULL; i++) {

if ((strcmp(arg[i], "<")) ==0) {

arg[i]=NULL; name = arg[i+1];

arg[i+1]=NULL; in\_out = 0; outfl=0; input(name); break;}

}

return in\_out;

}

При выполнении функции str\_parse массив tmp используется как буфер, в который посимвольно переписывается очередной аргумент команды:

if ((\*(cmd+i)!=' ')&&(\*(cmd+i)!='\0')) {

\*(tmp+j)=\*(cmd+i); j++; //запись в tmp очередного символа

}

Символы каждой опции считываются из cmd в tmp, а затем в массив arg. Символ пробела является символом разделения аргументов команды и, соответственно, при его обнаружении в массиве cmd завершится очередной цикл записи символов в массив tmp и будет выполнена запись массива tmp в очередной элемент массива arg.

Цикл do ..... while (\*(cmd+i)!='\0'); завершится, когда будет обнаружен символ конца строки команды (символ \0).

После завершения разбора строки команды и формирования массива arg производится просмотр аргументов команды для поиска в них символов перенаправления ввода-вывода. Тип найденного перенаправления сохраняется в переменной outfl (0 для перенаправления выходного потока, 1 - для входного потока), а имя файла потока переписывается в переменную name, значение которой передается в функции input или output (листинг 4) для соответствующей замены файлов стандартного потока ввода или вывода. После этого аргументы, содержащие символы перенаправления и имя потока, очищаются. Кроме этого, записывается 0 в переменную in\_out, значение которой будет значением, возвращаемым функцией str\_parse. Отметим, что при отсутствии перенаправления ввода-вывода в команде в переменной in\_out будет значение -1.

**Листинг 4. (*myshell.c*) Функции input () и output () простейшей оболочки ядра**

//Перенаправление ввода

int input(char \*name)

{

int fd;

if ((fd = open(name, O\_RDONLY)) < 0) return 0;

dup2(fd, 0); close(fd); return 1;

}

//Перенаправление вывода

int output(char \*name)

{

int fd;

if ((fd = open(name, O\_WRONLY)) <0) {

if (errno == ENOENT) //если файла нет, создадим

if ((fd = creat(name, 0644)) <0) return 0; // 0 - не успешно

if (errno == EACCES) { //права доступа?

write(2, "Shell: Доступ запрещен\n",25); return 0; }

//не имеем права писать в файл

}

// ">> file" указатель на конец файла

else if (outfl == 1) lseek(fd, 1, SEEK\_END);

if (outfl == 0) {

//"> file" пытаемся опустошить файл

if ((fd = creat(name, 0644)) <0) return 0; }

//перенаправляем вывод

dup2(fd, 1);

close(fd); return 1; //1 = успешно

}

Функция input заменяет стандартный поток ввода (поток stdin) с файловым дескриптором 0 на поток из файла, имя которого сохранено в переменной name.

Функция output заменяет стандартный поток вывода (поток stdout) с файловым дескриптором 1 на поток в файл, имя которого сохранено в переменной name.

**Выполнение интерактивной команды**

После завершения подготовки массива аргументов текущей команды в массиве arg, функция main вызывает функцию cmdExec (листинг 5) для выполнения интерактивной команды.

**Листинг 5. (*myshell.c*) Функция cmdExec () простейшей оболочки ядра**

void cmdExec(char \*\*env)

{

pid\_t pid;

int trys, y=0;

char \*search\_path; // Массив команд на базе путей из PATH

if (!strchr(arg[0], '/')) {

if ((search\_path = getenv("PATH")) == NULL)

search\_path = "/bin:/usr/bin:/usr/local/bin";

// Подготовка команд, выполняемых через PATH

trys = execat(search\_path, arg[0]);

}

// Порождаем потомка для выполнения команды

if ((pid = fork()) < 0) {

write(2, "Shell: cannot fork\n", 20); exit(1); }

else if(pid ==0) { // Работает потомок

if (strchr(arg[0], '/')) // Команда содержит путь к исполнимому файлу

execve(arg[0], arg, env); // Выполнение команды без PATH

else {

retry:

execve(s\_path[trys], arg, env);

if (trys != 0) { trys--; goto retry; }

}

write(2, "Shell: cannot execute\n", 23);

exit(1);

}

else

waitpid(pid, NULL, 0); //ожидание завершения потомка

return;

}

Если в arg[0] хранится только имя команды, местонахождение исполнимого файла этой команды будет найдено посредством перебора всех путей, которые находятся в переменной PATH. Содержимое переменной PATH и имя команды передаются в функцию execat (листинг 6), которая выполняет разбор содержимого переменной PATH и для каждого пути из PATH добавляется (через прямой слэш) имя команды. Полученный набор возможных полных (с путями) имен команды будет сохранен в массиве s\_path, а индекс последней команды в этом массиве - в переменной cnt\_path.

**Листинг 6. (*myshell.c*) Функция execat () простейшей оболочки ядра**

int execat(char \*search\_path, char \*path)

{

char tmp[MAXLINE], p\_tmp[MAXLEN];

char\* pp\_tmp = p\_tmp;

int point=0, j=0, cnt\_path=0,k,i;

for (i=0; i<MAXPATH; i++) free(s\_path[i]), s\_path[i]= NULL; // очистка массива s\_path

snprintf(tmp, sizeof(tmp), "%s", search\_path);

do

{

if((\*(tmp+point)!=':')&&(\*(tmp+point)!='\0'))

{

\*(p\_tmp+j)=\*(tmp+point); j++;

}

else

{

strcat(pp\_tmp,"/"); strcat(pp\_tmp, path);

s\_path[cnt\_path] = strdup(pp\_tmp);

cnt\_path++;

k=0; for(k=0;k<=MAXLEN;k++) p\_tmp[k]='\0'; k=0;

j=0;

}

point++;

}

while(\*(tmp+point)!='\0');

strcat(pp\_tmp,"/"); strcat(pp\_tmp, path);

s\_path[cnt\_path] = strdup(p\_tmp);

return cnt\_path;

}

**Контрольный пример для проверки работы оболочки**

На рис. 1 приведен контрольный пример для проверки работы разработанной оболочки ядра операционной системы Linux.

Изображение выглядит как текст, снимок экрана, программное обеспечение, Шрифт

Автоматически созданное описание

Рис. 1. Результаты компиляции и выполнения оболочки ядра

## Лекция №11. Семафоры стандарта Posix для синхронизации процессов и потоков в системе Linux

**Семафоры Posix и взаимные исключения**

Приложения Linux обычно имеют многомодульную структуру типа клиент-сервер. Каждый модуль приложения выполняет функции програм­много клиента и/или сервера и реализуется посредством процесса или программного потока. Эти модули взаимодействуют между собой посредством объектов межпроцессного или межпотокового взаимодействия, называемых в дальнейшем объектами IPC (IPC – Interprocess Communication).

IPC включает в себя следующие виды объектов:

* объекты для передачи сообщений (каналы pipe и fifo, очереди сообщений);
* объекты для синхронизации (взаимные исключения, условные переменные, блокировки чтения-записи, семафоры);
* разделяемая память (неименованная и именованная).

Очереди сообщений, семафоры, разделяемая память различаются по своей реализации и функциональным возможностям на объекты IPC стандарта System V и объекты IPC стандарта Posix.

Семафоры представляют собой простейшее средство синхронизации процессов и потоков. Они подразделяются на три типа:

* именованные семафоры Posix;
* размещаемые в разделяемой памяти (неименованные) семафоры Posix;
* семафоры System V, обслуживаемые ядром.

Все три типа семафоров могут использоваться для синхронизации как отдельных процессов, так и потоков одного процесса. Семафоры System V появились задолго до появления семафоров Posix, они являются громоздкими и сложными. В данной работе семафоры System V рассматриваться не будут.

Для работы с семафорами Posix могут применяться следующие три операции.

1. Создание семафора. Необходимо указать начальное значение (value) семафора – обычно это число ресурсов, выделением которых процессам и потокам будет управлять семафор. Если ресурс один, то семафор называется бинарным, а в качестве начального значения указывается 1 или 0.
2. Ожидание освобождения ресурса (wait). При этом производится проверка значения семафора и процесс блокируется, если значение оказывается равным 0. Если же в момент выдачи запроса wait семафор открыт (value положительное), значение семафора уменьшается на 1, т.е. процесс или поток получает соответствующий ресурс. У этой операции есть несколько общеупотребительных имен. Изначально она называлась P (P-операция), от голландского proben (проверка, попытка), - это название было введено Эдгером Дейкстрой. Используются также и термины down (поскольку значение семафора уменьшается) и lock (блокировка), но мы будем следовать стандарту Posix и говорить об ожидании (wait).
3. Установка значения семафора (post). Значение семафора увеличивается (ресурс освобождается), и если в системе имеются процессы или потоки, ожидающие освобождения ресурса, один из них может быть пробужден (разблокирован) для завершения выполнения операции wait. При этом значение семафора уменьшится на 1 и семафор опять закроется. Для этой операции также имеется несколько общеупотребительных терминов. Изначально она называлась V (V-операция), от голландского verhogen (увеличивать). Называют ее также up (значение семафора увеличивается), unlock и signal. Мы, следуя стандарту Posix, называем эту операцию post.

Эти операции могут использоваться как для бинарных, так и многозначных семафоров, называемых также семафорами-счетчиками. В системной реализации между этими семафорами нет никакой разницы.

Бинарный семафор может использоваться вместо IPC, называемого взаимным исключением (mutex). Взаимные исключения иногда называют исключающим семафором, что не совсем корректно, поскольку семафоры имеют больше возможностей: взаимное исключение должно быть разблокировано именно тем потоком (или процессом), который его заблокировал, в то время как увеличение значения семафора может быть выполнено другим потоком или процессом.

В листинге 1 приведен пример (на псевдокоде) для сравнения этих IPC Posix.

**Листинг 1.** Сравнение бинарных семафоров и взаимных исключений.

|  |  |
| --- | --- |
| инициализация взаимного исключения;  pthread\_mutex\_lock(&mutex);  критическая область  pthread\_mutex\_unlock(&mutex); | инициализация семафора единицей;  sem\_wait(&sem);  критическая область  sem\_post(&sem); |

Можно привести пример использования бинарных семафоров. В листинге 2 приведен фрагмент программы на псевдокоде с двумя бинарными семафорами для решения упрощенной версии задачи "производитель – потребитель", а на рис. 1 представлена сама модель, состоящая из двух процессов, буфера и двух семафоров - put и get.

Семафор put ограничивает возможность помещения более одного объекта в общий буфер, а семафор get управляет потребителем при считывании объекта из буфера (при отсутствии объекта в буфере вызов sem\_wait(get) заблокирует выполнение потребителя). Поскольку семафор put инициализируется единицей, только производитель вначале имеет возможность работать с буфером и поместить в него объект. После этого семафор put остается закрытым, а открывается семафор get для возможности считывания этого объекта потребителем. После считывания потребитель открывает семафор put для производителя и начинается новый цикл передачи данных от производителя к потребителю.

