СОДЕРЖАНИЕ

ЛАБОРАТОРНАЯ РАБОТА № 1

Установка операционной системы Linux

в контейнер виртуальной машины VirtualBox………………….. ..……………….. 4

ЛАБОРАТОРНАЯ РАБОТА № 2

Файлы и каталоги в системе Linix…………………… …………………………….21

ЛАБОРАТОРНАЯ РАБОТА № 3

Интерпретаторы команд и командные процедуры

в системе Linix………………………………………………… …………………….34 ЛАБОРАТОРНАЯ РАБОТА № 4

Доступ к файловой системе через интерфейс системных вызовов..……………..45

ЛАБОРАТОРНАЯ РАБОТА № 5

Работа с процессами в командной и программной средах

операционной системы Linix ……………..………………………………………...61

ЛАБОРАТОРНАЯ РАБОТА № 6

Создание неименованных каналов и приложений типа клиент-сервер

в системе Linux ……………………………….………………………………….....73

ЛАБОРАТОРНАЯ РАБОТА № 7

Создание именованных каналов и приложений типа клиент-сервер

в приложениях WPF ………………………..……………………………………….81

ЛАБОРАТОРНАЯ РАБОТА № 8

Создание и синхронизация потоков в системе Linux……………………………. 89

ЛАБОРАТОРНАЯ РАБОТА № 9

Семафоры стандарта Posix для синхронизации процессов и потоков

в системе Linux ……………………………..……………………………………….96

Лабораторная работа № 1

Установка операционной системы Linux в контейнер виртуальной машины VirtualBox

**1. ЦЕЛЬ И ЗАДАЧИ РАБОТЫ**

Ознакомление с операционной системой Linux и программой виртуализации VirtualBox, получение практических навыков создания виртуальных машин на базе VirtualBox с различными гостевыми Linux-системами.

**2. ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ**

**2.1. 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.

**2.2. 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.

**2.3. 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.

**2.4. Установка системы Linux в контейнер виртуальной машины**

Система Linux устанавливается либо в раздел файловой системы, либо в контейнер виртуальной машины. 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;
* для раздела с пользовательскими файлами (/home) память выделяется в соответствии с предполагаемыми объемами обрабатываемых данных;
* под область свопинга (swap) выделяется примерно 1 – 2 Гбайт.

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.

**2.5. Создание гостевой системы Debian на базе виртуального диска**

Ниже показана последовательность создания ВМ с использованием имеющегося виртуального диска (файла с расширением **\*.vdi**). Нажмите кнопку «Создать» в главном окне программы (рис. 1).

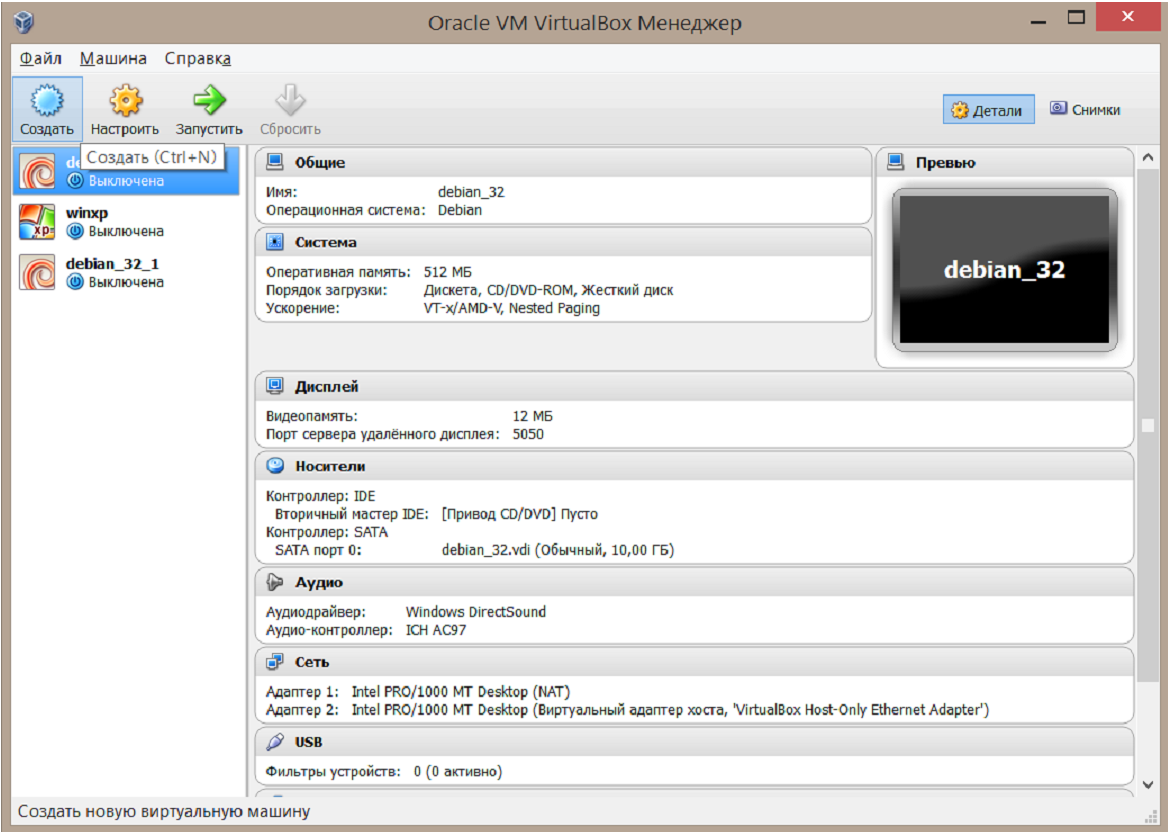


Рис 1. Кнопка "Создать" в главном окне программы VirtualBox

Укажите имя и тип будущей гостевой операционной системы (рис. 2).

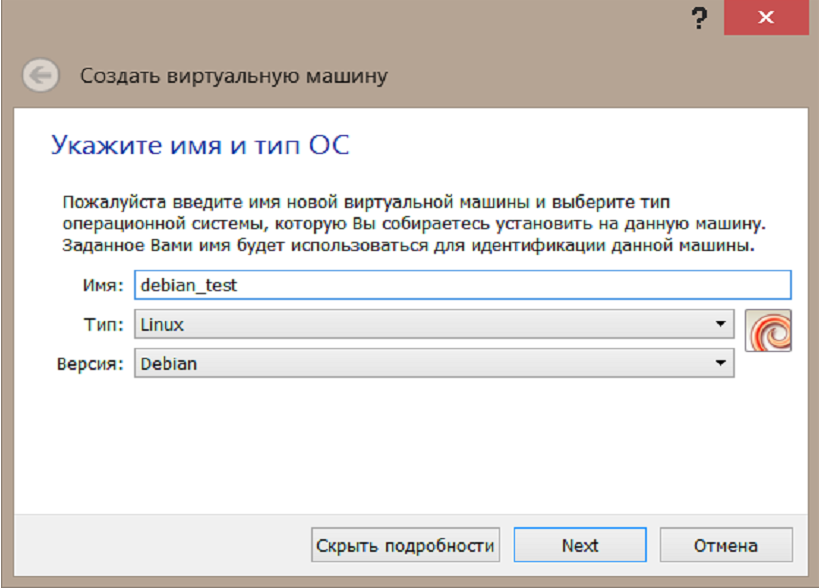


Рис 2. Экран выбора имени и типа гостевой операционной системы

Далее на экране выбора виртуального жесткого диска выберите пункт «Использовать существующий виртуальный жесткий диск» и укажите путь к файлу \*.vdi (рис. 3).

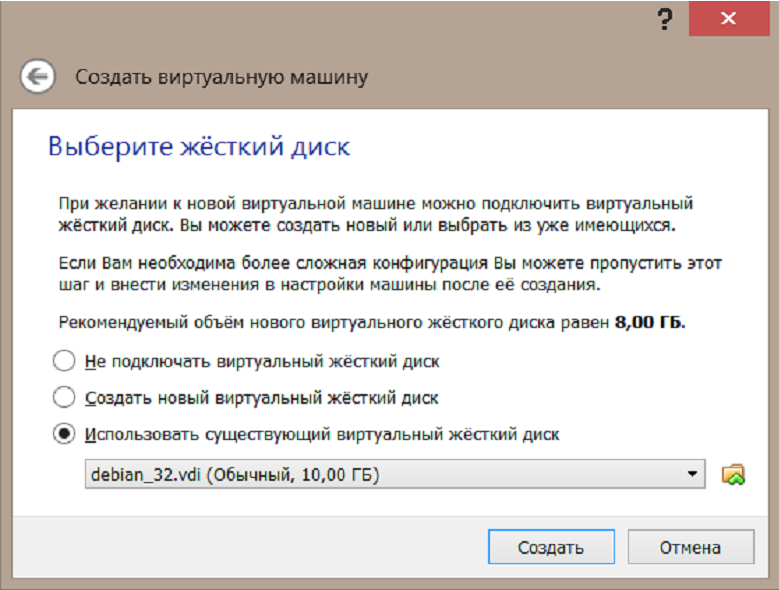


Рис 3. Экран выбора виртуального жесткого диска

Далее выполняем настройку виртуальной машины. Выберите виртуальную машину в списке доступных и нажмите кнопку «настроить» (рис. 4.).

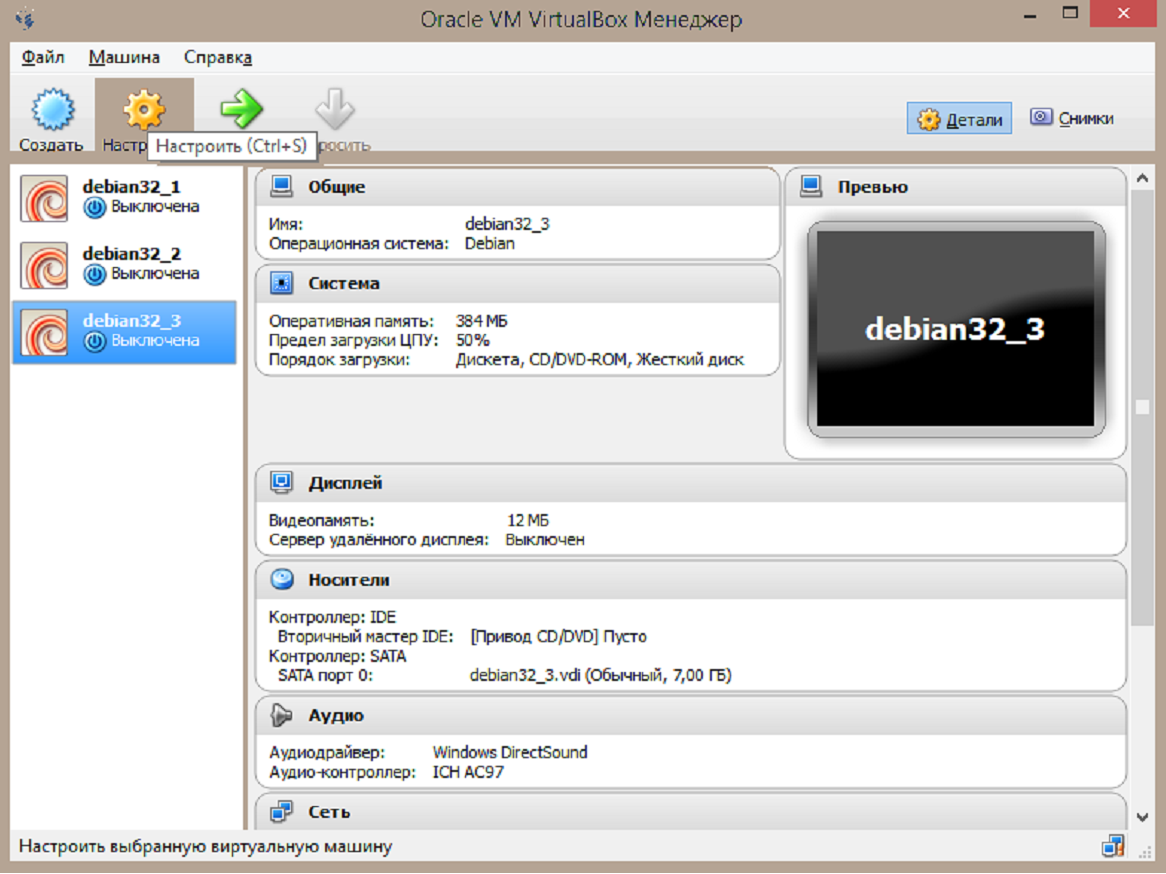


Рис 4. Расположение кнопки "настроить" в главном окне

**Е**сли устанавливается 32-битный вариант гостевой операционной системы, проверьте, включена ли поддержка PAE в настройках виртуальной машины. Для этого на вкладке «Система — Процессор» выберите опцию PAE/NX, иначе возможны ошибки при запуске виртуальной машины (рис. 5).