Изображение выглядит как текст, диаграмма, снимок экрана, линия

Автоматически созданное описание

Рис. 1. Использование семафоров в модели "производитель – потребитель"

**Листинг 2.** Псевдокод для задачи производитель – потребитель.

|  |  |
| --- | --- |
| **Производитель (producer)**  Инициализация семафора get значением 0;  Инициализация семафора put значением 1;  for ( ; ; ) {  Подготовка данных  sem\_wait (put);  Запись в буфер  sem\_post(get);  } | **Потребитель (consumer)**  for ( ; ; ) {  sem\_wait (get);  Чтение из буфера  sem\_post(put);  Печать буфера  } |

**Именованные семафоры Posix**

Стандартом Posix описано два типа семафоров: именованные (named) и размещаемые в памяти (memory-based или unnamed). На рис. 1 сравниваются функции, используемые обоими типами семафоров. Четыре из этих функций являются общими.

|  |  |
| --- | --- |
| **Именованный семафор**  sem\_open ()  sem\_wait ()  sem\_trywait ()  sem\_post ()  sem\_getvalue ()  sem\_close ()  sem\_unlink () | **Неименованный семафор**  sem\_init ()  sem\_destroy () |

Рис. 1. Системные вызовы для семафоров Posix

Функция sem\_open() создает новый именованный семафор или открывает существующий:

#include <semaphore.h>

sem\_t \*sem\_open(const char \**name*, int *oflag*, … /\* mode\_t *mode*, unsigned int *value* \*/ );

/\* Возвращает указатель на семафор или SEM\_FAILED в случае ошибки \*/

Имя семафора *name* должно иметь вид /semname, причем строка semname не должна содержать символа / и иметь длину не более 251 символа. Аргумент *oflag* может принимать значения 0, O\_CREAT, O\_CREAT | O\_EXCL. Если указано значение O\_CREAT, третий и четвертый аргументы функции являются обязательными. Аргумент *mode* указывает биты разрешений доступа (по чтению и/или записи), а *value* указывает начальное значение семафора.

Возвращаемое значение представляет собой указатель на тип sem\_t. Этот указатель впоследствии передается в качестве аргумента функциям sem\_close, sem\_wait, sem\_trywait, sem\_post и sem\_getvalue. Например, открыв семафор с помощью sem\_open(), можно потом закрыть его, вызвав sem\_close:

#include <semaphore.h>

int sem\_close (sem\_t \*sem);

/\* Возвращает 0 в случае успешного завершения, -1 - в случае ошибки \*/

Операция закрытия выполняется автоматически при завершении процесса (добровольного или по сигналу) для всех семафоров, которые были им открыты. Закрытие семафора не удаляет его из системы. Именованные семафоры Posix обладают по меньшей мере живучестью ядра. Значение семафора сохраняется, даже если ни один процесс не держит его открытым.

Именованный семафор удаляется из системы вызовом sem\_unlink:

#include <semaphore.h>

int sem\_unlink (const char \*name );

/\* Возвращает 0 в случае успешного завершения, -1 - в случае ошибки \*/

**Командный интерфейс для именованных семафоров Posix**

Различные виды объектов IPC обладают различной живучестью. Канал (именованный или неименованный) обладает живучестью процесса (process-persistent) и существуют до тех пор, пока не будет закрыт последним процессом, в котором он был открыт. Семафор System V обладает живучестью ядра (kernel-persistent) и существует до перезагрузки ядра или до явного удаления семафора. Живучесть именованного семафора Posix, реализованного через отображаемый файл, определятся файловой системой (filesystem-persistent), т.е. он будет существовать до тех пор, пока не будет удален явно.

Поскольку именованные семафоры Posix обладают по крайней мере живучестью ядра, мы может реализовать набор команд для работы с семафорами из командной строки или даже из командных процедур (скриптов).

В листинге 3 приведен текст программы, создающей именованный семафор. При вызове программы можно указать опцию (параметр) -e, обеспечивающую исключающее создание (если семафор уже существует, будет выведено сообщение об ошибке), а опция -i с числовым аргументом позволяет задать начальное значение семафора, отличное от 1.

**Листинг 3. (*semcreate.c*) Создание именованного семафора**

#include <semaphore.h>

#include <fcntl.h>

#include <getopt.h>

#include <stdio.h>

#include <stdlib.h>

#include <sys/types.h>

#include <sys/stat.h>

#define FILE\_MODE (S\_IRUSR | S\_IWUSR | S\_IRGRP | S\_IROTH)

#define err\_quit(x) {fprintf(stderr, x); exit(EXIT\_FAILURE);}

int main(int argc, char \*\*argv)

{

int c, flags;

sem\_t \*sem;

unsigned int value;

flags = O\_RDWR | O\_CREAT;

value = 1;

while ( (c = getopt(argc, argv, "ei:")) != -1) {

switch (c) {

case 'e':

flags |= O\_EXCL;

break;

case 'i':

value = atoi(optarg);

break;

}

}

if (optind != argc - 1)

err\_quit("usage: semcreate [ -e ] [ -i initialvalue ] <name>");

sem = sem\_open(argv[optind], flags, FILE\_MODE, value);

sem\_close(sem);

exit(0);

}

Здесь библиотечная функция getopt() выполняет синтаксический анализ аргументов командной строки. Ее объявление находится в файле <getopt.h>. При вызове getopt() ей передается командная строка и строка с описанием возможных коротких опций (каждая из которых представлена одной буквой). Если опция предполагает наличие аргумента, после нее ставится двоеточие. В программе semcreate.c эта строка опций имеет вид "ei:". Это говорит о том, что программа поддерживает опции –e, -i, причем вторая опция сопровождается аргументом (начальным значением семафора).

Обычно функция getopt() вызывается в цикле для обработки всех опций командной строки. Выбор конкретной опции осуществляется посредством конструкции switch. При обработке опции, имеющей аргумент, в глобальную переменную optarg помещается указатель на строку с содержимым аргумента. Когда функция getopt() завершает анализ опций, в глобальную переменную optind записывается индекс того элемента массива argv, в котором содержится первый аргумент, не являющийся опцией (имя семафора в нашем случае).

В листингах 4, 5 приведены программа удаления именованного семафора и программа, которая открывает именованный семафор, получает и выводит его текущее значение.

**Листинг 4. (*semunlink.c*) Удаление именованного семафора**

int main(int argc, char \*\*argv)

{

if (argc != 2)

err\_quit("usage: semunlink <name>");

sem\_unlink(argv[1]);

exit(0);

}

**Листинг 5. (*semgetvalue.c*) Получение и вывод значения семафора**

int main(int argc, char \*\*argv)

{

sem\_t \*sem;

int val;

if (argc != 2)

err\_quit("usage: semgetvalue <name>");

sem = sem\_open(argv[1], 0);

sem\_getvalue(sem, &val);

printf("value = %d\n", val);

exit(0);

}

Отметим, что в программе semgetvalue.c мы открываем семафор, созданный другой программой. Поэтому вторым аргументом в sem\_open() является 0.

Программа в листинге 6 открывает именованный семафор, вызывает sem\_wait() (которая приостанавливает выполнение процесса, если значение семафора меньше либо равно нулю, а при положительном значении семафора уменьшает его на 1), получает и выводит значение семафора, а затем останавливает свою работу навсегда при вызове pause().

**Листинг 6. (*semwait.c*) Ожидание открытия семафора и вывод его нового значения**

int main(int argc, char \*\*argv)

{

sem\_t \*sem;

int val;

if (argc != 2)

err\_quit("usage: semwait <name>");

sem = sem\_open(argv[1], 0);

sem\_wait(sem);

sem\_getvalue(sem, &val);

printf("pid %ld has semaphore, value = %d\n", (long)getpid(), val);

pause(); /\* блокируется, пока не будет удален \*/

exit(0);

}

В листинге 7 приведена программа, которая выполняет операцию post для указанного семафора (то есть увеличивает его значение на 1), а затем получает значение этого семафора и выводит его.

**Листинг 7. (*sempost.c*) Увеличение значения семафора**

int main(int argc, char \*\*argv)

{

sem\_t \*sem;

int val;

if (argc != 2)

err\_quit("usage: sempost <name>");

sem = sem\_open(argv[1], 0);

sem\_post(sem);

sem\_getvalue(sem, &val);

printf("value = %d\n", val);

exit(0);

}

Ниже приведен пример создания семафора /sem1 и его использование в консольном режиме:

Изображение выглядит как текст, снимок экрана, Шрифт

Автоматически созданное описание

Этот пример показывает, что данный семафор обладает живучестью ядра. Кроме этого отметим, что при выходе из программы semwait, заблокировавшей семафор, значение его не изменяется, то есть ресурс остается заблокированным. Это отличает семафоры от блокировок fcntl, которые снимаются автоматически при завершении работы процесса.

**Реализация модели "производитель-потребитель" с буфером в виде файла**

Расширим командный интерфейс работы с семафорами Posix программами trader (производитель или поставщик) и user (потребитель), исходные коды которых представлены в листингах 8 и 9.

**Листинг 8. (*trader.c*) Поставщик**

#include <semaphore.h>

#include <fcntl.h>

#include <stdio.h>

#include <stdlib.h>

#include <sys/types.h>

#include <sys/stat.h>

int main(int argc, char \*\*argv)

{

sem\_t \*get = sem\_open(argv[1], 0);

sem\_t \*put = sem\_open(argv[2], 0);

char\* data = argv[3];

sem\_wait (put);

FILE\* buf = fopen("buf.txt", "w");

fprintf(buf, "%s", data);

fclose(buf);

sem\_post(get);

sem\_close(get);

sem\_close(put);

exit(0);

}

**Листинг 9. (*user.c*) Потребитель**

#include <semaphore.h>

#include <pthread.h>

#include <fcntl.h>

#include <stdio.h>

#include <stdlib.h>

#include <sys/types.h>

#include <sys/stat.h>

int main(int argc, char \*\*argv)

{

sem\_t \*get = sem\_open(argv[1], 0);

sem\_t \*put = sem\_open(argv[2], 0);

char data[50];

sem\_wait (get);

FILE\* buf = fopen("buf.txt", "r");

fgets (data, 50, buf);

printf("read buf: %s\n", data);

fclose(buf);

sem\_post(put);

sem\_close(get);

sem\_close(put);

exit(0);

}

На рис. 2 представлен процесс компиляции программ и инициализации семафоров get и put.

Изображение выглядит как текст, снимок экрана, Шрифт

Автоматически созданное описание

Рис. 2. Компиляция программ и создание семафоров

На рисунках 3 - 5 представлены примеры работы программа user и trader. Семафор put ограничивает возможность записи более одной строки текста в общий буфер, в роли которого выступает файл buf.txt, а семафор get управляет потребителем при считывании строки из буфера (при отсутствии записи в буфере sem\_wait(get) заблокирует выполнение потребителя). Поскольку семафор put инициализируется единицей, а get нулем, только производитель вначале имеет возможность работать с буфером. После этого семафор put остается закрытым, а открывается семафор get для возможности считывания буфера потребителем. После считывания потребитель открывает семафор put для производителя и возможен новый цикл передачи данных от производителя к потребителю.

Изображение выглядит как текст, снимок экрана, Шрифт

Автоматически созданное описание

Рис. 3. Работа программ и процесс изменения значений семафоров

Изображение выглядит как текст, снимок экрана, Шрифт

Автоматически созданное описание

Рис. 4. Программа trader ожидает, когда программа user считает данные из буфера

Изображение выглядит как текст, снимок экрана, Шрифт

Автоматически созданное описание

Рис. 5. Программа user ожидает, когда программа trader запишет данные в буфер

**Реализация модели "производитель-потребитель" с несколькими буферами**

Рассмотрим пример использования бинарных семафоров Posix для модели "производитель-потребитель" с несколькими буферами, находящимися в оперативной памяти. В листинге 9 приведен исходный код программы, где в качестве буферов используется целочисленный массив data[3], а в качестве семафоров используются массивы put[3] и get[3]. Таким образом, бинарный семафор put[k] управляет записью в буфер data[k], а бинарный семафор get[k] разрешает потребителю читать из буфера data[k], когда там появились новые данные.

Для генерации целых чисел производитель использует функцию produce. Поскольку при выполнении системной функции sem\_wait процесс может быть заблокирован, если семафор окажется закрытым, целесообразно предварительно проверить состояние семафора функцией sem\_trywait. Это позволит не выполнять функцию sem\_wait для закрытых семафоров.

В функции main мы создаем 6 семафоров (если они уже созданы, то создаются заново) и запускаем программные потоки для производителя и потребителя. В каждом потоке организован бесконечный цикл. Таким образом, в данной программе семафоры будут синхронизировать работу программных потоков, а не процессов.

**Листинг 9. (*produce\_consume\_3.c*) Программа "производитель-потребитель" с тремя буферами**

#include <pthread.h>

#include <stdio.h>

#include <semaphore.h>

#include <stdlib.h>

#include <unistd.h>

#include <errno.h>

#include <fcntl.h>

sem\_t \*put[3];

sem\_t \*get[3];

pthread\_t prod,cons;

int data[3];

int produce()

{

return rand()%1000;

sleep(1);

}

void \*producer(void \*arg)

{

for (;;)

{

sleep(1);

for(int k=0;k<3;k++)

{

if(sem\_trywait(put[k])==0){

sem\_wait(put[k]);

// производитель отправил данные в нужный буфер

data[k]=produce();

printf("%d buffer. Data produce: %d\n",k,data[k]);

sem\_post(get[k]);

}

}

}

return NULL;

}

void \*consumer(void \*arg)

{

for (;;)

{

for(int k=0;k<3;k++)

{

if(sem\_trywait(get[k])==0){

int t;

sem\_wait(get[k]);

// потребитель получил данные от нужного буфера

t=data[k];

printf("%d buffer. Data consume: %d\n",k,t);

sem\_post(put[k]);

}

}

}

return NULL;

}

int main(int argc, char \*\*argv)

{

put[0]=sem\_open("/put0",O\_RDWR | O\_CREAT,S\_IRUSR | S\_IWUSR | S\_IRGRP | S\_IROTH,1);

get[0]=sem\_open("/get0",O\_RDWR | O\_CREAT,S\_IRUSR | S\_IWUSR | S\_IRGRP | S\_IROTH,0);

put[1]=sem\_open("/put1",O\_RDWR | O\_CREAT,S\_IRUSR | S\_IWUSR | S\_IRGRP | S\_IROTH,1);

get[1]=sem\_open("/get1",O\_RDWR | O\_CREAT,S\_IRUSR | S\_IWUSR | S\_IRGRP | S\_IROTH,0);

put[2]=sem\_open("/put2",O\_RDWR | O\_CREAT,S\_IRUSR | S\_IWUSR | S\_IRGRP | S\_IROTH,1);

get[2]=sem\_open("/get2",O\_RDWR | O\_CREAT,S\_IRUSR | S\_IWUSR | S\_IRGRP | S\_IROTH,0);

pthread\_create(&prod, NULL, producer, NULL);

pthread\_create(&cons, NULL, consumer, NULL);

pthread\_join(prod, NULL);

pthread\_join(cons, NULL);

exit(0);

}

На рис. 6 показаны результаты работы программы.

Изображение выглядит как текст, снимок экрана

Автоматически созданное описание

Рис. 6. Результаты работы программы produce\_consume \_3.c

## Лекция №12. Создание командной оболочки ядра Linux на базе процессов и программных потоков

В версии 1 командной оболочки ядра системы Linux (см. лабораторную работу № 11) была приведена реализация на базе процессов для выполнения интерактивных команд. Версия 2 командной оболочки расширяет функциональность оболочки и добавляет три параллельных сервера-потока для выполнения неинтерактивных команд.

**Функция main простейшей оболочки ядра операционной системы Linux**

Код функции main () командной оболочки приведен в листинге 1.

**Листинг 1. (myshell.c) Код функции main () сервера ввода/вывода.**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <unistd.h>

#include <sys/stat.h>

#include <sys/ipc.h>

#include <sys/types.h>

#include <sys/time.h>

#include <time.h>

#include <errno.h>

#include <fcntl.h>

#include <limits.h>

#include <signal.h>

#include <sys/wait.h>

#include <sys/param.h>

#include <sys/resource.h>

#include <syslog.h>

#include <semaphore.h>

#include <sys/mman.h>

#include <pthread.h>

#define MAXLINE 4096

#define MAXBUFF 256

#define MAXQUEUE 6

#define MAXPATH 120

#define MAXLEN 256

#define MAXSERVERS 3

/\*Разрешения по умолчанию для вновь создаваемых файлов\*/

#define FILE\_MODE (S\_IRUSR | S\_IWUSR | S\_IRGRP | S\_IROTH)

#define SEM\_FLAGS (O\_RDWR | O\_CREAT)

char\* arg[MAXLEN]; //массив аргументов

char\* s\_path[MAXPATH];

char\* queue[MAXQUEUE]; // массив для хранения неинтерактивных команд

char\* queue0[MAXQUEUE]; // массив для хранения интерактивных команд

char\* t\_cmd[MAXSERVERS]; // массив команд, выполняемых серверами-потоками

char cmd[MAXLEN]; //строка, считанная с терминала

int errno;

int outfl; //флаг перенаправления вывода (замещение/дополнение)

pid\_t s0, s1;

int in\_out, p, NumberNoIntPr;

int fd\_1, fd\_0; //для сохранения дискрипторов потоков стандартного ввода/вывода

pthread\_t t\_id[3];

int flag [3], flag1[3], nflags = 0; // флаги состояния потоков

int i=0; // индекс потока

void preparse(char\* cmd);

int str\_parse(char \*cmd);

int input(char \*name);

int output(char \*name);

void ClientIntServ(char \*\*env);

void ClientNoIntServ();

void \*pt\_func0(void \*arg);

void \*pt\_func1(void \*arg);

void \*pt\_func2(void \*arg);

int main(int argc, char \*\*argv, char \*\*env)

{

setbuf(stdout, NULL); //отмена буфферизации

for(;;) {

write(1, "$>", 2); // вывод приглашения к вводу командной строки

gets(cmd); // чтение командной строки в массив cmd

preparse(cmd); // разбиение командной строки cmd на отдельные команды

// Запуск клиента сервера для выполнения интерактивных команд

ClientIntServ(env);

// Запуск клиента серверов для выполнения неинтерактивных команд

ClientNoIntServ();

}

return 0;

}

Функция main организует цикл для ввода и выполнения командных строк. Для каждой командной строки выдается промптер (приглашение к вводу), состоящее из трех символов. Пример командной строки:

date;pwd;ls;ls -al > f1&;echo "Метаданные" >> f1&;/bin/ls -l add.c >> f1&;cat add.c

Данная командная строка содержит четыре интерактивных и три неинтерактивных команды. После чтения и записи этой строки в массив cmd функции main, содержимое этого массива передается (копируется через аргумент функции) в массив cmd другой функции - функции preparse.

**Разбиение командной строки на отдельные команды**

Первый этап обработки командной строки, который выполняется функцией preparse (листинг 2), - это разбиение командной строки на отдельные команды и запись этих команд в массивы интерактивных (массив queue0) и неинтерактивных (массив queue) команд. Для приведенного выше примера командной строки будут сформированы следующие два массива:

|  |  |
| --- | --- |
| **Массив queue0** | **Массив queue** |
| date | ls -al > f1& |
| pwd | echo "Метаданные" >> f1& |
| ls | /bin/ls -l add.c >> f1& |
| cat add.c |  |
|  |  |

**Листинг 2. (*myshell.c*) Функция preparse() простейшей оболочки ядра**

void preparse(char\* cmd)

{

int i=0, j=0, k=0, cntr=0, cntr0=0;

char tmp[80];

char\* ptmp = tmp;

for (k=0; k<=79; k++) tmp[k]='\0'; // очистка массива tmp

// Очистка массивов команд

for (i=0; i<MAXQUEUE; i++) free(queue0[i]), queue0[i]=NULL; // очистка массива queue0

for (i=0; i<MAXQUEUE; i++) free(queue[i]), queue[i] = NULL; // очистка массива queue

i=0;

strcat(cmd, ";");

// разбиение командной строки cmd на отдельные команды

do {

if ((\*(cmd+i)!=';')&&(\*(cmd+i)!='\0')) {

\*(tmp+j)=\*(cmd+i); j++; //запись в tmp очередного символа

}

else { // сохранить очередную команду

if (strchr(ptmp, '&')) {

queue[cntr]=strdup(tmp); // сохранение неинтерактивной команды

cntr++;

}

else { // сохранение интерактивной команды

queue0[cntr0]=strdup(tmp);

cntr0++;

}

k=0; for(k=0; k<=79; k++) { tmp[k]='\0'; } // очистка массива tmp

j = 0;

}

i++;

}

while(\*(cmd+i)!='\0');

return;

}

При выполнении функции preparse массив tmp используется как буфер, в который посимвольно переписывается очередная команда. Символ ; является символом разделения команд и, соответственно, при его обнаружении завершится цикл записи символов в массив tmp:

if ((\*(cmd+i)!=';')&&(\*(cmd+i)!='\0')) {

\*(tmp+j)=\*(cmd+i); j++; //запись в tmp очередного символа

}

В данном фрагменте используются указатели и адресная арифметика языка Си. Этот фрагмент можно переписать с использованием массивов с индексами следующим образом:

if ((cmd[i])!=';')&&(cmd[i]!='\0')) {

tmp[j]= cmd[i]; j++; //запись в tmp очередного символа

}

Цикл do ..... while (\*(cmd+i)!='\0'); завершится, когда будет обнаружен символ конца командной строки (символ \0).

Отметим, что для сохранения очередной команды функция strdup дублирует строку, на которую указывает аргумент. Память под дубликат строки выделяется из кучи с помощью функции malloc и по окончанию работы с дубликатом должна быть очищена с помощью функции free.

**Синтаксический анализ строк каждой команды**

Каждая команда перед выполнением переписывается в массив cmd, который затем передается в функцию str\_parse (листинг 3) для разбиения ее на строки аргументов и записи их в массив arg. Отметим, что в строке arg[0] будет находиться имя команды, включая путь или без него.

Пример массива arg, полученного для последней команды из массива queue:

|  |  |
| --- | --- |
| **Массив arg** | **Элементы массива arg** |
| /bin/ls | arg[0] |
| -l | arg[1] |
| add.c | arg[2] |
| >> | arg[3] |
| f1 | arg[4] |
|  |  |

**Листинг 3. (*myshell.c*) Функция str\_parse () простейшей оболочки ядра**

/\*Разбор строки команды. Символы перенаправления ввода/вывода воспринимаются как

отдельные аргументы - пишутся через пробел. \*/

int str\_parse(char \*cmd)

{

char tmp[80];

int i=0,j=0,k,CNTR\_arg=0;

char \*name;

int in\_out = -1; //0 - поток перенаправлен

// Очистка массива-буфера для очередного аргумента

k=0; for (k=0; k<=79; k++) tmp[k]='\0'; // очистка массива tmp

// Очистка массива аргументов

for (i=0; i<MAXLEN; i++) free(arg[i]), arg[i] = NULL; // очистка массива arg

strcat(cmd, " "); //добавление пробела в конце команды

i=0;

// разбиение команды на аргументы

do {

if ((\*(cmd+i)!=' ')&&(\*(cmd+i)!='\0')) {

\*(tmp+j)=\*(cmd+i); j++; //запись в tmp очередного символа

}

else { // сохраняем очередной аргумент

arg[CNTR\_arg]=strdup(tmp);

CNTR\_arg++;

for(k=0; k<=79; k++) { tmp[k]='\0'; } // очистка массива tmp

j = 0;

}

i++;

}

while(\*(cmd+i)!='\0');

// Поиск символов перенаправления, выборка имени файла и

// модификация массива аркументов

for (i=0; arg[i] != NULL; i++) {

if ((strcmp(arg[i], ">")) ==0) {

arg[i]=NULL; name = arg[i+1];

arg[i+1]=NULL; in\_out = 0; outfl=0; output(name); break;}

else if ((strcmp(arg[i], ">>")) ==0) {

arg[i]=NULL;name = arg[i+1];

arg[i+1]=NULL; in\_out = 0; outfl=1; output(name); break;}

}

for (i=0; arg[i] != NULL; i++) {

if ((strcmp(arg[i], "<")) ==0) {

arg[i]=NULL; name = arg[i+1];

arg[i+1]=NULL; in\_out = 0; outfl=0; input(name); break;}

}

return in\_out;

}

При выполнении функции str\_parse массив tmp используется как буфер, в который посимвольно переписывается очередной аргумент команды:

if ((\*(cmd+i)!=' ')&&(\*(cmd+i)!='\0')) {

\*(tmp+j)=\*(cmd+i); j++; //запись в tmp очередного символа

}

Символы каждой опции считываются из cmd в tmp, а затем в массив arg. Символ пробела является символом разделения аргументов команды и, соответственно, при его обнаружении в массиве cmd завершится очередной цикл записи символов в массив tmp и будет выполнена запись массива tmp в очередной элемент массива arg.