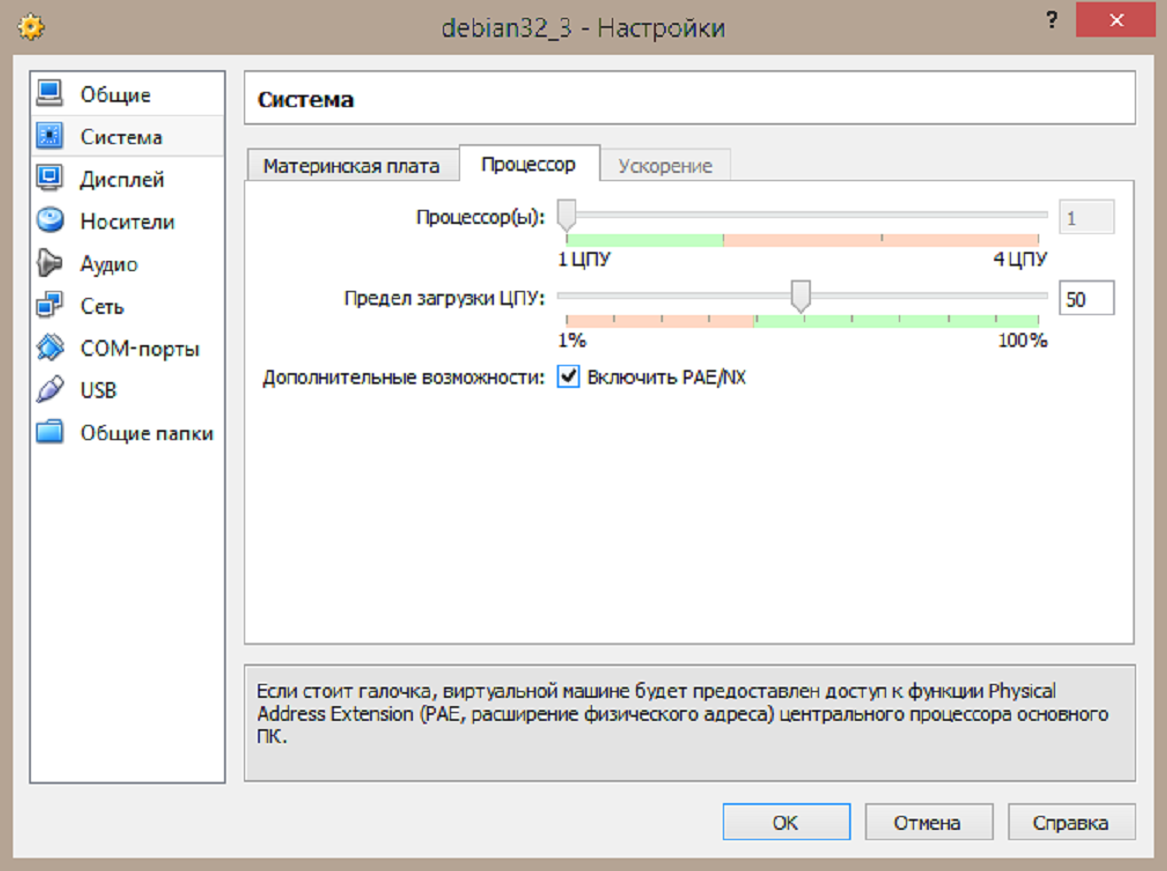


Рис 5. Пункт "Включить PAE/NX" в разделе "Система"

Для запуска созданной машины закройте окно настройки виртуальной машины и нажмите кнопку Запуск. Для входа под учетной записью студента укажите, например, логин:student, пароль:password.

Пример окна входа в систему Debian представлено на рисунке 6.

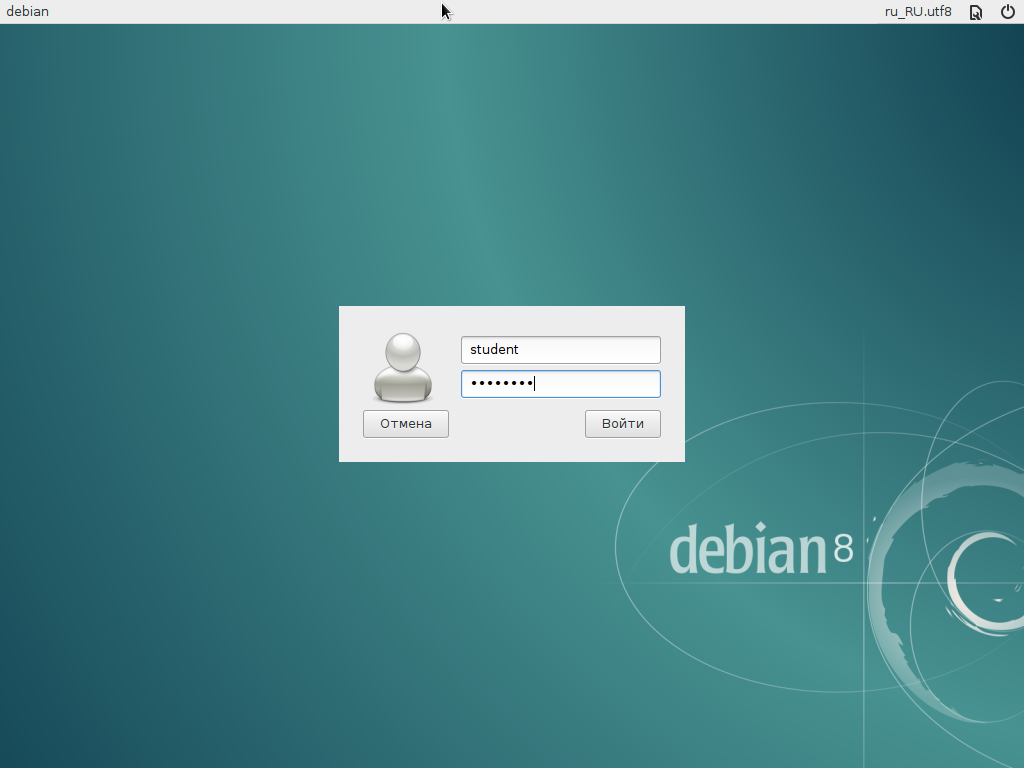


Рис. 6. Окно входа в систему Debian

**2.6. Создание гостевой системы Ubuntu 20.04 из дистрибутива**

Создадим новую виртуальную машину: Машина -> Создать. Укажем имя (например, Ubuntu-01) и тип: Linux. Версия: Ubuntu (64 bit). Нажмём кнопку Next (Далее). Укажем объем памяти, например, 2048 MB.

Далее создаем новый виртуальный диск и указываем его тип - VDI и формат хранения - динамический виртуальный жесткий диск. Это будет означать, что файл, в котором хранится вся информация виртуальной машины, изначально будет иметь минимально возможный размер, а затем будет увеличивать свой размер по мере заполнения виртуального диска.

Укажем максимальный размер виртуального жёсткого диска, например, 20 Гб (рекомендуется более 30 Гб). Здесь же можно нажать на значок справа и задать местоположение виртуального диска на хосте (если не устраивает путь, заданный по умолчанию). Далее нажимаем кнопку Создать.

Виртуальная машина создана, её имя и перечень заданных параметров появилось в списке доступных виртуальных машин в окне менеджера Oracle VirtualBox (рис. 7).

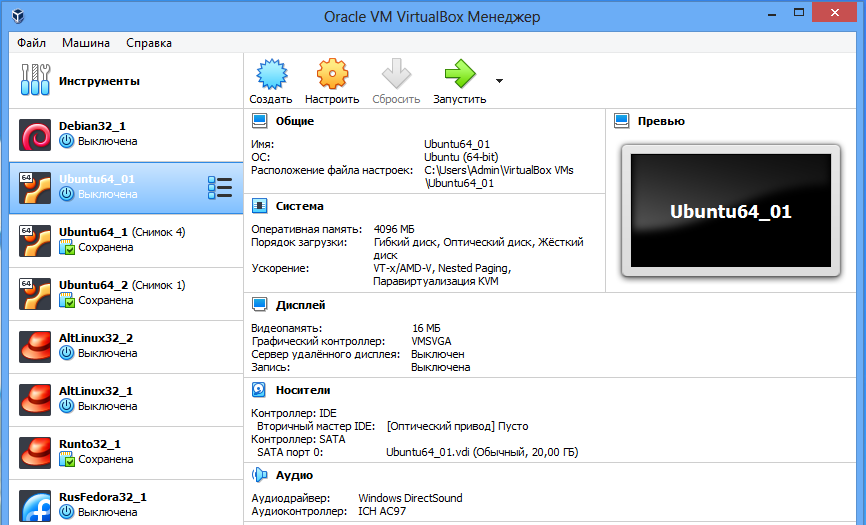


Рис. 7. Окно менеджера Oracle VirtualBox

Выделим щелчком левой кнопки мыши только что созданную виртуальную машину и нажмём кнопку Настроить. В появившемся диалоговом окне слева выберем пункт Система и на вкладке Материнская плата выбераем опцию PAE/NX. Далее на вкладке Дисплей задаем размер видеопамяти (64 или 128 Мб), коэффициент масштабирования (например, 1.50) и графический контроллер VBoxSVGA. Эти параметры для дисплея обычно подбираются экспериментально, правильность выбора определяется уже на этапе установки системы.

Далее в разделе Носители щелкаем по диску **Пусто** в дереве носителей и нажимем кнопку выбора диска (рис. 8), а из открывшегося меню выбираем команду **Выбрать образ оптического диска**. Откроется диалоговое окно выбора файлов, в котором нужно выбрать записанный на винчестер образ диска (файл ISO). Имя образа появится в поле Контроллер: IDE раздела Носители (рис. 9).

Параметры остальных разделов можно не изменять.

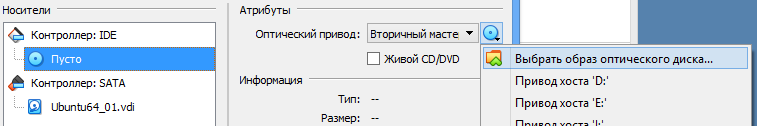


Рис. 8. Вкладка Носители

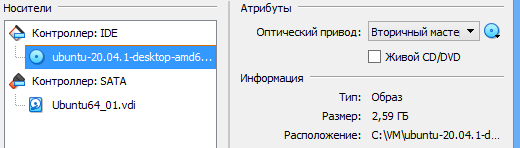


Рис. 9. Задание образа оптического диска

**2.7. Установка гостевой системы Ubuntu 20.04 из дистрибутива**

В окне менеджера Oracle VirtualBox щёлкнем левой кнопкой мыши по названию нашей виртуальной машины и нажмём кнопку Запустить. Через некоторое время появится окно начала установки (рис. 10), где мы выбираем язык и нажимаем кнопку Установить Ubuntu. Далее обычно появляется окно выбора раскладки клавиатуры, которое мы оставляем без изменений и нажимаем кнопку Продолжить.

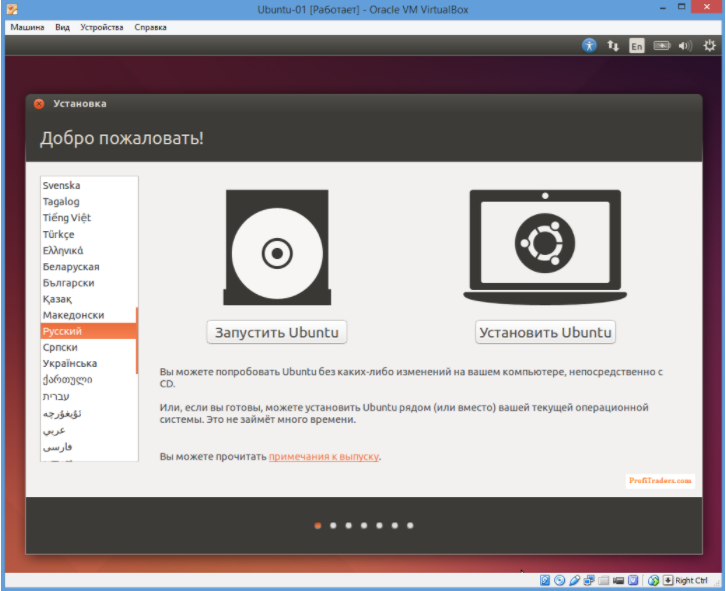


Рис. 10. Окно выбора языка

В следующем окне (рис. 11) мы откажемся от загрузки обновлений во время установки и выберем обычную установку (минимальная установка позволит съэкономить менее одного Гб на виртуальном диске).

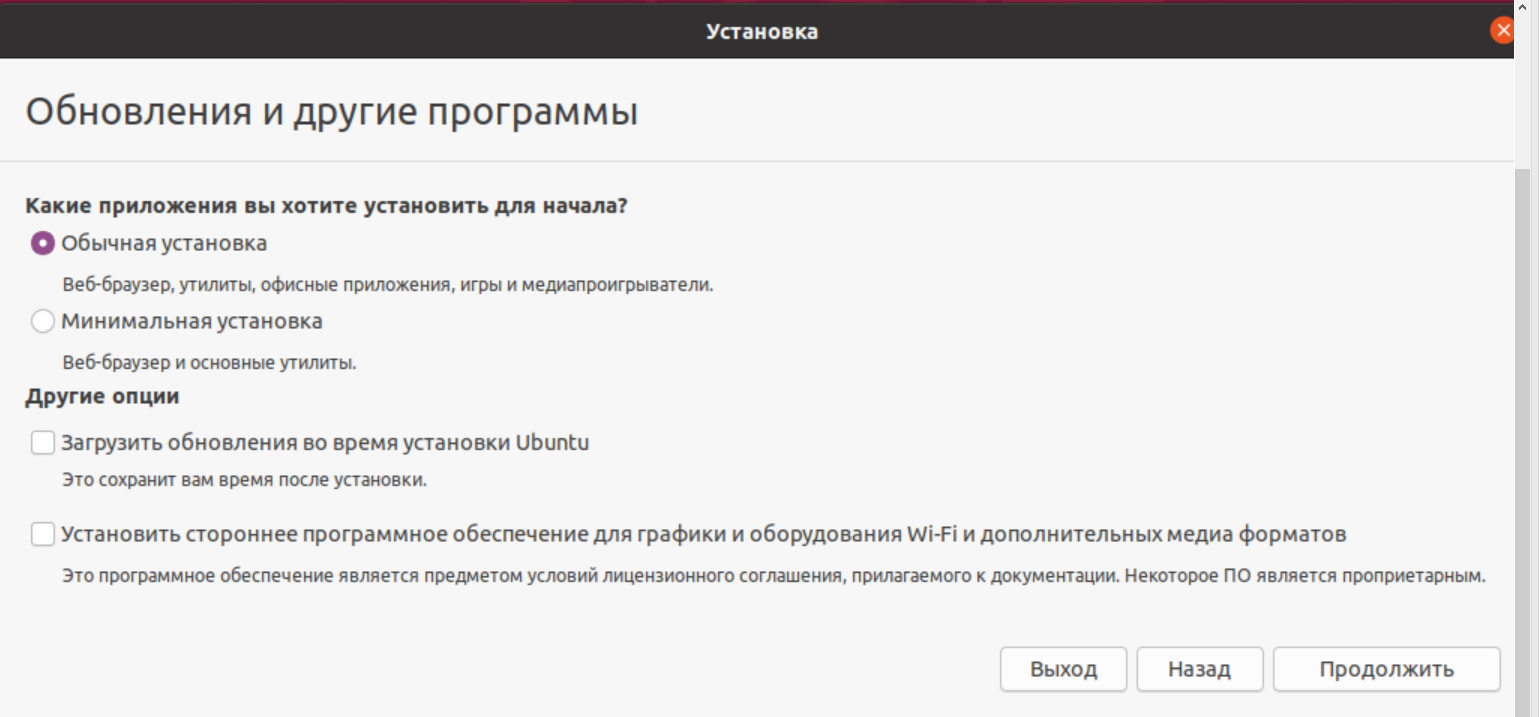


Рис. 11. Окно выбора программного обеспечения

Поскольку система устанавливается в виртуальную среду и весь диск принадлежит нашей системе, его можно стереть перед установкой (рис. 12). В данной работе создание и изменение разделов на диске не рассматривается. Нажимаем кнопку Установить сейчас.