Цикл do ..... while (\*(cmd+i)!='\0'); завершится, когда будет обнаружен символ конца строки команды (символ \0).

После завершения разбора строки команды и формирования массива arg производится просмотр аргументов команды для поиска в них символов перенаправления ввода-вывода. Тип найденного перенаправления сохраняется в переменной outfl (0 для перенаправления выходного потока, 1 - для входного потока), а имя файла потока переписывается в переменную name, значение которой передается в функции input или output (листинг 4) для соответствующей замены файлов стандартного потока ввода или вывода. После этого аргументы, содержащие символы перенаправления и имя потока, очищаются. Кроме этого, записывается 0 в переменную in\_out, значение которой будет значением, возвращаемым функцией str\_parse. Отметим, что при отсутствии перенаправления ввода-вывода в команде в переменной in\_out будет значение -1.

**Листинг 4. (*myshell.c*) Функции input () и output () простейшей оболочки ядра**

//Перенаправление ввода

int input(char \*name)

{

int fd;

if ((fd = open(name, O\_RDONLY)) < 0) return 0;

dup2(fd, 0); close(fd); return 1;

}

//Перенаправление вывода

int output(char \*name)

{

int fd;

if ((fd = open(name, O\_WRONLY)) <0) {

if (errno == ENOENT) //если файла нет, создадим

if ((fd = creat(name, 0644)) <0) return 0; // 0 - не успешно

if (errno == EACCES) { //права доступа?

write(2, "Shell: Доступ запрещен\n",25); return 0; }

//не имеем права писать в файл

}

// ">> file" указатель на конец файла

else if (outfl == 1) lseek(fd, 1, SEEK\_END);

if (outfl == 0) {

//"> file" пытаемся опустошить файл

if ((fd = creat(name, 0644)) <0) return 0; }

//перенаправляем вывод

dup2(fd, 1);

close(fd); return 1; //1 = успешно

}

Функция input заменяет стандартный поток ввода (поток stdin) с файловым дескриптором 0 на поток из файла, имя которого сохранено в переменной name.

Функция output заменяет стандартный поток вывода (поток stdout) с файловым дескриптором 1 на поток в файл, имя которого сохранено в переменной name.

**Выполнение интерактивной команды**

Функция main вызывает функцию ClientIntServ (листинг 5), в которой организован цикл

while (queue0[p]!=NULL) { ... }

для чтения и выполнения интерактивных команд. Каждая считанная из массива queue0 команда записывается в массив cmd. Затем клиент с помощью fork порождает рабочий процесс сервера S0, который вызывает функцию str\_parse для разбиения команды в массиве cmd на аргументы и записи их в массив arg. После этого сервер с помощью системной функции execvp выполняет интерактивную команду, имя которой находится в arg[0].

**Листинг 5. (*myshell.c*) Код клиента и сервера S0.**

void ClientIntServ(char \*\*env)

{

char bufs0[MAXLINE];

pid\_t ex;

int n;

p=0;

// цикл по интерактивным командам

while (queue0[p]!=NULL)

{

// запись очередной команды в массив cmd

snprintf(cmd, sizeof(cmd), "%s", queue0[p]);

p++;

// создание рабочего процесса сервера S0

if((ex = fork()) == 0)

{

in\_out = str\_parse(cmd); //разбор строки команды

// Выполнение сервером команд – исполнимых файлов

if(execvp(arg[0], arg) == -1)

{

// Выполнение встроенных в оболочку команд

if (!strcmp(arg[0], "showenv"))

{

int i;

printf("\nEnvironment:\n");

for (i=0; env[i] != NULL; i++)

printf("\n%s", env[i]);

printf("\n");

}

else

printf("I MySh: command < %s > not found!\n",arg[0]);

}

// завершение рабочего процесса

printf("I Pid [%d] is Dying\n", ex);

exit(0);

}

waitpid(ex,NULL,0);

printf("I Pid [%d] is Dead\n", ex);

}

printf("ClientInt End\n");

}

**Выполнение неинтерактивных команд**

В листинге 2 показан код клиента и серверов-потоков для выполнения неинтерактивных команд.

**Листинг 2. (*myshell.c*) Код клиента и серверов для выполнения неинтерактивных команд.**

void ClientNoIntServ()

{

p=0; // индекс команды

i=0; // индекс потока

int k;

// Инициализация флагов

for (k = 0; k<3; k++){

flag[k] = 0;

flag1[k] = 0;

}

nflags = 0;

// цикл по неинтерактивным командам

while (queue[p]!=NULL) {

// запись очередной команды в массив cmd

snprintf(cmd, sizeof(cmd), "%s", queue[p]);

p++;

// Удаление символа & из команды

for(k=0; k<MAXLEN; k++) {

if ((cmd[k] == '&')) {

cmd[k]= '\0';

break;

}

}

// команда для потока с индексом i

t\_cmd[i]=strdup(cmd); // подготовлена команда

flag[i] = 1; // запущена потоковая функция

flag1[i] = 1; // создана среда потока в ядре

nflags++; // счетчик запущенных потоков

switch(i){

case 0:

pthread\_create (&t\_id[0], NULL, &pt\_func0, NULL);

break;

case 1:

pthread\_create (&t\_id[1], NULL, &pt\_func1, NULL);

break;

case 2:

pthread\_create (&t\_id[2], NULL, &pt\_func2, NULL);

break;

}

i++;

if(i==3) break; // запущены все потоки

}

if (i<3){ // список команд исчерпан

for (k = 0; k<p; k++){

pthread\_join(t\_id[k], NULL);

flag1[k] = 0; // поток завершен в ядре

nflags--;

}

}

else{

if (queue[p]!=NULL){

while (queue[p]!=NULL) {

sleep(1);

// запись очередной команды в массив cmd

snprintf(cmd, sizeof(cmd), "%s", queue[p]);

// Удаление символа & из команды

for(k=0; k<MAXLEN; k++) {

if ((cmd[k] == '&')) {

cmd[k]= '\0';

break;

}

}

// Ищем поток, который завершился

for (k = 0; k<3; k++){

if (!flag[k] && flag1[k]){ // потоковая функция завершилась

pthread\_join(t\_id[k], NULL);

flag1[k] = 0; // поток завершен в ядре

nflags--;

break;

}

}

if (k<3){

// команда для потока с индексом k

t\_cmd[k]=strdup(cmd); // подготовлена команда

flag[k] = 1;

flag1[k] = 1;

nflags++;

switch(k){

case 0:

pthread\_create (&t\_id[0], NULL, &pt\_func0, NULL);

break;

case 1:

pthread\_create (&t\_id[1], NULL, &pt\_func1, NULL);

break;

case 2:

pthread\_create (&t\_id[2], NULL, &pt\_func2, NULL);

break;

}

p++;

}

}

}

}

while (nflags) {

// Ищем поток, который завершился

for (k = 0; k<3; k++){

if (!flag[k] && flag1[k]){ // поток завершился

pthread\_join(t\_id[k], NULL);

flag1[k] = 0; // поток завершен в ядре

nflags--;

}

}

}

printf("ClientNoInt End\n");

return;

}

// потоковая функция

void \*pt\_func0(void \*arg)

{

system (t\_cmd[0]);

flag[0] = 0;

return NULL;

}

// потоковая функция

void \*pt\_func1(void \*arg)

{

system (t\_cmd[1]);

flag[1] = 0;

return NULL;

}

// потоковая функция

void \*pt\_func2(void \*arg)

{

system (t\_cmd[2]);

flag[2] = 0;

return NULL;

}

Отметим, что в массиве queue может находиться максимум 6 команд (#define MAXQUEUE 6), а для выполнения этих команд создано 3 сервера-потока (#define MAXSERVERS 3). Следовательно, алгоритм работы клиента и серверов выполнения неинтерактивных команд должен предусмотреть следующие два случая: а) число неинтерактивных команд в командной строке не превышает числа серверов-потоков; б) число неинтерактивных команд превышает число серверов-потоков. И если в первом случае алгоритм работы будет очень простым - запустить число потоков, равное числу команд, а затем дождаться завершения работы всех потоков, то во втором случае алгоритм будет достаточно сложным. Рассмотрим его.

Вначале мы запускаем все 3 сервера-потока для выполнения команд. Далее мы ожидаем, когда один из этих потоков завершит выполнение своей команды. Тогда для этого сервера мы перезапускаем поток для выполнения следующей команды из массива queue. Перезапуски серверов повторяем до тех пор, пока не будет запущено выполнение всех команд из массива queue. После этого будем ожидать завершение работы всех запущенных серверов.

Чтобы контролировать состояние серверов-потоков, введены следующие переменные и массивы:

nflags - число работающих серверов;

flag[k] - флаг, сообщающий, что у сервера k выполняется потоковая функция;

flag1[k] - флаг, сообщающий, что у сервера k создана (еще не удалена) среда потока в ядре.

Оба флага устанавливаются при создании потока. Флаг flag[k] сбрасывается, когда потоковая функция завершает свое выполнение, например, для k=0:

// потоковая функция

void \*pt\_func0(void \*arg)

{

system (t\_cmd[0]);

flag[0] = 0;

return NULL;

}

Флаг flag1[k] сбрасывается, когда для потока будет выполнена функция

pthread\_join(t\_id[k], NULL);

Отметим, что для каждого k-го сервера-потока используются следующие переменные, массивы или элементы массивов:

pt\_func0 - имя потоковой функции для сервера k=0;

t\_cmd[k]- команда, выполняемая сервером k;

t\_id[k] - идентификатор потоковой функции, созданной для сервера k.

**Контрольный пример**

На рис. 1 представлены результаты проверки разработанной простейшей командной оболочки ядра Linux.

Изображение выглядит как текст, снимок экрана, программное обеспечение, Шрифт

Автоматически созданное описание

Рис. 1. Тестирование оболочки ядра Linux на базе процессов и потоков

## Лекция №13. Создание командной оболочки ядра Linux на базе процессов, программных каналов и семафоров Posix

Версия 1 командной оболочки ядра системы Linux была создана на базе процессов для выполнения интерактивных команд. Версия 2 командной оболочки содержит три параллельных сервера-потока для выполнения неинтерактивных команд. Данная версия оболочки (версия 3) использует программные каналы и семафоры Posix для передачи данных между серверами и синхронизации работы серверов.

**Функция main простейшей оболочки ядра операционной системы Linux**

Код функции main () командной оболочки приведен в листинге 1.

**Листинг 1. (myshell.c) Код функции main () сервера ввода/вывода.**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <unistd.h>

#include <sys/stat.h>

#include <sys/ipc.h>

#include <sys/types.h>

#include <sys/time.h>

#include <time.h>

#include <errno.h>

#include <fcntl.h>

#include <limits.h>

#include <signal.h>

#include <sys/wait.h>

#include <sys/param.h>

#include <sys/resource.h>

#include <syslog.h>

#include <semaphore.h>

#include <sys/mman.h>

#define ARGMAXLEN 256

#define CMDMAXLEN 512

#define TMPMAXLEN 128

#define QUEMAXLEN 20

#define RESULTTYPEMAXLEN 2

// Семафоры

#define SEM0get "/sem0get" // Интерактивная команда передана в канал

#define SEM1get "/sem1get" // Неинтерактивная команда передана в канал

#define SEM01end "/sem01end" // Завершено выполнение команды сервером

#define SEMhget "/semhget" // Команда для сервера истории передана в канал

#define SEMhend "/semhend" // Окончание выполнения команды сервером истории

// Права доступа для создаваемых семафоров

#define FILE\_MODE (S\_IRUSR | S\_IWUSR | S\_IRGRP | S\_IROTH)

#define SEM\_FLAGS (O\_RDWR | O\_CREAT)

pid\_t s0, s1, sh; // серверы-процессы

// Объявление семафоров Posix

sem\_t \*sem0get;

sem\_t \*sem1get;

sem\_t \*sem01end;

sem\_t \*semhget;

sem\_t \*semhend;

char\* arg[ARGMAXLEN]; // массив аргументов

char cmdline[CMDMAXLEN]; // командная строка

char\* cmdqueue[QUEMAXLEN]; // массив для хранения команд

int typequeue[QUEMAXLEN]; // массив для хранения типов команд

char\* cmdhistory[QUEMAXLEN]; // массив для хранения команд

//Каналы

int cmdpipe [2]; // канал к дочерним процессам-серверам s0, s1, sh

int historypipe [2]; // канал для передачи типа команды серверу истории

int wrcmdpipe, rdcmdpipe, wrhistorypipe, rdhistorypipe; // входы и выходы каналов

int fd\_1, fd\_0; // для сохранения дискрипторов потоков стандартного ввода/вывода

int main(int argc, char \*\*argv, char \*\*env)

{

int cmdcnt = 0; // Число введенных команд

pipe(cmdpipe); // создание неименованного канала для серверов выполнения команд

pipe(historypipe); // создание неименованного канала для сервера истории

rdcmdpipe = cmdpipe[0];

wrcmdpipe = cmdpipe[1];

rdhistorypipe = historypipe[0];

wrhistorypipe = historypipe[1];

// Создание и инициализация семафоров

sem\_unlink(SEM1get);

sem\_unlink(SEM0get);

sem\_unlink(SEM01end);

sem\_unlink(SEMhget);

sem\_unlink(SEMhend);

sem1get = sem\_open(SEM1get, SEM\_FLAGS, FILE\_MODE, 0);

sem0get = sem\_open(SEM0get, SEM\_FLAGS, FILE\_MODE, 0);

sem01end = sem\_open(SEM01end, SEM\_FLAGS, FILE\_MODE, 0);

semhget = sem\_open(SEMhget, SEM\_FLAGS, FILE\_MODE, 0);

semhend = sem\_open(SEMhend, SEM\_FLAGS, FILE\_MODE, 0);

// создание процесса сервера интерактивных команд

if ((s0 = fork()) == 0)

{

IntServ(env);

}

// создание процесса сервера неинтерактивных команд

if ((s1 = fork()) == 0)

{

NoIntServ(env);

}

// создание процесса сервера неинтерактивных команд

if ((sh = fork()) == 0)

{

HistoryServ();

}

for(;;)

{

char tmp[TMPMAXLEN];

for (int k=0; k<TMPMAXLEN; k++)

{

tmp[k]='\0';

}

for (int k=0; k<QUEMAXLEN; k++)

{

cmdqueue[k]=strdup(tmp);

typequeue[k]=0;

}

write(1, "$> ", 3); // вывод приглашения к вводу командной строки

gets(cmdline); // чтение командной строки в массив cmd

cmdcnt = parsecmd(cmdline); // разбиение командной строки cmd на отдельные команды

for(int i = 0;i < cmdcnt;i++)

{

write(wrcmdpipe, cmdqueue[i],strlen(cmdqueue[i])); // Запись команды в канал

if (typequeue[i]==1) // передача неинтерактивной команды

{

sem\_post(sem1get);

}

else if (typequeue[i]==2) // передача интерактивной команды

{

sem\_post(sem0get);

}

sem\_wait(sem01end); // ожидание завершения выполнения команды

}

}

// принудительное завершение процессов серверов

kill (s0, SIGTERM);

kill (s1, SIGTERM);

kill (sh, SIGTERM);

}

Функция main организует цикл для ввода и выполнения командных строк. Для каждой командной строки выдается промптер (приглашение к вводу), состоящее из трех символов. Пример командной строки:

date;pwd;(ls &);(ls -al &);cat add.c;(/bin/sort < add.c &);history

Данная командная строка содержит четыре интерактивных и три неинтерактивных команды. Отметим, что в данной версии оболочки реализован только один вид перенаправления ввода-вывода - перенаправление стандартного потока ввода (символ < в командной строке).

После запуска оболочки создаются три сервера-процесса: сервер истории sh и серверы для выполнения интерактивных и неинтерактивных команд - серверы s0 и s1 соответственно. Создаются также два программных канала: cmdpipe и historypipe, которые при создании серверов как дочерних процессов будут доступны из этих серверов. Аналогично создаются и будут доступны из серверов 4 семафора Posix, открытие которых соответствует определенным событиям:

sem0get - интерактивная команда передана в канал cmdpipe (сервер s0 может ее читать);

sem1get - неинтерактивная команда передана в канал cmdpipe (сервер s1 может ее читать);

sem01end - завершено выполнение интерактивной или неинтерактивной команды сервером;

semhget - данные для сервера истории переданы в каналы cmdpipe и historypipe;

semhend - окончание выполнения команды сервером истории.

Командная строка считывается в массив cmdline и вызывается функция parsecmd для преобразования командной строки в два массива: массив команд cmdqueue и массив типов команд typequeue. Тип каждой команды - неинтерактивная или интерактивная - задается числами 1 или 2 соответственно.

После формирования этих массивов начинает выполняться цикл запуска команд на выполнение: каждая команда из массива cmdqueue передается в канал cmdpipe и в зависимости от типа этой команды открывается семафор sem0get или sem1get для начала обработки команды в сервере s0 или s1 соответственно. После завершения выполнения команды открывается семафор sem01end и следующая команда из массива cmdqueue будет запущена на выполнение.

**Разбиение командной строки на отдельные команды**

Первый этап обработки командной строки, который выполняется функцией parsecmd (листинг 2), - это разбиение командной строки на отдельные команды и запись этих команд в массивы cmdqueue и typequeue.

**Листинг 2. (*myshell.c*) Функция parsecmd () простейшей оболочки ядра**

int parsecmd(char\* cmdline)

{

int i=0, j=0, cntr=0;

char tmp[TMPMAXLEN];

char\* ptmp = tmp;

for (int k=0; k<TMPMAXLEN; k++)

{

tmp[k]='\0';

}

if(cmdline[strlen(cmdline)-1]!=';')

strcat(cmdline,";"); // добавление в конец ";"

while(\*(cmdline+i)!='\0')

{

if(\*(cmdline+i)==';')

{

if (strchr(ptmp, '&'))

{ // сохранение признака неинтерактивной команды

typequeue[cntr]=1;

}

else

{ // сохранение признака интерактивной команды

typequeue[cntr]=2;

}

cmdqueue[cntr]=strdup(tmp); // сохранение команды

cntr++;

for(int k=0; k<TMPMAXLEN; k++)

{

tmp[k]='\0';

}

j = 0;

}

else if((\*(cmdline+i)!='(')&&(\*(cmdline+i)!=')'))

{

\*(tmp+j)=\*(cmdline+i); //запись в tmp очередного символа

j++;

}

i++;

}

return cntr;

}

При выполнении функции parsecmd массив tmp используется как буфер, в который посимвольно переписывается очередная команда. При этом символы ( и ) пропускаются. Символ ; является символом разделения команд и, соответственно, при его обнаружении завершится цикл записи символов в массив tmp:

\*(tmp+j)=\*(cmdline+i); j++; //запись в tmp очередного символа

В данном фрагменте используются указатели и адресная арифметика языка Си. Этот фрагмент можно переписать с использованием массивов с индексами следующим образом:

tmp[j]= cmdline[i]; j++; //запись в tmp очередного символа

Отметим, что для сохранения очередной команды функция strdup дублирует строку, на которую указывает аргумент:

cmdqueue[cntr]=strdup(tmp); // сохранение команды

Память под дубликат строки выделяется из кучи с помощью функции malloc и по окончанию работы с дубликатом память должна быть очищена с помощью функции free.

**Синтаксический анализ строк каждой команды**

При работе серверов s0 и s1 каждая команда после пересылки ее на сервер истории и перед началом ее выполнения передается в функцию converttoarg (листинг 3) для разбиения ее на строки аргументов и записи их в массив arg. Отметим, что в строке arg[0] будет находиться имя команды, включая путь или без него.

**Листинг 3. (*myshell.c*) Функция converttoarg () простейшей оболочки ядра**

void converttoarg(char \*cmd)

{

int i=0, j=0, cntr=0;

char \*name;

char tmp[TMPMAXLEN];

for (int k=0; k<TMPMAXLEN; k++)

{

tmp[k]='\0';

}

for (int k=0; k<ARGMAXLEN; k++)

{

arg[k]=NULL;

}

strcat(cmd, " "); // добавление пробела в конец команды

while(\*(cmd+i)!='\0')

{

if(\*(cmd+i)==' ')

{

if((strcmp(tmp,"&")!=0)&&(strcmp(tmp,"\0")!=0))

{

arg[cntr]=strdup(tmp); // сохранение команды

cntr++;

}

for(int k=0; k<TMPMAXLEN; k++)

{

tmp[k]='\0';

}

j = 0;

}

else

{ // выборка очередного символа

\*(tmp+j)=\*(cmd+i);

j++;

}

i++;

}

for (i=0; arg[i] != NULL; i++)

{

if ((strcmp(arg[i], "<")) ==0)

{

arg[i]=NULL;

name = arg[i+1];

arg[i+1]=NULL;

int fd;

if ((fd = open(name, O\_RDONLY)) >= 0)

dup2(fd, 0);

close(fd);

break;

}

}

}

При выполнении функции converttoarg массив tmp используется как буфер, в который посимвольно переписывается очередной аргумент команды.

Символы каждой опции считываются из cmd в tmp, а затем в массив arg. Символ пробела является символом разделения аргументов команды и, соответственно, при его обнаружении в массиве cmd завершится очередной цикл записи символов в массив tmp и будет выполнена запись массива tmp в очередной элемент массива arg.

После завершения разбора строки команды и формирования массива arg производится просмотр аргументов команды для поиска в них символов перенаправления ввода-вывода. Тип найденного перенаправления сохраняется в переменной outfl (0 для перенаправления выходного потока, 1 - для входного потока), а имя файла потока переписывается в переменную name, значение которой используется для соответствующей замены файлов стандартного потока ввода или вывода. При этом аргументы, содержащие символы перенаправления и имя потока, очищаются.

**Выполнение интерактивных команд**

Сервер s0 выполняет функцию IntServ (листинг 4), в которой организован цикл

while (1) { ... }

для чтения из канала cmdpipe и выполнения интерактивных команд. Каждая считанная из канала команда записывается в массив tmp. Затем сервер s0 с помощью системной функции fork порождает рабочий процесс, который пересылает команду на сервер истории, а по завершению работы сервера истории вызывает функцию converttoarg для разбиения команды в массиве tmp на аргументы и записи их в массив arg. После этого сервер с помощью системной функции execvp выполняет интерактивную команду, имя которой находится в arg[0].