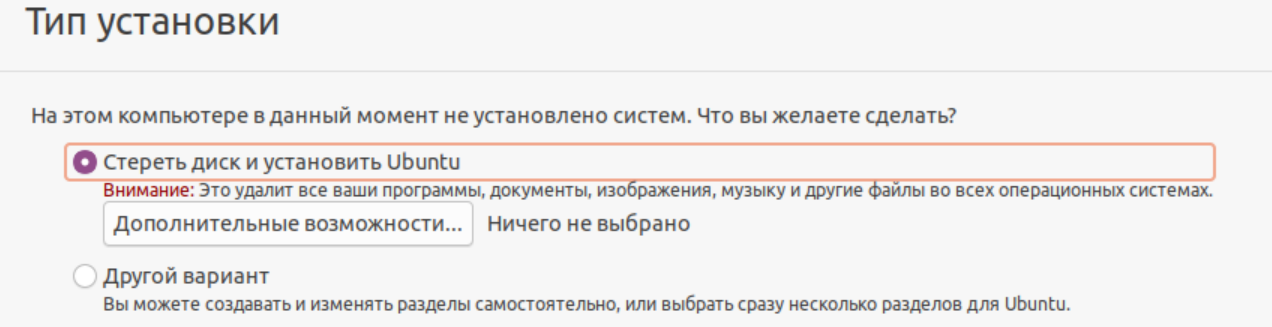


Рис. 12. Окно выбора типа установки

Появляется окно, где напоминают о начале форматирования диска. Нажимаем кнопку Продолжить. В окне Выбор часового пояса левой клавишей мыши выбираем на карте наше местоположение (появится слово Moscow), после чего нажимаем Продолжить.

Следующее окно (рис. 13) позволяет задать имя пользователя и пароль. Желательно, чтобы пароль включал в себя заглавные и строчные буквы латинского алфавита, цифры и специальные знаки, например, точку, запятую или звёздочку. Не забудьте записать пароль. Нажмём кнопку Продолжить.

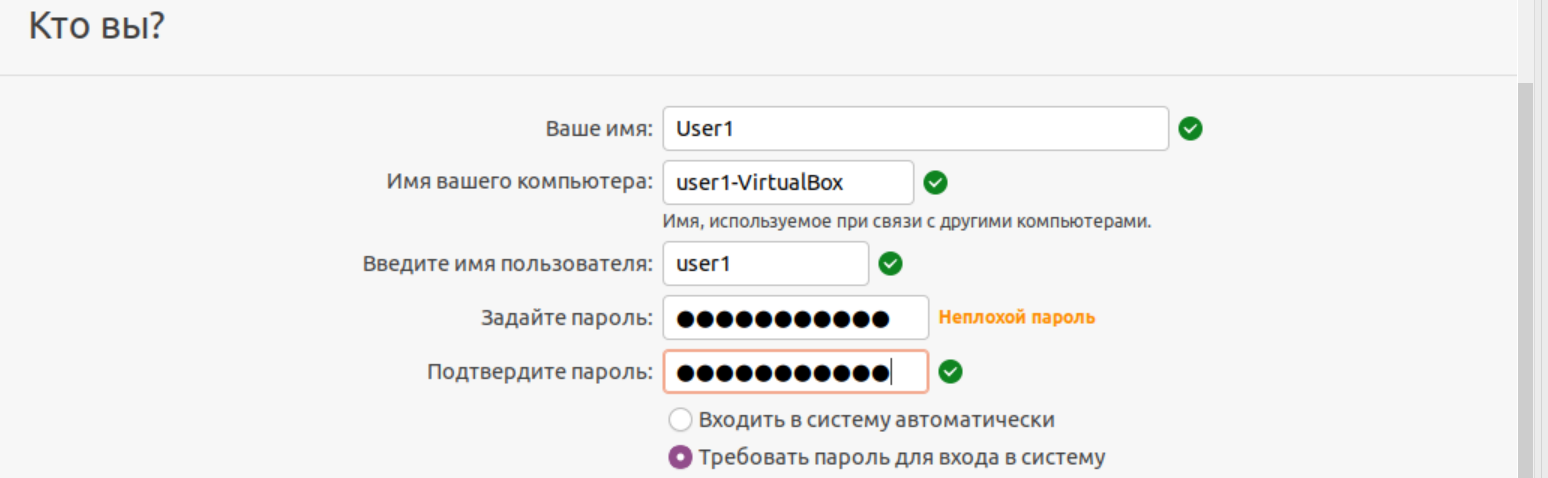


Рис. 13. Задание имени пользователя и пароля

Далее начинается процесс установки системы Ubuntu (рис. 14).

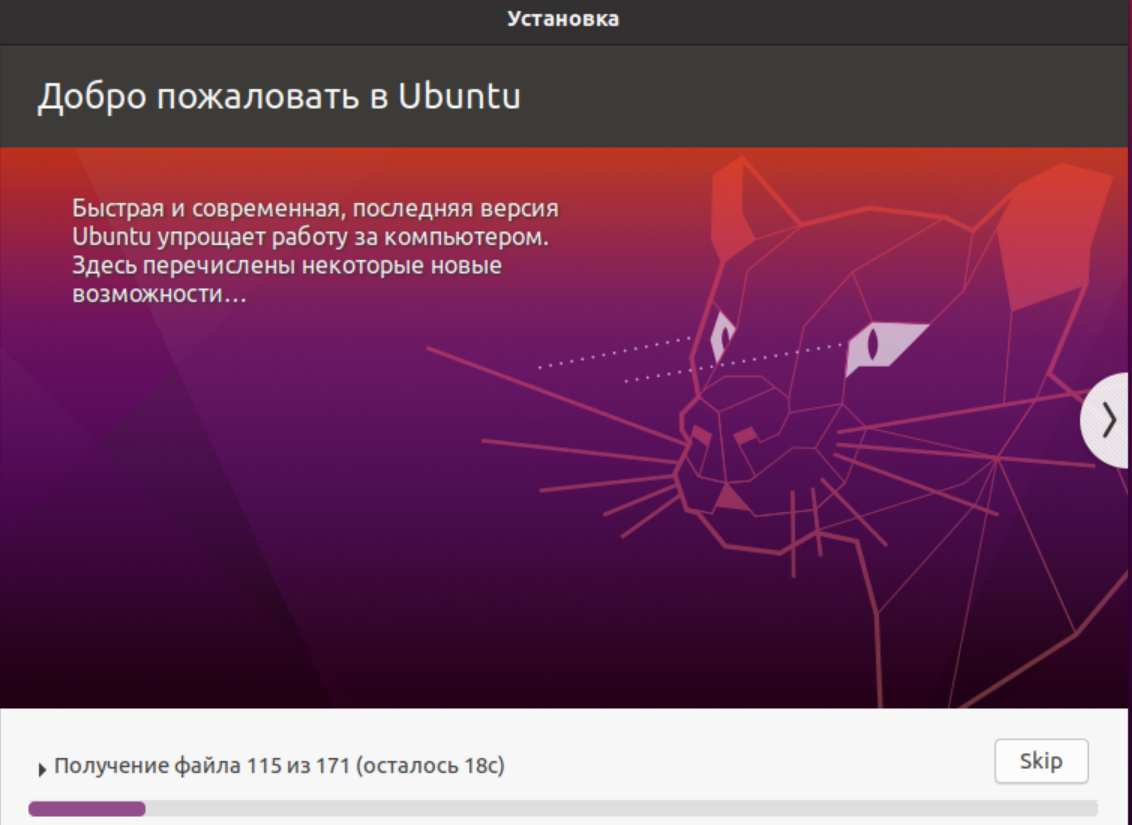


Рис. 14. Процесс установки системы Ubuntu 20.04

По завершении установки появится окно Установка завершена (рис. 15). Однако если мы нажмем кнопку Перезагрузить, то после перезагрузки у нас заново начнется установка с оптического диска (из файла ISO). Поэтому вместо перезагрузки надо выключить созданную виртуальную машину, убрать файл ISO из привода, а затем уже включить созданную виртуальную машину.

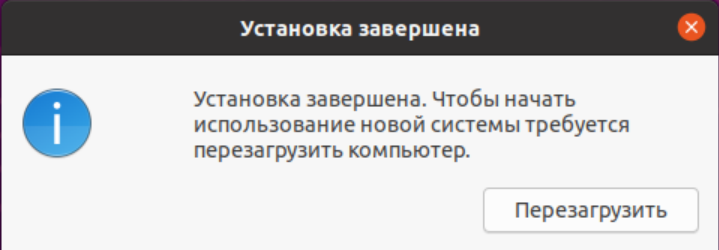


Рис. 15. Сообщение о завершении установки

После выполнения команд меню Машина->Выключить может появиться окно, показанное на рис. 16. В документации не прописана дальнейшая последовательность. Если это окно вообще не появлялось, а виртульная машина просто выключалась, то требуется запустить настройку машины (см. рис. 8), мышку устанавить на имя файлы ISO, нажатием правой клавиши открыть контекстное меню и выполнить команду извлечения дистрибутива из привода. Во-втором случае, когда окно, показанное на рис. 16, появилось, необходимо повторно выполнть команду Машина->Выключить, после чего дистрибутив (образ ISO) автоматически удалялится из привода.

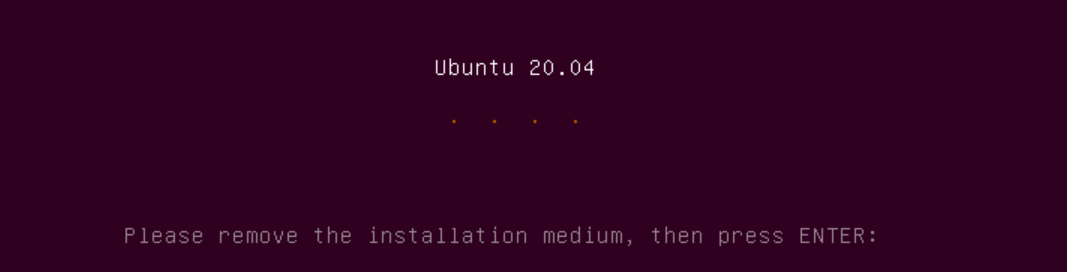


Рис. 16. Сообщение о необходимости удаления дистрибутива

В существующих рекомендациях этот этап описан следующим образом. Если на экране появится совет удалить установочный диск, то в окне виртуальной машины выполним пункт меню Устройства -> Приводы оптических дисков -> Изъять диск из привода, а затем щёлкнем мышкой по консольному экрану виртуальной машины и нажмём клавишу Enter.

Далее, когда машина выключилась, повторно нажимаем кнопку Запустить. Появляется начальное окно системы Ubuntu (рис. 17), где мы вводим логин, например, user1, и пароль.

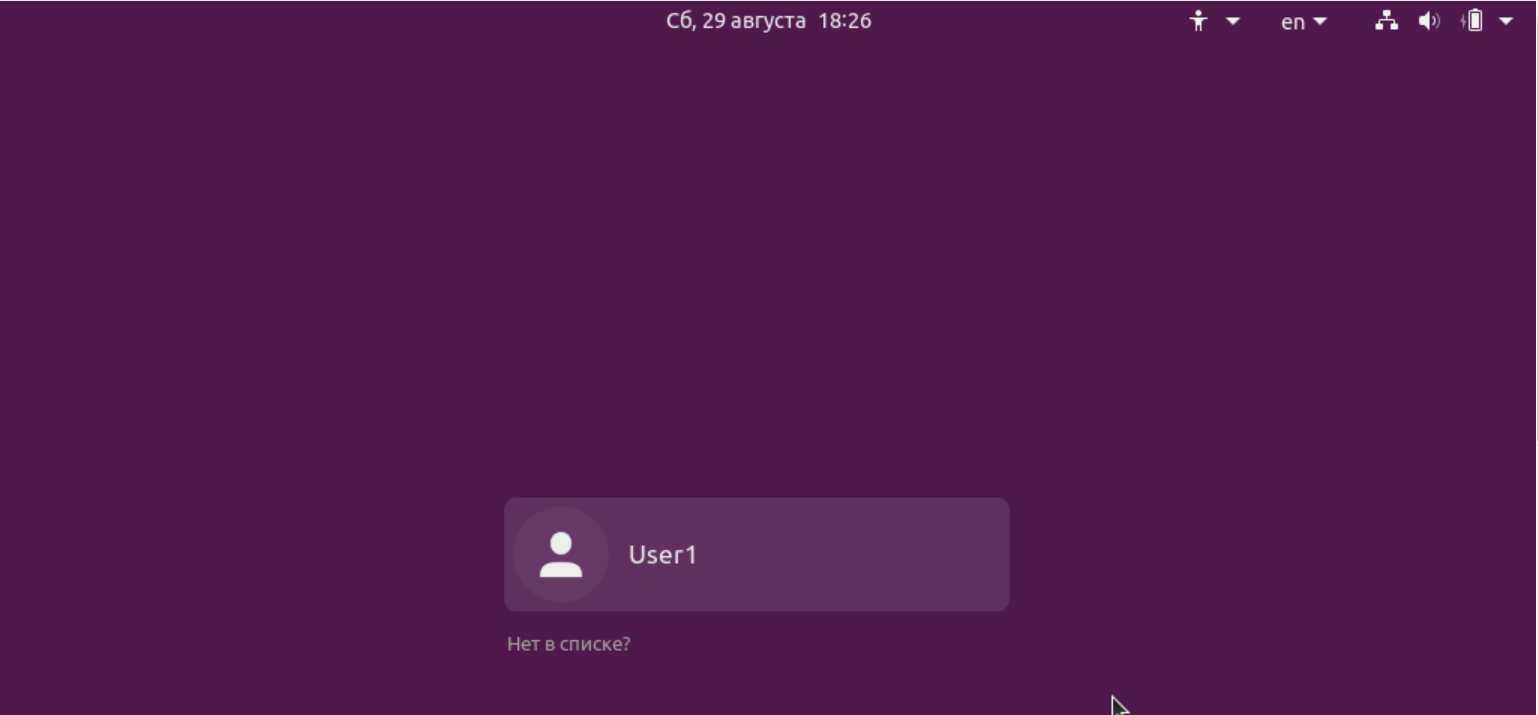


Рис. 17. Начальное окно системы Ubuntu

Далее выполняется ряд этапов, некоторые из которых можно просто пропустить (рис. 18 - 26).

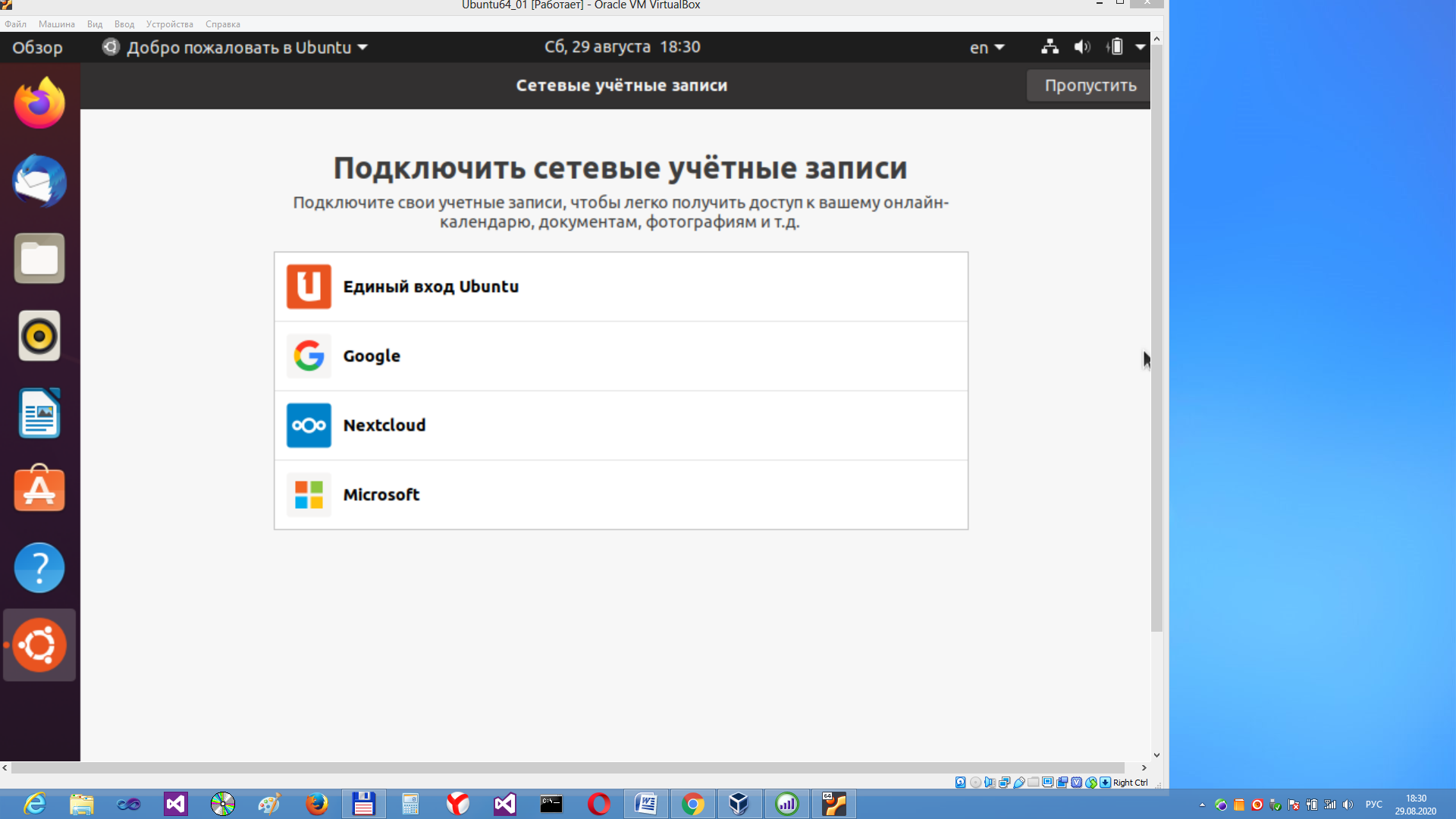


Рис. 18. Подключение сетевых учетных записей

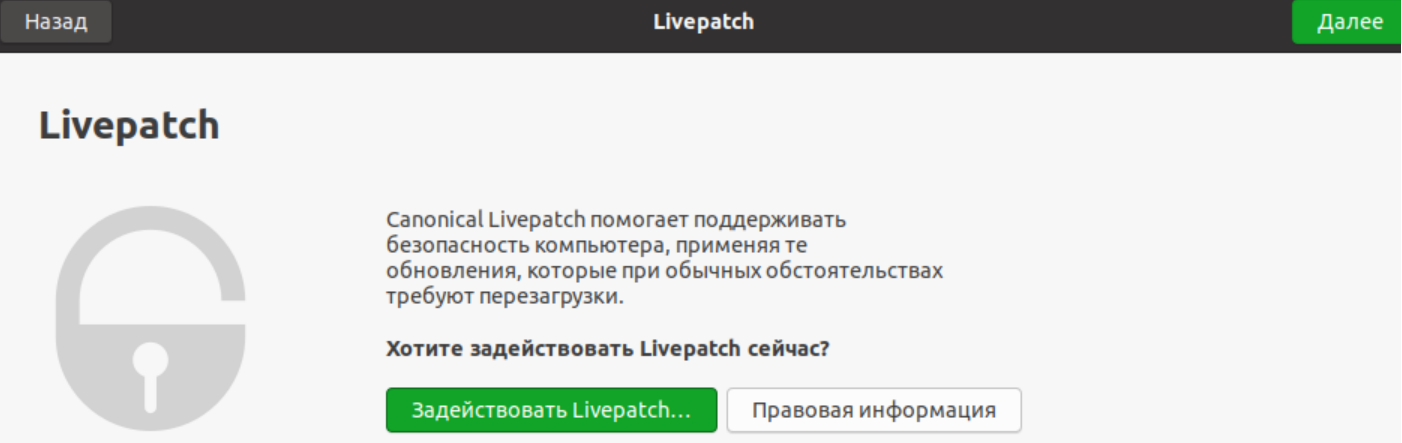


Рис. 19. Включение режима Livepatch

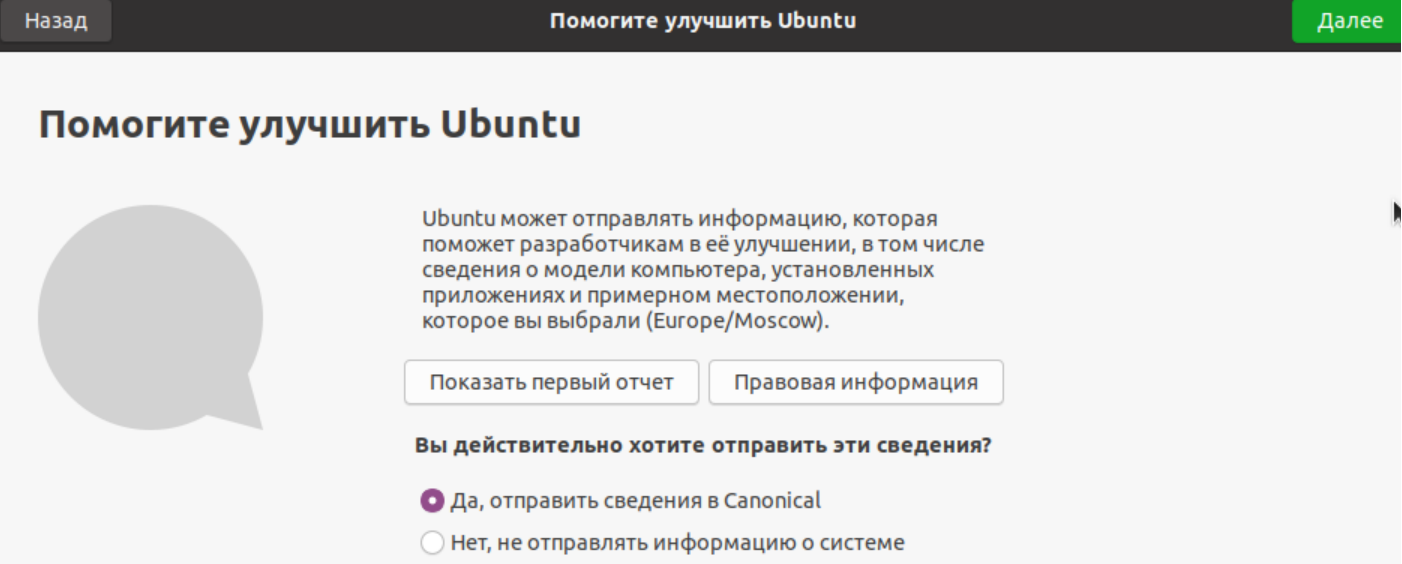


Рис. 20. Включение режима отправки информации о работе системы

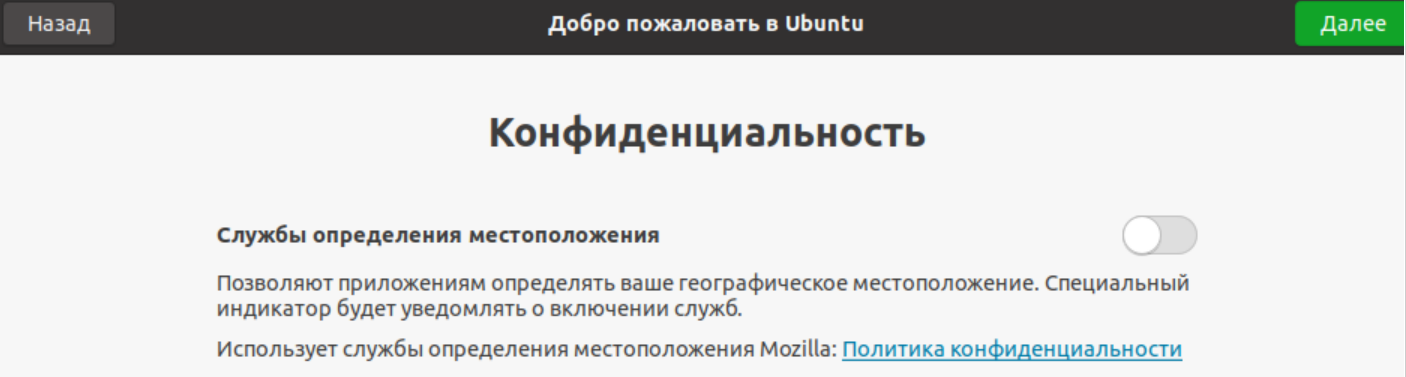


Рис. 21. Разрешение определять местоположение

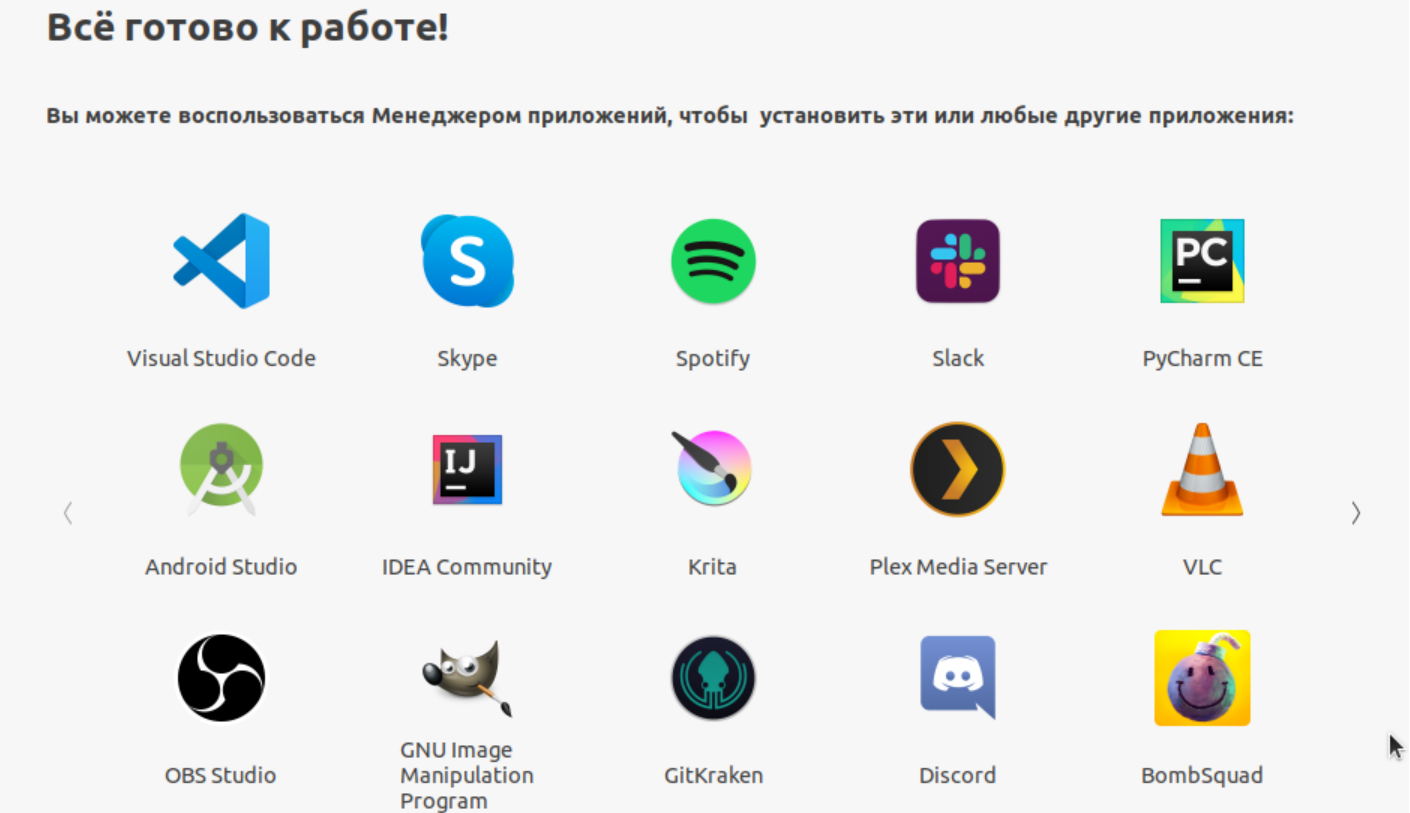


Рис. 22. Перечень наиболее часто устанавливаемых приложений

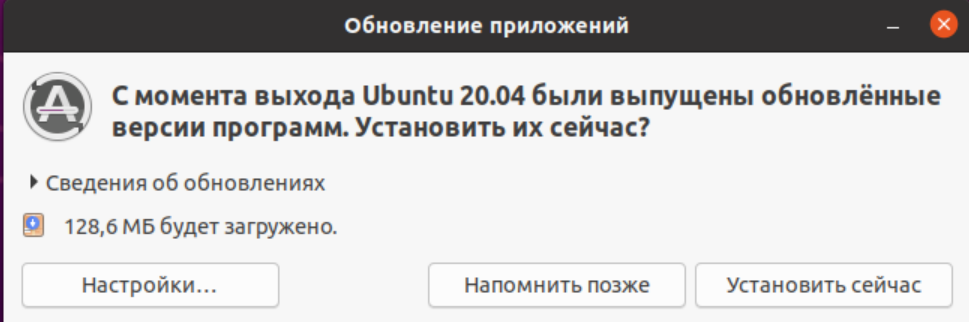


Рис. 23. Приглашение обновить установленные программы

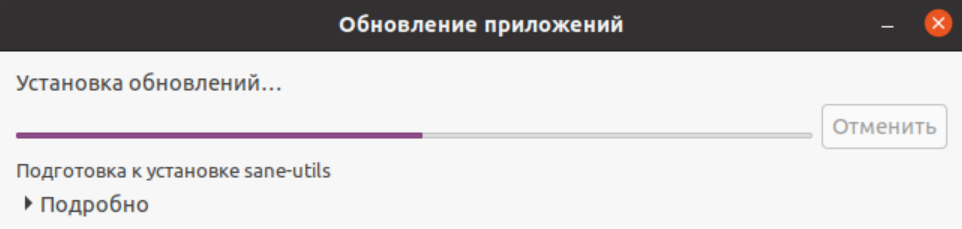


Рис. 24. Окно обновления приложений



Рис. 25. Главное окно системы Ubuntu

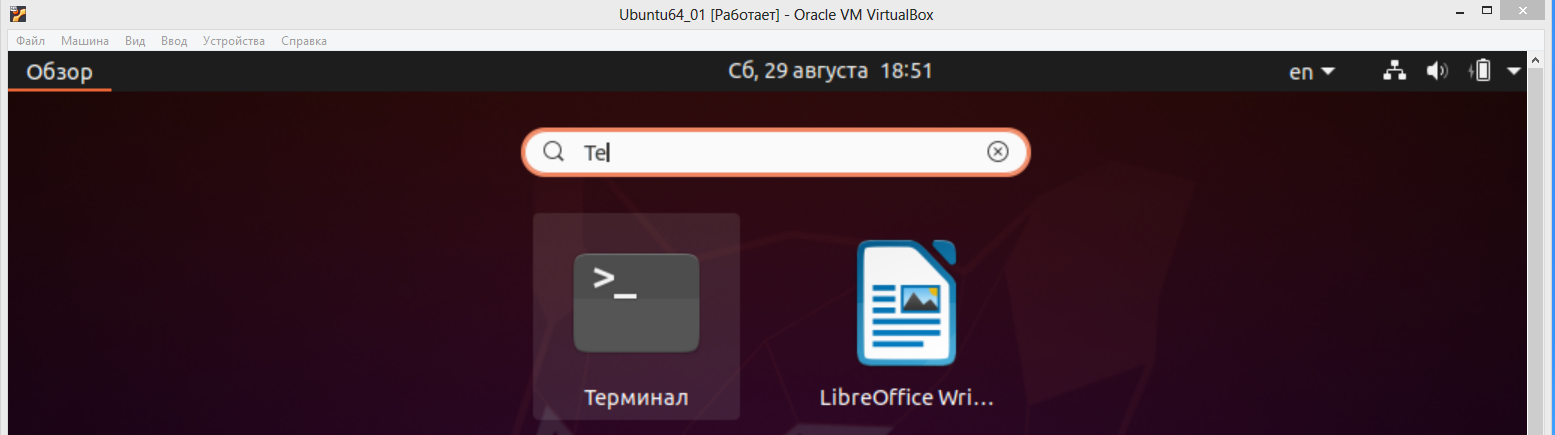


Рис. 26. Нахождение установленного приложения Терминал

Обычно при завершении работы виртуальной машины можно не закрывать запущенные программы, если завершать работу с сохранением состояния. Кроме этого, после выполнения важных этапов изменения программного обеспечения системы можно зафиксировать текущее состояние машины, создавая специальные файлы, называемые снимками состояния (рис. 27). В этом случае, если при работе с виртуальной машиной что-то произошло, мы можем сделать откат к одному из снимков (обычно к последнему).

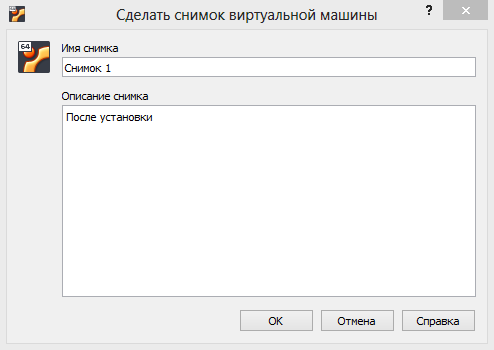


Рис. 27. Выполнение команды Машина -> Сделать снимок

**3. ОБОРУДОВАНИЕ**

Персональный компьютер типа IBM PC Pentium с установленной операционной системой MS Windows 7/8/10, установщик и пакет расширений для программы VirtualBox, дистрибутивы операционной системы Linux, файл МУ\_ЛР\_1.doc, содержащий методические указания к лабораторной работе.

**4. ЗАДАНИЕ НА РАБОТУ**

4.1. Ознакомиться с операционной системой Linux и программой виртуализации VirtualBox.

4.2. Ознакомиться с созданием виртуальной машины на базе виртуального диска с ранее созданной ВМ, содержащей гостевую систему Debian 8/32.

4.3. Создать новую ВМ на базе согласованного с преподавателем дистрибутива системы Linux. Предлагаемые варианты дистрибутивов: Ubuntu 32/64, Fedora 32/64, Debian 9/64, CentOS 64, openSUSE 64, ALT Linux 64, Runtu 32/64, ROSA Fresh R10, Linux Mint, LMDE.

**6. ОФОРМЛЕНИЕ ОТЧЕТА**

Отчет должен содержать краткое описание дистрибутивов системы Linux и скриншоты процесса установки гостевой системы Linux.

**БИБЛИОГРАФИЧЕСКИЙ СПИСОК**

1. Колисниченко Д.Н. Linux. От новичка к профессионалу. – 5-е изд., перераб. и доп. – СПб.: БХВ-Петербург, 2016. – 608 с.

2. Блог начинающего линуксоида.\_ Русский Linux\_ обзор Российских дистрибутивов общего назначения – URL: <http://linuxoidblog.blogspot.ru/2015/11/linux_20.html> . Дата последнего обращения: 29.08.20.