**Листинг 4. (*myshell.c*) Код сервера для выполнения интерактивных команд.**

void IntServ(char \*\*env)

{

char tmp[TMPMAXLEN];

char resulttype[RESULTTYPEMAXLEN];

while(1)

{

pid\_t ex;

sem\_wait(sem0get); // ожидаем поступления интерактивной команды в канал

for (int k=0; k<TMPMAXLEN; k++)

{

tmp[k]='\0';

}

for (int k=0; k<RESULTTYPEMAXLEN; k++)

{

resulttype[k]='\0';

}

read(rdcmdpipe,tmp,TMPMAXLEN); // читаем команду из канала

//сохранение текущих дискрипторов потоков стандартного ввода/вывода

fd\_1=dup(1);

fd\_0=dup(0);

printf("IntServ = <%s>\n",tmp);

if((ex = fork()) == 0)

{

resulttype[0]='c';

write(wrhistorypipe, resulttype, RESULTTYPEMAXLEN); // Пересылка типа команды

write(wrcmdpipe, tmp,TMPMAXLEN); // Пересылаем команду на сервер истории

sem\_post(semhget);

sem\_wait(semhend); // ожидаем завершения обработки команды сервером истории

converttoarg(tmp); //преобразование строки команды в массив аргументов

// Выполнение команд – исполнимых файлов

if(execvp(arg[0], arg) == -1)

{

// Выполнение встроенных в оболочку команд

if (!strcmp(arg[0], "showenv"))

{

printf("\nEnvironment:\n");

for (int i=0; env[i] != NULL; i++)

printf("\n%s", env[i]);

printf("\n");

}

else if (!strcmp(arg[0], "history"))

{

printf("History:\n");

resulttype[0]='h';

// Пересылка типа команды на сервер истории

write(wrhistorypipe, resulttype, RESULTTYPEMAXLEN);

sem\_post(semhget);

// ожидаем завершения обработки команды сервером истории

sem\_wait(semhend);

}

else

{

printf("I MySh: command < %s > not found!\n",arg[0]);

resulttype[0]='e';

// Пересылка типа команды на сервер истории

write(wrhistorypipe, resulttype, RESULTTYPEMAXLEN);

sem\_post(semhget);

// ожидаем завершения обработки команды сервером истории

sem\_wait(semhend);

}

}

// завершение рабочего процесса

exit(0);

}

waitpid(ex,NULL,0);

close(fd\_1);

close(fd\_0);

sem\_post(sem01end); // сервер завершил выполнение команды

}

}

**Выполнение неинтерактивных команд**

Сервер s1 выполняет функцию NoIntServ (листинг 5), в которой организован цикл

while (1) { ... }

для чтения из канала cmdpipe и выполнения неинтерактивных команд. Каждая считанная из канала команда записывается в массив tmp. Затем сервер s1 с помощью системной функции fork порождает рабочий процесс, который пересылает команду на сервер истории, а по завершению работы сервера истории вызывает функцию converttoarg для разбиения команды в массиве tmp на аргументы и записи их в массив arg. После этого сервер с помощью системной функции execvp выполняет интерактивную команду, имя которой находится в arg[0].

**Листинг 5. (*myshell.c*) Код сервера для выполнения неинтерактивных команд.**

void NoIntServ(char \*\*env)

{

char tmp[TMPMAXLEN];

char resulttype[RESULTTYPEMAXLEN];

while(1)

{

pid\_t ex;

sem\_wait(sem1get); // ожидаем поступления неинтерактивной команды в канал

for (int k=0; k<TMPMAXLEN; k++)

{

tmp[k]='\0';

}

for (int k=0; k<RESULTTYPEMAXLEN; k++)

{

resulttype[k]='\0';

}

read(rdcmdpipe,tmp,TMPMAXLEN); // читаем команду из канала

//сохранение текущих дискрипторов потоков стандартного ввода/вывода

fd\_1=dup(1);

fd\_0=dup(0);

printf("NoIntServ = <%s>\n",tmp);

if((ex = fork()) == 0)

{

resulttype[0]='c'; // задаем тип команды

// Пересылка типа команды на сервер истории

write(wrhistorypipe, resulttype, RESULTTYPEMAXLEN);

write(wrcmdpipe, tmp,TMPMAXLEN); // Пересылаем команду на сервер истории

sem\_post(semhget);

sem\_wait(semhend); // ожидаем завершения обработки команды сервером истории

converttoarg(tmp); //преобразование строки команды в массив аргументов

// Выполнение команд – исполнимых файлов

if(execvp(arg[0], arg) == -1)

{

// Выполнение встроенных в оболочку команд

if (!strcmp(arg[0], "showenv"))

{

printf("Environment:\n");

for (int i=0; env[i] != NULL; i++)

printf("\n%s", env[i]);

printf("\n");

}

else if (!strcmp(arg[0], "history"))

{

printf("History:\n");

resulttype[0]='h';

// Пересылка типа команды на сервер истории

write(wrhistorypipe, resulttype, RESULTTYPEMAXLEN);

sem\_post(semhget);

// ожидаем завершения обработки команды сервером истории

sem\_wait(semhend);

}

else

{

printf("I MySh: command < %s > not found!\n",arg[0]);

resulttype[0]='e';

// Пересылка типа команды на сервер истории

write(wrhistorypipe, resulttype, RESULTTYPEMAXLEN);

sem\_post(semhget);

// ожидаем завершения обработки команды сервером истории

sem\_wait(semhend);

}

}

// завершение рабочего процесса

exit(0);

}

// вывод номера и pid рабочего процесса

printf("Start NotInt Pid: [%d]\n", ex);

waitpid(ex,NULL,0);

// вывод pid завершившегося рабочего процесса

printf("Done NotInt Pid: [%d]\n", ex);

close(fd\_1);

close(fd\_0);

sem\_post(sem01end); // сервер завершил выполнение команды

}

}

**Сервер истории**

Серверы s0 и s1 обращаются к серверу истории (серверу sh), когда получают из канала cmdpipe команду и создают рабочий процесс для ее выполнения, например:

resulttype[0]='c'; // задаем тип команды

// Пересылка типа команды на сервер истории

write(wrhistorypipe, resulttype, RESULTTYPEMAXLEN);

write(wrcmdpipe, tmp,TMPMAXLEN); // Пересылаем команду на сервер истории

sem\_post(semhget);

sem\_wait(semhend); // ожидаем завершения обработки команды сервером истории

Серверы s0 и s1 повторно обращаются к серверу истории, если команду, ранее переданную на сервер истории, не удалось выполнить:

resulttype[0]='e';

// Пересылка типа команды на сервер истории

write(wrhistorypipe, resulttype, RESULTTYPEMAXLEN);

sem\_post(semhget);

// ожидаем завершения обработки команды сервером истории

sem\_wait(semhend);

При выполнении команды history также происходит обращение к серверу истории:

else if (!strcmp(arg[0], "history"))

{

printf("History:\n");

resulttype[0]='h';

// Пересылка типа команды на сервер истории

write(wrhistorypipe, resulttype, RESULTTYPEMAXLEN);

sem\_post(semhget);

// ожидаем завершения обработки команды сервером истории

sem\_wait(semhend);

}

Код сервера истории приведен в листинге 6.

**Листинг 6. (*myshell.c*) Код сервера истории.**

void HistoryServ()

{

//printf("HistoryServ start\n");

int historycntr=0;

char tmp[TMPMAXLEN];

char resulttype[RESULTTYPEMAXLEN];

while(1)

{

sem\_wait(semhget); // ожидаем поступления данных для сервера истории

for (int k=0; k<TMPMAXLEN; k++)

{

tmp[k]='\0';

}

for (int k=0; k<RESULTTYPEMAXLEN; k++)

{

resulttype[k]='\0';

}

read(rdhistorypipe,resulttype,RESULTTYPEMAXLEN); // чтение типа команды

if ((strcmp(resulttype, "c")) ==0)

{

read(rdcmdpipe,tmp,TMPMAXLEN); // чтение команды

cmdhistory[historycntr]=strdup(tmp);

historycntr++;

}

else if ((strcmp(resulttype, "e")) ==0)

{

historycntr--;

cmdhistory[historycntr]=strdup(tmp);

}

else if ((strcmp(resulttype, "h")) ==0)

{

for (int k=0; k<historycntr; k++)

{

printf("%d %s\n", k, cmdhistory[k]);

}

}

sem\_post(semhend);

}

}

**Контрольный пример**

На рис. 1 представлены результаты проверки разработанной простейшей командной оболочки ядра Linux.

Изображение выглядит как текст, снимок экрана

Автоматически созданное описание

Рис. 1. Тестирование оболочки ядра Linux на базе процессов, каналов и семафоров

## Лекция №14. Приложения и компоненты графического интерфейса операционных систем семейства Linux

**Устройство системы X Window**

**X Window System** - это оконная система, обеспечивающая стандартные инструменты и протоколы для построения графических интерфейсов пользователей. Она была разработана в Массачусетском технологическом институте (MIT) в 1984 году и используется в UNIX-подобных операционных системах.

Система X Window построена на основе модели "клиент/сервер" (рис. 1). Правда, модель эта в данном случае используется как бы в "перевернутом" виде. Дело в том, что X-сервер запускается на компьютере пользователя (а не на каком-то удаленном "сервере") и обеспечивает вывод изображения на экран монитора. Эта программа работает непосредственно с "железом" и обеспечивает управление как устройствами ввода (клавиатура, мышь и так далее), так и устройствами вывода (дисплей, монитор, динамик). X-сервер "захватывает" оборудование и предоставляет его возможности другим программам (клиентам сервера) как ресурсы (собственно, именно поэтому он и считается сервером) по особому протоколу, который называется X-протокол, или протокол сетевой связи (X Network Protocol).

Изображение выглядит как текст, диаграмма, снимок экрана, дисплей

Автоматически созданное описание

Рис.1. Архитектура системы X Window

Система X Window реализует сетевую архитектуру и обеспечивает удаленный графический интерфейс к удаленным приложениям других ЭВМ, подключенных к сети (рис. 2). Отметим, что **X-сервер** взаимодействует с приложениями при помощи механизма **unix-сокетов** и **internet-сокетов**. Возможность взаимодействия через **internet-сокеты** позволяет **X-серверу** и **клиенту** работать в связке, находясь на разных компьютерах, объединенных в сеть.

**Проект архитектуры** X Window System не определяет, как должен выглядеть интерфейс пользователя приложения (кнопки, меню, заголовки окон и другие элементы). Эти вопросы решаются на уровнях:

* оконных менеджеров (window manager, WM), управляющих размещением и внешним видом окон приложений;
* элементов интерфейса (widget), - примитивов графического интерфейса пользователя, имеющих стандартный внешний вид и выполняющих стандартные действия по управлению этим интерфейсом;
* сред рабочего стола (desktop environment, DE), - разновидности графических интерфейсов пользователя, основанные на метафоре рабочего стола;
* на уровне отдельных приложений.

По этой причине, визуальное представление X-интерфейсов претерпело, с течением времени, большие изменения.

Изображение выглядит как текст, снимок экрана, Шрифт, линия

Автоматически созданное описание

Рис 2. Сетевая архитектура X Window System

Таким образом, чтобы получить на экране какие-то более содержательные изображения, одного X-сервера недостаточно, надо запустить менеджер окон и хотя бы одну программу-клиент, которая будет формировать изображение и обрабатывать сообщения о действиях пользователя (например, щелчок кнопкой мыши и т.п.). В роли "клиентов" X-сервера выступают приложения, работающие с X Window, например, графический или текстовый редакторы, эмулятор терминала xterm и другие.