3. Как включить Intel VT-x в BIOS вашего компьютера или в прошивке UEFI – URL: <https://www.howtogeek.com/213795/how-to-enable-intel-vt-x-in-your-computers-bios-or-uefi-firmware/> . Дата последнего обращения: 29.08.20.

4. Импорт и экспорт виртуальной машины в VirtualBox - URL: <http://tavalik.ru/import-i-ekport-virtualnoj-mashiny-virtualbox/> . Дата последнего обращения: 29.08.20

5. Установка Ubuntu на виртуальную машину Oracle VirtualBox - URL: <http://profitraders.com/Ubuntu/VirtualBoxUbuntuInstall.html> Дата последнего обращения: 29.08.20.

6. Настройка Ubuntu 20.04 после установки - URL: <https://losst.ru/nastrojka-ubuntu-20-04-posle-ustanovki> Дата последнего обращения: 29.08.20.

Лабораторная работа № 2

Файлы и каталоги в системе Linix

**1. ЦЕЛЬ И ЗАДАЧИ РАБОТЫ**

Ознакомление с файловой системой и командами работы с файлами в операционной системе Linux, а также с основными приемами работы нового пользователя в системе.

**2. ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ**

Операционная система Linux традиционно широко использует командный консольный интерфейс, который можно использовать либо в командной строке, либо в окне эмулятора терминала (терминала в дальнейшем). Предположим вначале, что пользователь работает в системе Debian и имеет логин virtualuser.

После запуска терминала целесообразно сменить фон в окне терминала с черного на белый, поскольку в отчетах по лабораторным работам не должно быть скриншотов окон с черным фоном. Смену фона можно сделать либо с помощью следующей команды setterm:



либо с помощью окна «Настройки терминала», вызываемое через команды меню терминала Правка / Параметры (рис. 1).



Рис. 1. Изменение цвета в настройках терминала

Кроме этого, в закладке **Внешний вид** необходимо установить темно-синий цвет для курсора и для цвета №11 в палитре, задать размер шрифта 9 пт, а также выбрать **Прозрачный фон** и степень прозрачности 1,0. Возможно, что установленные настройки терминала начнут работать полностью только после перезагрузки системы.

Для открытия окна терминала можно воспользоваться горячей клавишей Ctrl+Alt+T. В системе Ubuntu окно терминала для быстрого доступа можно добавить в панель "Избранное", т.е. разместить значок окна терминала в панели слева главного окна (рис. 2).

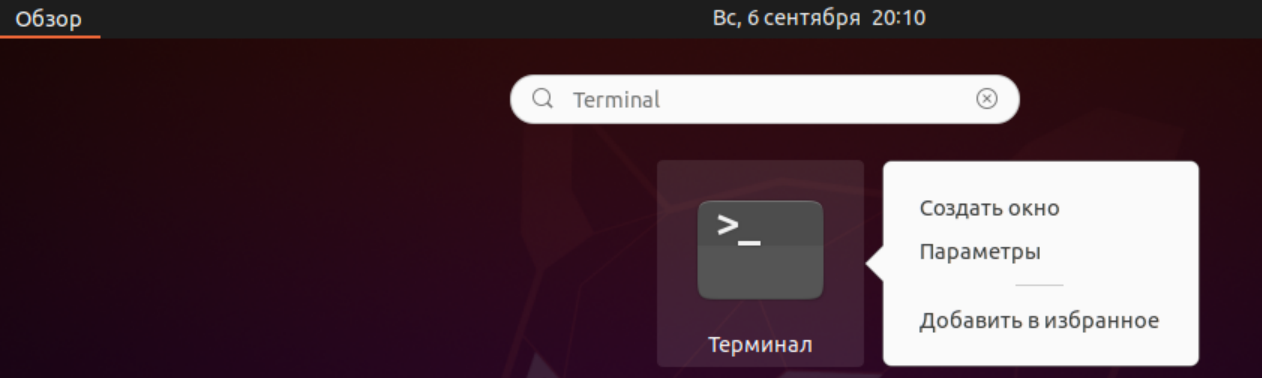


Рис. 2. Добавление значка окна терминала в панель "Избранное"

С помощью справочной системы **man** можно получить справку по любой консольной команде, реализованной в виде утилиты, например, для команды ls можно выполнить в терминальном окне команду **man ls**  (рис. 3).

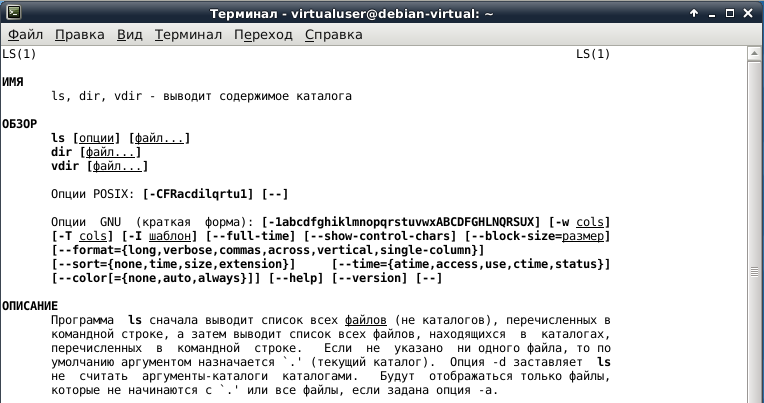


Рис. 3. Справка по команде ls

Отметим, что система **man** не выдает справки по внутренним командам оболочки shell, т.е. по командам, которые реализованы в самой оболочке. Для таких команд предназначена справочная система help (рис. 4).

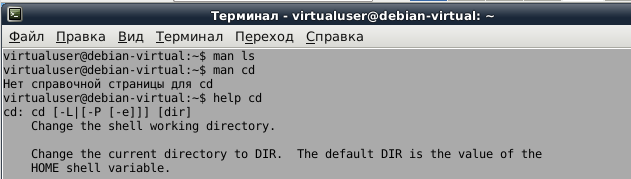


Рис. 4. Справка по команде cd

Альтернативной по отношению к системе **man** является справочная система **info**. Например, для получения информации о команде sudo выполним команду **info sudo** (рис. 5).

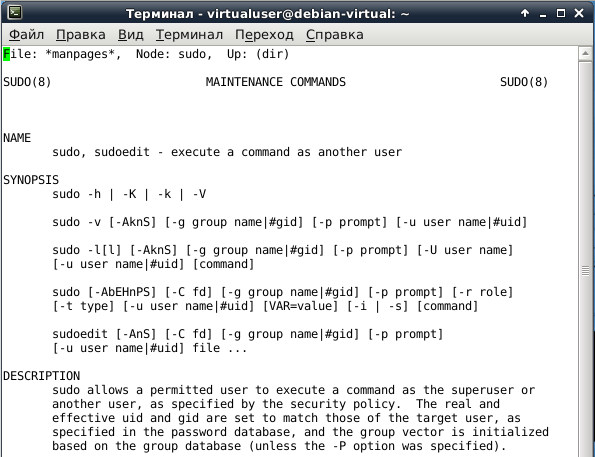


Рис. 5. Справка по команде sudo

Для перемещения по страницам справки наряду с прокруткой мышки обычно используются те же клавиатурные команды навигации, которые имеются в программе **less**, которую называют вьюером или листателем. Например, стрелки вверх/вниз перемещают курсор по строкам вверх/вниз, клавиши PgUp/PgDn листают страницы вверх или вниз, клавиши Home/End перемещают курсор в начало или в конец документа, клавиша **q** завершает работу этой справочной системы.

При работе в окне терминала мы можем просматривать и/или изменять значения переменных окружения HOME, PATH, PS1, PS2 и многих других, используя команду echo (рис. 6). Например, переменная PS1 является основным промптером, с помощью которого командная оболочка bash сообщает пользователю о необходимости ввода следующей команды. Если команда занимает несколько строк (например, это команда цикла или это целый скрипт на командном языке), переменная PS2 содержит промптер для второй и последующих строк команды.

Для кодирования основного промтера существует целый язык, который определяет, что будет представлено в промптере. Для получения справки о кодировании промтера необходимо открыть справку о командной оболочке bash, которая содержит несколько сот страниц (рис. 7).

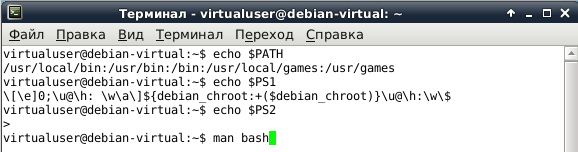


Рис. 6. Вывод содержимого переменных окружения

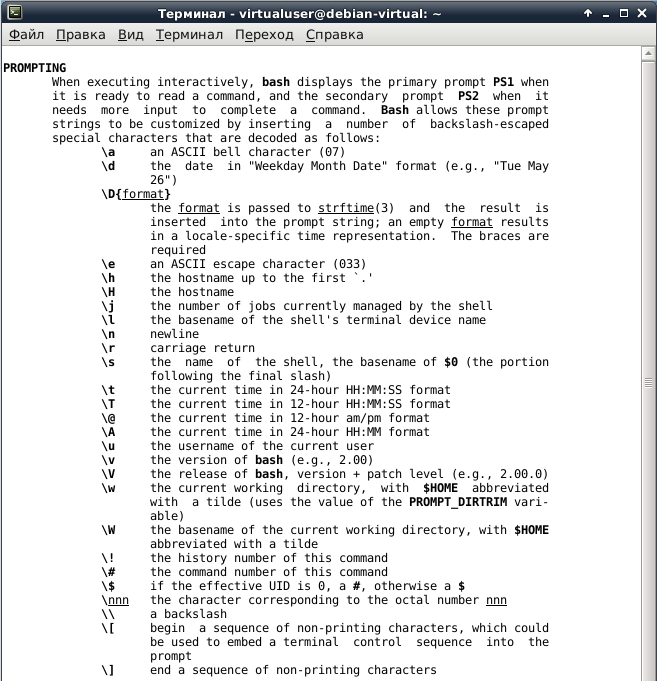


Рис. 7. Справка по команде bash (раздел о кодировании промптера)

Для выполнения лабораторных работ нам потребуется скопировать в виртуальную машину с ОС Debian информационно-методическое обеспечение (ИМО) лабораторных работ. Для этого соответствующий внешний носитель, содержащий ИМО в каталоге Labs\_2020, должен быть смонтирован и подключен к корневой файловой системе ОС Linux. Наиболее просто это сделать, используя оптические диски CD/DVD-ROM, поскольку они обычно используются для установки гостевой ОС в ВМ и поэтому монтируются автоматически. Если диск с ИМО уже вставлен в привод "I", то подключаем его через следующие команды меню VirtualBox: Устройства/Оптические диски/Привод хоста "I" (рис. 8).

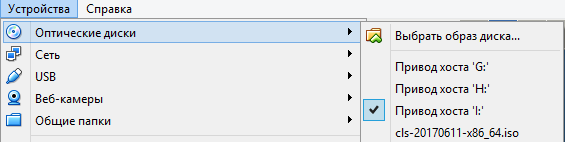


Рис. 8. Подключение оптического диска

На рабочем столе появляется значок оптического диска, устанавливаем на него курсор мыши и в контекстном меню выбираем **Подключить том**. Поскольку привод CD/DVD-ROM обычно монтируется по адресу /media/cdrom/cdrom0, то именно там мы и обнаружим каталоги и файлы оптического диска (рис. 9).

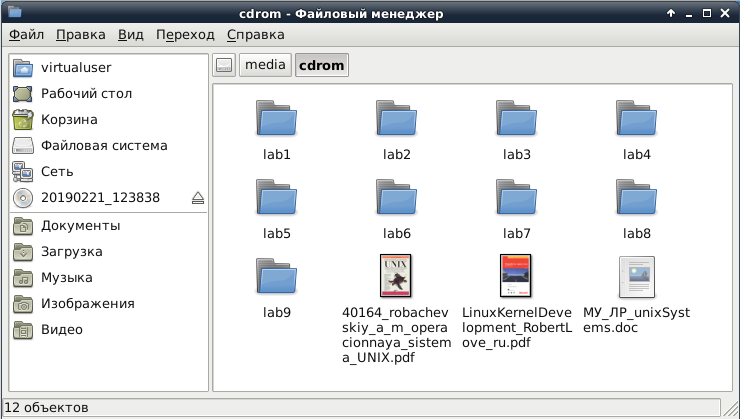


Рис. 9. Окно графического файлового менеджера

Отметим, что при монтировании привода CD/DVD-ROM каталог Labs\_2020 совместится с каталогом **cdrom0**, т.е. станет элементом связи двух файловых систем, который не будет отображаться в окне графического файлового менеджера.

Для копирования ИМО с оптического диска в каталог /home/virtualuser/Labs\_2020 виртуальной машины (каталог Labs\_2020 в каталоге virtualuser был создан заранее) воспользуемся файловым менеджером **mc**. Предварительно он должен быть установлен (инсталирован). Для его запуска выполним команду **mc** (рис. 10).

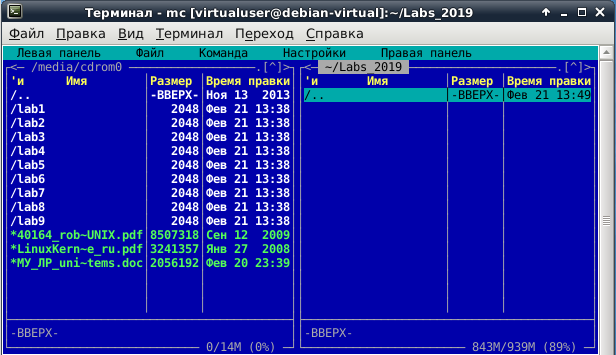


Рис. 10. Окно программы mc

Далее выделяем содержимое левого окна и командой F5 копируем все в правое окно. Затем командой exit завершаем работу менеджера **mc**.

Следующая операция является очень важной при работе с системой Linux. Устанавливаем курсор мыши на значок оптического диска (на рабочем столе) и в контекстном меню выбираем команду **Извлечь том** (рис. 11).

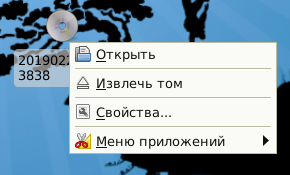


Рис. 11. Извлечение оптического диска из привода

Оптический диск автоматически отключается от системы и открывается для его извлечения из привода. После этого в меню **Устройства** программы VirtualBox необходимо убрать галочку выбора привода оптического диска.

Далее откроем в графическом файловом менеджере каталог Labs\_2020. Выполняя двойной клик на документах с расширениями .pdf и .doc, мы запустим установленные для них редакторы. Однако эти документы мы могли открыть и на хосте. Но в этом случае нам пришлось бы постоянно переключаться из окна ВМ на рабочий стол хоста и обратно.

Закроем графический файловый менеджер и откроем окно терминала. Перейдем в каталог Labs\_2020/lab2. По команде ls -l выведем на экран метаданные файлов в каталоге lab2 (рис. 12).

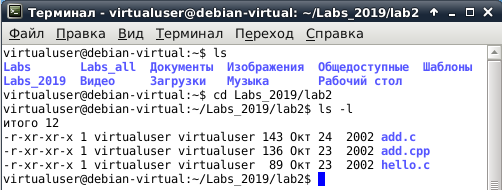


Рис. 12. Метаданные файлов в каталоге lab2

Здесь каждая строка метаданных начинается с тире, что соответствует типу файла – обычный файл (типами файлов в Linux являются также каталоги, символьные ссылки, каналы, сокеты). Далее в строке метаданных идет описание в виде rwx для трех категорий пользователей: владельца-пользователя, владельца-группы и остальных пользователей. Однако для каждой категории вместо rwx задано r-x, т.е. есть есть доступ по чтению (бит r) и доступ по выполнению (бит x), но отсутствует доступ по записи/редактированию (бит w). После символов прав доступа в метаданных указывается количество жестких связей у файла (значение 1), кто является владельцем-пользователем и владельцем-группой (virtualuser у нас), размер файла, дата создания файла. Отметим, что имя файла не входит в метаданные, так как у файла может быть несколько имен (говорят, что файл имеет несколько жестких связей).

Поскольку мы являемся владельцами файлов в каталоге lab2, мы имеем возможность изменить права доступа этих файлов, например, разрешить редактировать эти файлы пользователям-владельцам файлов командой chmod (рис. 13).

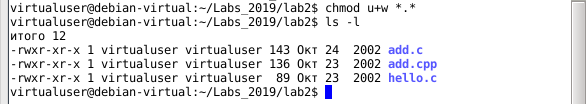


Рис. 13. Измененные права доступа файлов

Затем мы можем перейти в каталог Labs\_2020 и изменить доступ всех находящихся там каталогов и файлов документов.

В некоторых гостевых системах Linux не удается записать в них ИМО через оптический диск. Однако существует целый набор возможностей обмена данными между хостом и гостевой системой, которые можно будет использовать после выполнения еще одного этапа установки гостевой системы - установка гостевых дополнений VirtualBox в Ubuntu 20.04.

Гостевые дополнения обеспечивают поддержку следующих функций:

* интеграция указателя мыши хоста и гостевой системы;
* общие папки для удобного обмена файлами между хостом и гостем;
* обеспечивает лучшее и ускоренное воспроизведение видео;
* общие каналы связи между хостом и гостем, которые позволяют контролировать и управлять гостевым исполнением;
* обеспечивает синхронизацию времени между хостом и гостем;
* общий буфер обмена между хостом и гостевой виртуальной машиной;
* автоматический вход.

Чтобы установить гостевые дополнения VirtualBox в Ubuntu 20.04, вам потребуется файл ISO гостевых дополнений. ISO-файл гостевого дополнения можно вставить в виртуальную машину двумя способами:

* вставить файл ISO с гостевыми дополнениями вручную;
* автоматическая установка гостевых дополнений ISO из репозиториев Ubuntu с помощью диспетчера пакетов APT.

В первом случае перейдите в строку меню на своей виртуальной машине и нажмите Устройства -> Вставить образ компакт-диска с гостевыми дополнениями. Обратите внимание, что прежде чем вы сможете вставить файл ISO, у вас должен быть хотя бы один пустой контроллер IDE. Кроме этого, прежде чем вы сможете выполнить установку гостевого дополнения, вам сначала необходимо установить необходимые инструменты сборки, заголовки ядра Linux и модули поддержки, если они еще не установлены.

Выполните приведенную ниже команду, чтобы убедиться, что эти требования уже установлены.

sudo dpkg -l | grep -E "dkms | linux-headers - $ (uname -r) | build-essential"

Результат выполнения может быть следующим:

ii build-essential 12.8ubuntu1 amd64 Информационный список пакетов, необходимых для сборки

ii linux-headers-5.3.0-18-generic 5.3.0-18.19 amd64 Заголовки ядра Linux для версии 5.3.0 на 64-битном x86 SMP

Здесь  ii означает, что пакет установлен. Как видно из вывода, dkms не установлен. Вы можете просто использовать диспетчер пакетов APT для их установки. Если ничего не установлено, Вы выполняете следующие команды установки:

sudo apt update -y

sudo apt upgrade

sudo apt install dkms linux-headers - $(uname -r) build-essential

Как только это будет сделано, перейдите к вставке ISO-образа гостевых дополнений: Устройства -> Подключить образ диска дополнений гостевой ОС.

Откроется диалоговое окно, в котором вам будет предложено запустить или отменить установщик гостевых дополнений.