Менеджер окон – это особое приложение, которое тоже является клиентом X-сервера, но выполняет особую роль – оно управляет другими клиентами. Одна из основных функций, которые выполняет любой менеджер окон – это обеспечение возможности запуска других приложений. Кроме того, менеджер окон обеспечивает выполнение всех операций с окнами: прорисовку рамок, меню, иконок, полос прокрутки и других элементов окна, предоставляет возможность изменять вид и положение окна в процессе работы в соответствии с потребностями пользователя, а также обеспечивает вывод информации от приложения в соответствующее окно.

Взаимодействие между менеджером окон и X-сервером осуществляется в асинхронном режиме путем обмена сообщениями. Клиентские программы открывают соединение с сервером, и затем просто посылают ему запросы примерно такого типа: “нарисуй прямую линию от точки такой-то до точки такой-то” или “выведи эту строку текста таким-то шрифтом начиная с такой-то позиции экрана”. Эти запросы, в основном, обрабатываются путем вызова соответствующих процедур из библиотеки X-Lib, которая содержит набор стандартных функций, которые обеспечивают выполнение низкоуровневых операций с графическими образами. Менеджер окон вызывает функции из X-Lib для управления дисплеем и выполнения любых преобразований изображений в окнах.

Надо сказать, что хотя элементы графического интерфейса (иконки, кнопки, диалоговые окна, линейки прокрутки, различные рамки и оконные меню) и прорисовываются на экране с помощью низкоуровневых функций из библиотеки X-Lib, но приложения не вызывают эти функции непосредственно, а обращаются к функциям более высокого уровня, которые в англоязычной документации называют виджетами. Ведь чтобы нарисовать, например, с помощью X-Lib простую кнопку с надписью «Ввод», необходимо прорисовать два прямоугольника, скруглить углы, изобразить тень, вывести надпись, обеспечить изменение вида кнопки при наведении мыши и так далее. Чтобы не повторять каждый раз работу по программированию часто используемых элементов графического интерфейса, были разработаны несколько отдельных библиотек таких элементов (виджетов). Эти библиотеки иногда называют "тулкитами" ("toolkit"). Наиболее известными из них являются библиотеки Motif, Qt и GTk.

Можно еще отметить, что X-сервер может обеспечивать работу с несколькими "дисплеями" и "экранами" одновременно. Правда, термины “экран” и “дисплей” в системе X Window имеют специальное значение. Когда X-сервер запускается, он инициализирует один или несколько “дисплеев”. Каждый “дисплей” включает в себя не только видео-компонент, но и клавиатуру, мышь и другие устройства ввода. Конечно, отдельный пользователь в любой момент времени посредством одного комплекта устройств (монитора, клавиатуры и мыши) имеет доступ только к одному “дисплею”. Но имеется возможность запустить несколько “дисплеев”, которые могут быть как локальными (виртуальными), так и расположенными на других хостах, соединенных с данным компьютером по сети. Каждый “дисплей” может иметь собственную конфигурацию (например, другое разрешение). Пользователь может выбирать, какой дисплей он хочет использовать при входе в систему. Однако в самой типичной ситуации используется только один дисплей (с номером 0).

“Экран” ("screen") по отношению к X-серверу означает то изображение, которое вы видите, когда запускаете X Window. “Экранов” тоже может быть несколько, так же, как и “дисплеев”. Дополнительные “экраны" используются в случае “дисплеев с многими экранами”. Фактически имеется возможность использовать один X-сервер на несколько компьютеров.

Один и тот же X-сервер может одновременно использоваться многими приложениями. Для того, чтобы обеспечить одновременную работу различных приложений и уменьшить объем информации, передаваемой по сети, сервер для каждого из приложений хранит данные об используемых приложением шрифтах, информацию о цвете и многое другое. Эти данные в совокупности называются ресурсами сервера. Клиенты имеют возможность ссылаться на эти ресурсы с помощью кодов ресурсов, сокращая тем самым сетевой трафик и облегчая доступ к данным для различных процедур.

Существует много различных менеджеров окон, так что любой пользователь может построить собственную графическую среду, использовав любой набор ее составных частей по своему выбору. Однако здесь возникают проблемы, связанные с большим разнообразием подходов к взаимодействию с пользователем, которые применяются в разных программах.

Некоторые приложения, работающие в графическом режиме, используют широко распространенные библиотеки графических элементов типа Qt или GTk, а другие строятся на основе собственных наборов графических элементов. Наборы графических элементов обычно не являются взаимозаменяемыми, поскольку используют различные программные интерфейсы. С другой стороны, не существует никаких ограничений или требований для разработчиков приложений как в части употребления тех или иных наборов графических элементов, так и в части, например, способов запуска программ (через меню, командную строку или набором определенной комбинации клавиш). Это, естественно, вызывает трудности в работе пользователей. Но более существенно то, что это вызывает нерациональные траты ресурсов компьютера. Если 6 разных приложений используют 6 разных библиотек графических элементов, то мы вынуждены хранить эти 6 библиотек на диске, так и загружать их в память, хотя Linux (и UNIX вообще) поддерживают возможность использования динамически разделяемых библиотек.

Эти недостатки отсутствуют в так называемых интегрированных графических средах или оболочках. В таких средах все основные компоненты графического интерфейса, а также множество специально разработанных приложений, строятся на основе единой графической библиотеки и единого подхода к организации интерфейса. При этом графические среды типа KDE, GNOME и т.д. строятся по тем же принципам, которые были кратко описаны выше, и не замещают перечисленные выше компоненты системы X Window, а расширяют и дополняют их. KDE, например, использует библиотеку графических функций Qt и собственный менеджер окон kwm, который управляет поведением всех окон. Кроме того, KDE дополняет Qt своей особой библиотекой (kdelibs) для выполнения таких задач, как создание меню, диалоговых окон или окон сообщений, организации взаимодействия программ, печати, и других задач.

**Запуск системы X Window**

Существует два основных варианта запуска графического интерфейса пользователя в системе Linux. В первом варианте X-сессия запускается менеджером дисплея (X Display Manager , DM), например, xdm, после чего пользователь получает возможность войти в систему (логироваться) непосредственно в графическом режиме. Во втором варианте пользователь вначале входит в систему в текстовом режиме, а потом запускает X-сессию с помощью программы xinit (чаще всего для этого используется скрипт startx, который является просто оболочкой для программы запуска графического режима xinit). В любом случае система X Window запускается с правами суперпользователя, поскольку ей требуется доступ к аппаратным устройствам.

Если вместо startx выполнить команду xinit из командной строки, вы увидите почти пустой рабочий стол с единственным окном терминала (рис. 3).

Изображение выглядит как багет, Прямоугольник, снимок экрана, текст

Автоматически созданное описание

Рис. 3. Окно xterm как единственный клиент X-сервера

Поскольку менеджера окон нет (еще не запущен), вы ничего не можете сделать с этим окном (например, переместить, изменить размер и т. д.), но вы можете в этом окне запустить другие программы, в том числе и менеджер окон. Наберите, например, команду /usr/X11R6/bin/fvwm или /usr/X11R6/bin/twm (один из этих оконных менеджеров обычно по умолчанию установлен). После этого вид экрана несколько изменится (рис. 4) , вы сможете перемещать окна (обычным способом, захватывая мышкой заголовок окна), а по щелчку левой кнопкой по пустому полю рабочего стола получите выход в меню, посредством которого можно запустить еще один экземпляр xterm или любую другую программу (на рисунке 4 вы видите два запущенных окна терминала и окно программы gimp).

Изображение выглядит как текст, снимок экрана, Прямоугольник, дисплей

Автоматически созданное описание

Рис. 4. Запущен менеджер окон twm и программа gimp

Если остановиться на таком способе вызова графического интерфейса, то каждый раз при его запуске придется повторять одну и ту же последовательность команд (не считая других минусов этого метода). Естественно, что пользователю стоит воспользоваться возможностью создания скрипта .xinitrc в домашнем каталоге пользователя для автоматизации этой рутинной работы.

Ниже приведен пример скрипта .xinitrc, который запускает часы, несколько терминалов и оставляет менеджер окон в качестве "последнего" клиента.

#!/bin/sh

xrdb-load $HOME/.Xresources

xsetroot-solid gray &

xclock-g 50x50-0+0 -bw 0 &

xload-g 50x50-50+0 -bw 0 &

xterm-g 80x24+0+0 &

xterm-g 80x24+0-0 &

twm

Важно отметить, что программы, запускаемые из .xinitrc, должны запускаться в фоновом режиме, если только они не завершаются немедленно. Иначе эти программы будут препятствовать запуску других команд. Однако одна из запущенных программ (обычно менеджер окон или эмулятор терминала) должна выполняться не в фоновом режиме, а на переднем плане, чтобы работа скрипта не завершалась (завершением работы этой программы пользователь сообщает программе xinit, что закончил работу, и что сама программа xinit должна завершиться). В приведенном примере, если менеджер окон правильно сконфигурирован, то для завершения работы в X-сессии достаточно выбрать команду **Exit** в меню менеджера twm, которое появляется по щелчку правой кнопкой по пустому полю рабочего стола.

Отметим, что скрипт startx вызывает команду xinit после формирования нужных значений аргументов командной строки для нее. Для startx существует и man-страница, в которой говорится, как этот скрипт создается.

Итак, мы вкратце рассмотрели, как организовать запуск графического режима из текстового. Если запуск графического режима отлажен и многократно испробован, можно организовать его автоматическую загрузку при включении компьютера. Для этого используется программа, которая называется менеджером дисплея xdm. Для того, чтобы запускать xdm при загрузке ОС, надо отредактировать файл /etc/inittab. В этом файле имеется строка вида

id:3:initdefault:

определяющая уровень запуска по умолчанию. Замените эту строку строкой следующего вида:

id:5:initdefault:

Такое изменение заставляет Linux при запуске переходить на 5-й уровень. А в конце того же файла /etc/inittab обычно прописана строка

x:5:respawn:/usr/bin/X11/xdm–nodaemon,

которая означает, что на этом уровне запуска должен запускаться менеджер дисплея xdm.

Имейте в виду, что команда respawn в только что приведенной строке из файла /etc/inittab означает, что при попытках перезапуска системы будет происходить перезапуск менеджера дисплея. В частности, нажатие комбинации <Ctrl>+<Alt>+<Del> будет повторно запускать систему с той же конфигурации. Поэтому если вы после установки xdm будете как-то менять системные настройки и в результате ошибочных действий нарушите хрупкое равновесие системы X Window, вы попадете в очень затруднительную ситуацию. Именно поэтому включать автоматическую загрузку графического режима при запуске системы стоит только после того, как процедуры его запуска отлажены и опробованы.

В случае использования xdm пользователь при входе в систему сразу попадает в графическую среду, и нет необходимости специально запускать графический интерфейс командой startx. При этом сохраняется возможность переключиться в текстовую консоль, набрав <Ctrl>+<Alt>+<F#> (где # заменяют номером консоли в диапазоне 1 - 6), а потом вернуться обратно в графическую среду, используя комбинацию <Ctrl>+<Alt>+<F7>.

**Взаимодействие X-сервера и приложений**

Попробуем выполнить несколько практических упражнений, которые помогут разобраться с основами взаимодействия **X-сервера** и приложений. Как правило, на рабочей станции уже запущен экземпляр (с номером **:0**) **X-сервера**. Запустим еще один экземпляр командой (команда должна запускаться от имени суперпользователя):

X :1

или командой

X localhost:1

В результате на одной из виртуальных консолей будет запущен экземпляр **X-сервера**. На этой консоли вы увидите черный (либо серый) экран с крестиком вместо указателя мыши. Теперь перейдите на свободную консоль и наберите команду:

xclock -display :1 &

Перейдите на консоль в которой запущен **X-сервер**. Вы должны увидеть в левом верхнем углу аналоговые часы. Программа **xclock** в данном примере является клиентом для **X-сервера**. С помощью ключа **-display** мы указали программе, с каким **X-сервером** ей нужно взаимодействовать, то есть какой **X-сервер** выведет на устройство вывода (монитор) изображение программы **xclock**.