Нажмите Выполнить**,**  чтобы запустить установку гостевых дополнений в Ubuntu 20.04. Вам будет предложено ввести пароль для аутентификации установки. После успешной аутентификации будет запущена установка гостевого дополнения.

При установке гостевых дополнений часто выдается сообщение, что вам не хватает некоторых модулей ядра (рис. 14).

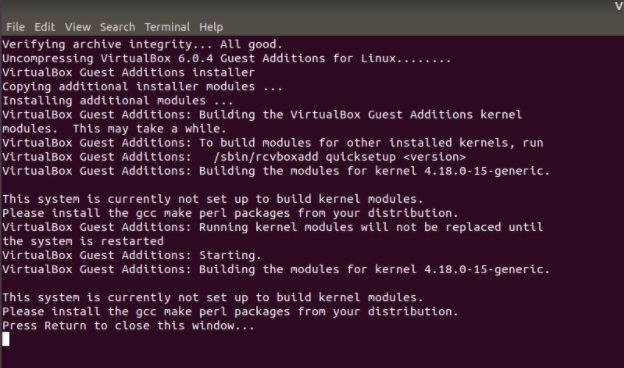


Рис. 14. Сообщение об ошибке

Вам придется установить еще несколько пакетов. Для этого вам нужно запустить следующие команды в виртуальной системе Ubuntu:

sudo apt install build-essential dkms linux-headers-generic

После этого запускаем повторную установку гостевых дополнений:

sudo rcvboxadd setupsumc

После установки гостевых дополнений размер экрана автоматически изменится. Нажмите Enter, чтобы закрыть экран установки. Вы также можете перезапустить виртуальную машину командой sudo reboot, чтобы перезагрузить модули ядра.

Для проверки наличия установленных гостевых дополнений выполним следующую командную строку:

lsmod | grep vbox

Результат выполнения должен быть примерно следующим:

vboxsf 81920

vboxguest 344064 6 vboxsf

Установка гостевых дополнений не всегда сразу проходит успешно. Поэтому рассмотрим второй способ установки ISO-файл гостевых дополнений - установка из репозиториев Ubuntu. Файл ISO гостевых дополнений предоставляется  virtualbox-guest-additions-iso пакетом.

Вначале проверим политику пакета:

sudo apt-cache policy virtualbox-guest-additions-iso

Если пакет доступен, просто запустите команду ниже, чтобы установить его:

sudo apt install virtualbox-guest-additions-iso

После установки VirtualBox Guest Additions ISO теперь доступен как /usr/share/virtualbox/VBoxGuestAdditions.iso.

Чтобы установить его, нужно его смонтировать:

sudo mount -o loop /usr/share/virtualbox/VBoxGuestAdditions.iso /media/

Далее запускаем установку:

sudo /media/VBoxLinuxAdditions.run

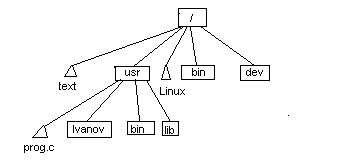
После завершения установки перезагрузите систему, чтобы перезагрузить модули ядра.

Теперь вы должны быть в полноэкранном режиме и можете включить общий буфер обмена, общие папки, перетаскивание…

Если после перезагрузки системы размер экрана не изменяется автоматически, не забудьте переключить (так сказать, выключить / включить)   View -> Auto-resize Guest Display в строке меню виртуальной машины, чтобы установить правильное разрешение виртуальной машины.

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.



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.

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

Операционная система Linux/Unix наиболее часто работает на серверах, где графические оболочки обычно не устанавливаются. В этом случае командный интерфейс вместе с горячими клавишами являются основным инструментом специалистов, обслуживающих серверы.

Консольный строковый редактор ed является самым простым текстовым редактором операционной системы Unix/Linux. Он может использоваться без изменения и настройки на терминалах любого типа и не требует присутствия в системе графической среды. Более мощными редакторами данного типа, являющимися фактически расширениями редактора ed, являются редакторы joe, vi, emacs и другие. Данные текстовые редакторы, работающие из командной строки, жизненно необходимы специалистам по операционным системам, поскольку при авариях системы они не смогут обойтись без них. Текстовый редактор ed может применяться, если не работает графическая система X Window или нельзя воспользоваться эмуляцией терминала.

Простейшими командами редактора ed являются:

a - вставить текст после текущей строки;

i - вставить текст перед текущей строкой;

d - удалить текущую строку;

p - вывести на консоль (терминал) текущую строку;

c - заменить текущую строку;

s - заменить фрагмент текста в строке;

g - найти строки по указанному фрагменту текста;

w - записать текст из буфера в выходной файл;

q - завершить работу с редактором.

В следующем фрагменте диалога с помощью редактора ed создается файл add1.cpp, содержащий исходный текст программы сложения двух чисел на языке С++ (символы, выводимые редактором или интерпретатором, подчеркнуты):

$ ed add1.cpp

\* a

#include <iostream.h>

main()

{ int x, y;

cout<<"Input two numbers: ";

cin>>x>>y;

cout<<"Summa = "<< x+y <<endl;

}

.

\* w

\* q

$ cat add.cpp

Здесь по команде a редактор входит в режим ввода текста, который должен заканчиваться точкой в пустой строке. При редактировании файла адресация строк производится как по номерам, так и по контексту, например:

5 - установить пятую строку в качестве текущей и вывести ее на экран;

8i,8a - вставить текст перед или после восьмой строки;

4dp - удалить четвертую строку и вывести на экран текущую (пятую) строку;

$ - установить последнюю строку в качестве текущей и вывести ее на экран;

/x=/ - установить в качестве текущей и вывести на экран строку, содержащую фрагмент текста "x=";

2,4d - удалить строки 2,3,4;

1,$g/begin/s/b/B/p - найти в файле строки со словом begin, заменить в них символ b на B и вывести измененные строки на экран;

s/^/ /p - вставить три пробела в начало текущей строки;

s/$/;/p - добавить литеру ; в конец строки.

**3. ОБОРУДОВАНИЕ**

Персональный компьютер типа IBM PC Pentium с установленной операционной системой MS Windows 7/8/10, установщик и пакет расширений для программы VirtualBox, дистрибутивы операционной системы Linux, файл МУ\_ЛР\_1.doc, содержащий методические указания к лабораторной работе.

**4. ПОРЯДОК ВЫПОЛНЕНИЯ РАБОТЫ**

5.1. Загрузить операционную систему Linux и после появления приглашения login войти в систему.

5.2. Ознакомиться с рабочим столом графической оболочки. Запустить эмулятор консольного окна (терминал) и с помощью команды pwd вывести в консольное окно путь к начальному (домашнему) каталогу пользователя.

5.3. Командой ls вывести на экран текущий каталог пользователя. Для вывода более подробной информации необходимо в команде ls использовать (поочередно, а затем вместе) флажки l,a,i. Выяснить: назначение каждого флажка, обозначения для текущего и родительского каталогов, выводимые атрибуты файлов.

5.4. Вывести на экран содержимое текстового файла большого размера, используя поочередно команды more, less и cat. Выяснить, в чем различие данных команд. Ознакомиться с языком командных файлов (скриптов), хранящихся в домашнем каталоге пользователя в виде скрытых файлов .profile, .bash\_profile и других.

5.5. Использовать команду cat для создания нового текстового файла (например, файла addxx.cpp, где xx - номер варианта задания) и команды ls и cat для проверки наличия и содержимого созданного файла.

5.6. Перейти к рабочему столу графической оболочки и с помощью текстового редактора графической среды отредактировать и сохранить файл addxx.cpp. Вернуться в консольное окно и командой cat просмотреть измененный файл.

5.7. Путем смены текущего каталога (посредством команды cd) произвести движение вверх по дереву файлов, а затем вернуться в начальный каталог пользователя. После каждой смены текущего каталога вывести в консольное окно его имя (путь) и содержимое. Выяснить, в каких каталогах находятся жесткие и символические связи (ссылки) и какими буквами обозначаются типы этих файлов в выводе команды ls. Для этого необходимо в команде ls использовать (поочередно, а затем вместе) флажки l,a,i.

5.8. После возврата в домашний каталог создать в нем командой mkdir подкаталог lab2 (если он еще не создан). Сделать каталог lab2 текущим, вывести на экран полное имя текущего каталога и содержимое родительского каталога.

5.9. С помощью консольного строкового редактора ed создать файл add2.cpp, содержащий исходный текст программы сложения двух чисел на языке С++. С помощью команды cat вывести текст созданного файла на экран.

5.10. С помощью консольного строкового редактора ed отредактировать файл add2.cpp: вставить строки комментариев в начало, в конец файла и в конце каждой строки программы. Проверить работу различных команд редактора. Измененный текст файла просмотреть сначала в редакторе, а затем с помощью команды cat интерпретатора команд.

5.11. С помощью команды set (или printenv) вывести в окно консоли текущие установки среды, хранящиеся в системных переменных HOME, PATH, PS1, PS2 и других. Проверить значения системных переменных HOME и SHELL с помощью команды echo. Изменить приглашение, выдаваемое интерпретатором команд.

5.12. По команде history (или алиасу h) вывести в окно консоли список ранее выполнявшихся команд интерпретатора. Проверить возможность повторного запуска команды по ее номеру в списке.

5.13. Ознакомиться со средствами оперативной помощи - командами help, man, whatis, whereis, which, info.

5.14. Ознакомиться с содержимом регистрационного файла /etc/passwd и определить атрибуты, установленные для пользователя.

5.15. С помощью команды ln создать жесткие связи с файлом add.cpp в различных каталогах и ознакомиться с их атрибутами, используя флажок i в команде ls для связей и флажок l команды ls для каталогов, содержащих жесткие связи.

5.16. С помощью команды ln -s создать символические связи с файлом add.cpp и ознакомиться с их атрибутами. Сравнить атрибуты файлов-жестких связей с атрибутами файлов-символических связей.

5.17. С помощью команды mc (или midc) запустить менеджер файлов GNU Midnight Commander - аналог программы Norton Commander. Ознакомиться с системными каталогами bin, lib, etc, usr и другими. По завершении работы закрыть командой exit менеджер mc и вернуться в консольное окно.

5.18. Ознакомиться с правами доступа, установленными для различных файлов и каталогов. Командой chmod изменить права доступа для каталога varxx и файла addxx.cpp в этом каталоге. Выяснить, как изменение прав доступа для каталога влияет на возможность просмотра каталога командой ls, возможность просмотра атрибутов файла командой ls -l, возможность перехода в этот каталог или на возможность чтения, редактирования или удаления файла в каталоге. (*Примечание*. Прежде, чем удалять файл из каталога командой rm, сделайте одну или несколько копий этого файла командой cp).

5.19. Создать в каталоге lab2 "темный" подкаталог, в котором файлы доступны только в случае, если пользователь заранее знает их имена.

5.20. Ознакомиться с возможностями использования дополнительных атрибутов для управления доступом к файлам и каталогам.

**5. ОФОРМЛЕНИЕ ОТЧЕТА**

Отчет должен содержать результаты выполнения различных команд в процессе лабораторной работы.

**6. КОНТРОЛЬНЫЕ ВОПРОСЫ**

7.1. Перечислите типы файлов в системе UNIX.

7.2. Как реализованы каталоги в системе UNIX?

7.3. В каком файле хранится учетная информация о пользователях?

7.4. Укажите назначение различных флажков команды ls.

7.5. Чем отличаются жесткие связи от символических?

7.6. Как изменяется счетчик ссылок на файл при его копировании?

7.7. Что хранится в различных системных каталогах системы Linux?

7.8. Перечислите команды строкового редактора ed.

7.9. Какие действия выполняют следующие команды строкового редактора ed:

|  |  |
| --- | --- |
| s/THEN/then/p  .,$p  /beg/;/end/d  1,20g/^[0-9].\*/t$ | s/x+y/(&)/p  .,/end/d  /beg/,/end/s/;.\*$/;/p  /end/r file1 |

7.10. Как создать "темный" каталог?

**БИБЛИОГРАФИЧЕСКИЙ СПИСОК**

1. Робачевский, А.М. Операционная система UNIX : учеб.пособие / А.М.Робачевский, С.А.Немнюгин, О.Л.Стесик .— 2-е изд.,перераб.и доп. — СПб. : БХВ-Петербург, 2007 .— 635с.

2. Колисниченко Д.Н. Linux. От новичка к профессионалу. – 5-е изд., перераб. и доп. – СПб.: БХВ-Петербург, 2016. – 608 с.

3. Кетов Д.В. Внутреннее устройство Linux. - СПб.: БХВ-Петербург, 2017. – 320 с.

4. К.Кристиан. Введение в операционную систему UNIX. - М.: Финансы и статистика, 1985.

5. Р.Готье. Руководство по операционной системе UNIX. - М.: Финансы и статистика, 1986.

6. М.Банахан, Э.Раттер. Введение в операционную систему UNIX. - М.: Радио и связь, 1986.

7. П.Браун. Введение в операционную систему UNIX. - М.: Мир, 1987.

8. Install VirtualBox Guest Additions on Ubuntu 20.04 - kifarunix.com.html - URL: <https://kifarunix.com/install-virtualbox-guest-additions-on-ubuntu-20-04/> . Дата последнего обращения: 29.08.20.

9. Как установить и использовать гостевые дополнения VirtualBox в Ubuntu - URL: <https://omgubuntu.ru/kak-ustanovit-i-ispolzovat-ghostievyie-dopolnieniia-virtualbox-v-ubuntu/> . Дата последнего обращения: 29.08.20.

Лабораторная работа № 3

Интерпретаторы команд и командные процедуры в системе Linix

**1. ЦЕЛЬ И ЗАДАЧИ РАБОТЫ**

Ознакомление с командным языком системы Linux и получение практических навыков программирования и отладки командных процедур.