**X-сервер**, который запускается по умолчанию при загрузке системы, имеет порядковый номер 0. То есть команда **xclock -display :0** выведет часы на седьмую виртуальную консоль.

Вернемся к нашим часам на втором экземпляре **X-сервера**. Вы видите что у программы нет привычной окантовки окна с кнопками свернуть, развернуть, закрыть. Дело в том, что их рисует не **X-сервер**, а другой процесс, который называется **менеджер окон**. **Менеджер окон** также занимается масштабированием и перемещением изображения программ. Такое разделение функций также повышает гибкость всей графической подсистемы **X Window**, поскольку вы можете выбирать, какой менеджер окон использовать для прорисовки рамок приложений.

Чтобы на нашем втором экземпляре **X-сервера** запустить менеджер окон по умолчанию, выполните команду **x-window-manager –display=0:1** и снова вернитесь на консоль где запущен второй **X-сервер**. Теперь вы видите вокруг часов привычную рамку и верхнюю полоску окна с тремя привычными кнопками. Окно можно масштабировать и перемещать по экрану, используя “мышь”, а также закрыть его нажав на привычный “крестик”.

Как уже было сказано, **X-сервер** работает с устройствами ввода/вывода информации. Основной конфигурационный файл, в котором хранятся настройки **X-сервера,** называется **/etc/X11/xorg.conf**. Хотя во многих современных настольных дистрибутивах этот файл пустой, так как за настройку оборудования отвечает другая подсистема, тем не менее **X-сервер** продолжает читать конфигурацию из этого файла.

**Экранный менеджер**

Экранный менеджер (Display Manager , DM) или менеджер входа — графический экран, который отображается в конце процесса загрузки вместо стандартного приглашения командной строки. Экранный менеджер представляет собой экран ввода имени пользователя и пароля для входа в систему. Существует множество экранных менеджеров, как и окружений рабочего стола. Практически все экранные менеджеры можно настраивать, изменяя их стиль и поведение.

Экранные менеджеры подразделяются на консольные и графические. Консольные:

* CDM — ультра-минималистичный, но полностью функциональный менеджер входа, написанный на Bash;
* Console TDM — расширение для xinit, также написанное на Bash;
* Nodm — минималистичный экранный менеджер для автоматического входа;
* Ly — экспериментальный менеджер входа.

Графические экранные мереджеры:

* GDM — экранный менеджер GNOME;
* LightDM — независимый от среды рабочего стола экранный менеджер, основанный на WebKit;
* LXDM — экранный менеджер LXDE. Может быть использован отдельно от среды рабочего стола LXDE;
* SDDM — экранный менеджер, основанный на QML. Продолжение KDE4 kdm, рекомендуется для Plasma 5 и LXQt;
* XDM — экранный менеджер с поддержкой XDMCP;
* Fly-DM - используется в AstraLinux.

Чтобы включить экран входа, запустите соответствующую службу. Например для SDDM включите sddm.service. Это должно работать без дополнительных настроек. Если нет, возможно, вам придется удалить символическую ссылку default.target, чтобы указать на graphical.target файл по умолчанию. После включения SDDM, в /etc/systemd/system/ должен быть установлен symlink display-manager.service. Возможно, вам придется использовать --force для переопределения старых символических ссылок:

$ file /etc/systemd/system/display-manager.service

/etc/systemd/system/display-manager.service: symbolic link to /usr/lib/systemd/system/sddm.service

Чтобы иметь возможность проверить статус вашей сессии, вы можете использовать loginctl. Все действия, такие, как перевод системы в ждущий режим или монтирование внешних устройств, будут работать "из коробки":

$ loginctl show-session $XDG\_SESSION\_ID

Большинство экранных менеджеров получают список доступных сеансов из каталога /usr/share/xsessions/. Он содержит стандартные файлы .desktop для каждого экранного/оконного менеджера. Чтобы добавить/удалить записи в список сеансов вашего экранного менеджера, создайте/удалите соответствующий файл .desktop в /usr/share/xsessions/. Типичный файл .desktop выглядит примерно так:

[Desktop Entry]

Name=Openbox

Comment=Log in using the Openbox window manager (without a session manager)

Exec=/usr/bin/openbox-session

TryExec=/usr/bin/openbox-session

Icon=openbox.png

Type=Application

Можно использовать ~/.xinitrc для запуска xinitrc в качестве сеанса. Установите xinitrc в качестве сеанса в настройках вашего экранного менеджера (установите xinit-xsessionAUR для запуска xinitrc в качестве сеанса) и убедитесь, что файл ~/.xinitrc является исполняемым.

Для экранных менеджеров, которые используют AccountsService, язык для пользовательского сеанса может быть установлен путем редактирования:

/var/lib/AccountsService/users/$USER

[User]

Language=your\_locale

Выйдите из системы, а затем снова войдите, чтобы изменения вступили в силу.

Настройка автоматического входа в Linux чуть различается в зависимости от того, какой именно менеджер отображения вы используете. Самые популярные это **GDM** и **LightDM**.

Настройку входа в Linux без пароля нужно начать с определения, какой именно запущен Display Manager. Для этого выполните команду:

systemctl status display-manager.service

Для включения автоматического входа с GDM, добавьте в файл /etc/gdm/custom.conf следующие строки (замените имя\_пользователя на ваше собственное имя в системе):

sudo gedit /etc/gdm/custom.conf

# Включение автоматического входа для пользователя

[daemon]

AutomaticLogin=имя\_пользователя

AutomaticLoginEnable=True

**Оконные менеджеры**

**Оконный менеджер** (window manager, WM) - это программа, которая управляет окнами приложений в системе, а именно: осуществляет позиционирование окон на экране, отвечает за изменение их размера, фокусировку и прочее. При этом оконный менеджер работает поверх существующей оконной системы. В простейшем случае, оконный менеджер просто управляет окнами и взаимодействием с ним оборудования ввода, и не делает с ними ничего другого. При этом нагрузка за отрисовку ложится на центральный процессор (так называемая **программная отрисовка**). Некоторые оконные менеджеры, помимо управления окном, могут отрисовывать тени на его гранях, добавлять различные анимации, плавности, полупрозрачности и так далее. В этом случае, оконный менеджер является **композитным**. Одной из важнейших (но необязательных) функций композитного оконного менеджера является задействование для отрисовки окна возможностей видеокарты, как правило посредством **OpenGL**. Тем самым увеличивая качество отрисовки и значительно снижая нагрузку на центральный процессор. Функционал композитного менеджера может быть как встроенным в оконный менеджер, так и являться отдельной программой. Часто отрисовку окна через композитный оконный менеджер называют просто - **композитингом**. Пример из Windows 7: когда в ней отключены эффекты **Aero**, отрисовка ведётся силами центрального процессора. Нагрузка на видеокарту меньше, однако при воспроизведении видео на экране появляются артефакты, известные как тиринг (когда кадры меняются слишком быстро и посередине видно прозрачную мерцающую полосу).

Когда эффекты Aero включены - отрисовка ведётся силами видеокарты. Что становится очевидным, так как появляются анимации появления и сворачивания окна, полупрозрачности и так далее.

Однако, композитинг имеет и обратную сторону. При отрисовке рабочего стола силами видеокарты частота кадров в секунду синхронизируется с частотой монитора (как правило 60 кадров в секунду, что соответствует стандартным 60 герцовым мониторам), поэтому в играх частота будет несколько ниже, так как будет идти двойная синхронизация кадра. В тяжёлых случаях - производительность игры может упасть вдвое. Поэтому часто можно встретить рекомендации отключать графические эффекты при запуске игр.

В данный момент доминирующей графической подсистемой в Линуксе является Xorg (иксы). И работа оконного менеджера в ней точно такая, как описана выше. При этом функционал композитинга не был изначально в Xorg, и его реализовали значительно позже, потому композитный менеджер работает там как-бы сбоку. В общем случае, получается весьма толстый бутерброд различных слоёв, через который происходит отрисовка картинки.

Первым широко известным оконным менеджером композитинга для X был Xfwm. Менеджер окон Compiz предоставляет ускоренный 3D-композитинг для платформы Linux. KDE KWin также поддерживает композитинг.

Сейчас на смену Xorg идут две графические подсистемы - Wayland и Mir. Первый не привязан к какому-либо дистрибутиву или графической оболочке, в то время как второй разрабатывается для Ubuntu и её графической оболочки Unity, и уже работает в мобильной редакции Ubuntu. Что касается Wayland - в нём отсутствуют привычные понятия оконного и композитного менеджера. В нём есть только композитор, который и производит все операции над окнами, без лишних прослоек. Отрисовка приложения при этом ложится на программный инструмент (тулкит), на котором оно написано, например Qt или GTK. Это так называемая отрисовка на стороне клиента. В случае же если отрисовка окна идёт непосредственно в Wayland-композиторе, то это называется отрисовкой на стороне сервера. В случае отрисовки на стороне клиента, заголовок окна приложения, его внешний вид и прочее целиком ложится на разработчика приложения. В результате может случиться так называемый "эффект Windows": если в приложении не заложен функционал изменения размера окна, то окно приложения будет всегда фиксированного размера. Пример окна с декорацией заголовка на стороне клиента (обратите внимание на кнопки управления приложением в заголовке окна):

Изображение выглядит как текст, снимок экрана, программное обеспечение, Значок на компьютере

Автоматически созданное описание

Этот функционал реализован в GNOME. В KDE же используется отрисовка на стороне сервера, в результате все окна будут иметь одинаковый заголовок и легко меняться в размере:

Изображение выглядит как текст, снимок экрана, программное обеспечение, Операционная система

Автоматически созданное описание

Кстати, если запустить приложение с CSD (Client-Side Decorations) в оконном менеджере, не поддерживающем отрисовку на стороне клиента - приложение получит два заголовка:

Изображение выглядит как текст, программное обеспечение, снимок экрана

Автоматически созданное описание

CSD, кстати говоря, оказался довольно удачным решением. Настолько удачным, что его взяла к себе Apple.

Wayland уже работает в автомобильных ОС, в мобильных операционках Tizen и SailfishOS, и много где ещё. Философия Wayland - "Каждый выводимый кадр должен быть идеальным". И это действительно так. Отрисовка в Wayland по качеству превосходит таковую в Xorg (к примеру в Wayland в принципе невозможен тиринг), плюс ко всему - в Wayland сильно затруднено создание кейлоггеров (перехватчиков клавиатурных нажатий), что положительно сказывается на безопасности. Однако Wayland пока не поддерживается фирменными проприетарными драйверами Nvidia и AMD, что сильно затрудняет его введение по умолчанию в дистрибутивах Linux.

Наряду с композитными существуют стековые, фреймовые и динамические менеджеры окон:

* **стековые** (также *плавающие*) оконные менеджеры следуют традиционной метафоре рабочего стола, которая используется в коммерческих операционных системах Windows и OS X. Окна отображаются подобно листкам бумаги на столе, накладываясь и перекрывая друг друга.
* **фреймовые** оконные менеджеры располагают окна на экране в виде плиток (фреймов) так, что они не перекрывают друг друга. Как правило, фреймовые оконные менеджеры подразумевают активное использование клавиатуры для управления окнами, и имеют слабую поддержку мыши (либо не имеют ее вовсе). Фреймовые оконные менеджеры могут предлагать набор стандартных расположений фреймов или позволять задавать их вручную.
* **динамические** оконные менеджеры позволяют динамически переключаться между двумя режимами — стековым и фреймовым.