**2. ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ**

Все современные системы Unix (в том числе и Linux) поставляются с различными оболочками ядра, являющимися командными интерпретаторами. Каждый интерпретатор имеет свой командный язык и использует соответствующие ему командные процедуры (скрипты) для инициализации среды, в которой будут выполняться команды пользователя:

|  |  |
| --- | --- |
| **Командный интерпретатор** | **Скрипт инициализации** |
| Bourne shell (sh) | .profile |
| C shell (csh) | .login и .cshrc |
| Korn shell (ksh) | .profile и .kshrc |
| Bourne-again shell (bash) | .profile и .bashrc |

В операционной системе Linux по-умолчанию обычно используется оболочка *bash*. Когда оболочка запускается, она инициализируется, затем выводит на экран промптер – символ или строку приглашения к вводу. Пользователь может изменить промптер, например, сделать так, чтобы он содержал имя текущего каталога и автоматически изменялся при смене каталога. После того, как пользователь введет командную строку, оболочка (или ее потомок, которого она обычно создает для выполнения команды) извлекает первое слово командной строки и ищет подпрограмму в оболочке (если команда относится к внутренним командам оболочки) либо готовую программу (обычно утилиту, находящуюся в виде файла в текущем каталоге или в одном из системных каталогов).

У команд могут быть аргументы, которые передаются запускаемой программе в виде текстовых строк. Например, командная строка

cp src dest

запускает программу *cp*, которая делает копию файла *src* и называет эту копию именем *dest*.

Аргументами могут быть флаги (обычно со знаком тире), которые управляют работой команды. Например, команда

head -20 file

напечатает первые двадцать строк файла *file* (вместо принятых по умолчанию 10 строк), а команда

head 20 file

даст указание программе *head* вывести первые 10 строк файла 20, а затем первые 10 строк файла *file*. Большинство команд Linux-систем могут принимать несколько флагов и аргументов.

Чтобы упростить указание нескольких файлов в команде, оболочка принимает так называемые групповые символы. Например, символ звездочка означает все возможные текстовые строки, так что строка

ls \*.c

даст указание программе *ls* вывести список всех файлов текущего каталога, имя которых оканчивается на *.с*. Если в каталоге находятся файлы *x.c*, *y.c* и *z.c*, то данная команда будет эквивалентна команде

ls x.c y.c z.c

Другим групповым символом является вопросительный знак, который заменяет один любой символ. Кроме того, в квадратных скобках можно указать множество символов, из которых программа должна будет выбрать один. Например, команда

ls [ame]\*

выводит список файлов, имя которых начинается с символов «а», «m» или «e».

Оболочка не должна открывать терминал (клавиатуру и монитор), чтобы прочитать с него или сделать на него вывод. Вместо этого запускаемые программы автоматически получают доступ для чтения к файлу (точнее, к устройству, поскольку в Linux каждое устройство является одним из видов файла), называемому стандартным устройством ввода (stdin), а для записи – к файлу, называемому стандартным устройством вывода (stdout), и к файлу, называемому стандартным устройством для вывода сообщений об ошибках (stderr). По умолчанию всем этим трем устройствам соответствует терминал, т.е. чтение производится с клавиатуры, а запись – на монитор. Многие Linux-программы читают данные со стандартного устройства ввода и пишут на стандартное устройство вывода. Например, команда

sort

запускает программу-утилиту, находящуюся в файле *sort*, которая читает строки с терминала (пока пользователь не нажмет комбинацию клавиш CTRL-D, чтобы обозначить конец файла), а затем сортирует их в алфавитном порядке и выводит результат на экран.

Стандартные ввод и вывод можно перенаправить, что является очень полезным свойством. Для этого используются символы «<» и «>» соответственно. Например, команда

sort <in >out

заставляет программу *sort* взять в качестве входного файла *in* и направить вывод в файл *out*. Поскольку стандартный вывод сообщений об ошибках не был перенаправлен, то все сообщения об ошибках попадут на экран. Программа, которая считывает данные со стандартного устройства ввода, выполняет определенную обработку этих данных и записывает результат в поток стандартного вывода, называется фильтром.

Рассмотрим следующую командную строку, состоящую из трех отдельных команд:

sort <in >temp; head -30 <temp; rm temp

Сначала запускается программа *sort*, которая принимает данные из файла *in* и записывает результат в файл *temp*. Когда она завершает свою работу, оболочка запускает программу *head*, дав ей указание вывести первые 30 строк из файла *temp* на стандартное устройство вывода, которым по умолчанию является терминал. По завершению вывода временный файл *temp* удаляется.

Часто используются командные строки, в которых первая программа в командной строке формирует вывод, используемый второй программой в качестве входа. В приведенном выше примере для этого использовался временный файл temp. Однако система Linux (как и Unix) предоставляет для этого более простой способ. В командной строке

sort <in | head -30

используется вертикальная черта, называемая символом канала. Она означает, что вывод программы *sort* должен использоваться в качестве входа для программы *head*, что позволяет обойтись без создания, использования и удаления временного файла.

Набор команд, соединенных символом канала, называется конвейером. Он может содержать произвольное количество команд. Пример четырехкомпонентного конвейера показан в следующей строке:

grep ter \*.t | sort | head -20 | tail -5 >foo

Здесь в стандартное устройство вывода записываются все строки, содержащие строку «ter» во всех файлах, оканчивающихся на *.t*, после чего они сортируются. Первые 20 строк выбираются программой *head*, которая передает их программе *tail*, записывающей последние пять строк (из переданных двадцати) в файл *foo*.

Linux является универсальной многозадачной системой. Один пользователь может одновременно запустить несколько программ, каждую в виде отдельного процесса. Синтаксис оболочки для запуска фонового процесса состоит в использовании амперсанда в конце строки. Таким образом, строка

wc -l <file1 >file2 &

запустит программу wc для подсчета количества символов/слов/строк, которая сосчитает число строк (флаг -l) во входном файле file1 и запишет результат в файл file2, но будет делать это в фоновом режиме. Как только команда будет введена пользователем, оболочка выведет символ приглашения к вводу и будет готова к обработке следующей команды.

Конвейеры также могут выполняться в фоновом режиме, например:

sort <file1 | head &

Можно одновременно запустить в фоновом режиме несколько конвейеров.

Список команд оболочки может быть помещен в файл, а затем можно будет запустить оболочку с этим файлом в качестве стандартного входа. В этом случае оболочка выполнит перечисленные в этом файле команды одну за другой – точно так же, как если бы эти команды вводились с клавиатуры. Файлы, содержащие команды оболочки, называются сценариями оболочки.

Сценарии оболочки могут присваивать значения переменным оболочки и затем считывать их. Они также могут иметь параметры и использовать конструкции if, for, while и case. Таким образом, сценарии оболочки представляют собой настоящие программы, написанные на языке оболочки.

Оболочки системы Linux позволяют компилировать и выполнять программы, написанные на различных языках программирования. В качестве примера рассмотрим программы hello.c, add.c и add.cpp, написанные на языках C и C++ соответственно:

// hello.c

#include <stdio.h>

using namespace std;

int main ()

{

printf ("Hello, world!\n");

return 0; }

// add.c

#include <stdio.h>

using namespace std;

int main ()

{ int x, y;

printf("Input 2 numburs: ");

scanf ("%d%d", &x, &y);

printf("Summa = %d\n", x+y);

return 0;

}

// add.cpp

#include <iostream>

using namespace std;

int main ()

{ int x, y;

cout<<"Input 2 numburs: ";

cin>>x>>y;

cout<<"Summa = "<<x+y<<endl;

return 0;

}

Для того, чтобы установить компилятор g++ для языка Си++ в системе Ubuntu (если он еще не установлен), открываем терминал, например, с помощью комбинации C**trl+Alt+T**, и вводим команды:

sudo apt-get install g++

sudo apt-get update

Ubuntu версии 16.04 и выше предоставляет своим пользователям apt - новый менеджер пакетов для Ubuntu, готовый взять на себя работу apt-get. Он разрабатывался, чтобы быть более дружелюбным, эффективным и безопасным. Поэтому для установки c++ можно использовать следующие команды:

sudo apt install g++

sudo apt update

Аналогично устанавливаются компиляторы cc и gcc для языка Си. В некоторых дистрибутивах присутствуют компиляторы с именами cpp и c++. Если компиляторы уже установлены, то мы можем проверить версию компилятора (или просто наличие компилятора), например, следующей командой для компилятора g++:

g++ --version

или командой

g++ -v

В базовую поставку компилятора языка с/c++ входят такие программы:

* libc6-dev - заголовочные файлы стандартной библиотеки Си;
* libstdc++6-dev - заголовочные файлы стандартной библиотеки С++;
* gcc - компилятор языка программирования Си;
* g++ - компилятор языка программирования C++;
* make - утилита для организации сборки нескольких файлов;
* dpkg-dev - инструменты сборки пакетов deb (пакетов Debian).

Пакеты этих программ являются зависимостями (составными частями) пакета build-essential, поэтому для установки всего необходимого достаточно установить этот пакет:

sudo apt-get update

sudo apt-get install build-essential

apt-get install g++-multilib

Первая команда обновляет базу пакетов. Её периодически стоит давать в консоли перед установкой любых новых пакетов. Вторая команда устанавливает софт, необходимый для сборки, третья — библиотеки для С++.

Debian 9, также как и Ubuntu, использует пакетный менеджер **apt**. Для обновления системы необходимо набрать несколько команд. Но выполнять их нужно от имени суперпользователя:

$ su

# apt update

# apt full-upgrade

Работая в других дистрибутивах, мы привыкли использовать sudo для запуска программ от имени суперпользователя. Но в Debian sudo не установлена по умолчанию. Поэтому, перед тем как что-то настраивать, давайте ее инсталлируем:

# apt install sudo

Затем добавим текущего пользователя (например, пользователя student) в группу wheel:

# usermod –a –G sudo student

Альтернативный вариант подключения sudo в системе Debian - не указывать пароль root во время установки системы.

В ALTLinux используется пакет gcc-c++-common либо пакет gcc-common.

Установка с/с++ в ALTLinux:

apt-get update

apt-get install gcc5-c++

Последняя команда может быть заменена на следующую:

apt-get install gcc5

Последние версии Fedora будут иметь компилятор GCC по умолчанию. Для установки компилятора GCC используйте следующую команду в терминале:

yum install gcc-c++

или команду

yum install -y gcc gcc-c++ autoconf automake

либо команду

dnf install gcc-c++

Отметим, что dnf является более современным менеджером пакетов и используется вместо yum. При этом синтаксис команд dnf и yum практически совпадает.

Компилятор clang на Fedora работает как для C++, так и для C, и установка такая же, как и для C:

# sudo dnf install clang

Единственная разница при компиляции заключается в том, что вам нужно использовать команду clang++:

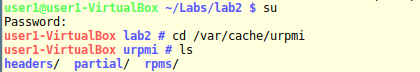
# clang++ -std=c++14 your\_source.cpp -o your\_binary

Это даст тот же результат, что и команда с g++. Чтобы увидеть больше вариантов clang, смотрите страницу руководства:

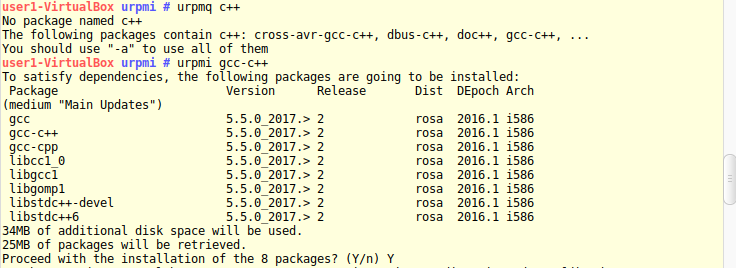
# man clang

При установке российской операционной системы ROSA Linux (например, с помощью дистрибутива **ROSA.FRESH.KDE.R10.i586.iso** или дистрибутива **ROSA.FRESH.KDE.R10.x86\_64.uefi.iso**) необходимо задать пароль как для суперпользователя root, так и для обычного пользователя (например, пользователя User1 с логином user1). После установки дистрибутива ROSA Linux оказалось, что мы можем использовать в консоли команды cc и gcc для запуска компилятора языка Си, а для языка Си++ команда g++ не поддерживается. Следовательно, для установки необходимых нам компиляторов будем использовать утилиты для управления пакетами.

Утилита urpmi (англ. User RedHat Package Manager Installer) - это инструмент управления пакетами Mageia / ROSA Linux / Mandriva Linux для установки, удаления, обновления и запросов пакетов программного обеспечения на локальных или удаленных (сетевых) носителях. Он расширяет возможности диспетчера пакетов RPM. Для работы с пакетами перейдем в режим суперпользователя:



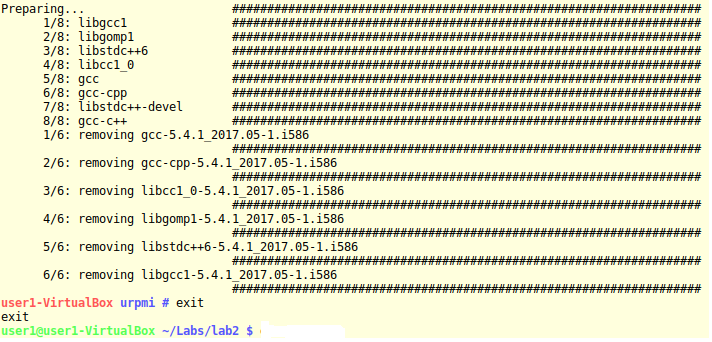
Используем команду urpmq утилиты urpmi чтобы найти пакеты, содержащие файлы, имена которых включают в себя подстроку c++:



Нам подходит имя файла gcc-c++, устанавливаем его командой urpmi gcc-c++ (см. выше). При этом будут установлены все необходимые библиотеки (все зависимости):



При установке будут обнаружены и удалены лишние или устаревшие программы/пакеты:



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.

**3. ОБОРУДОВАНИЕ**

Персональный компьютер типа IBM PC Pentium, операционная система Linux, файл МУ\_ЛР\_3.doc, содержащий методические указания к лабораторной работе, каталог lab3 c файлами hello.c, add.c, add.cpp, comf1, comf2, comf3, comf4, comf5 и clg.

**4. ЗАДАНИЕ НА РАБОТУ**

4.1. Ознакомиться с командным интерфейсом оаперационной системы Linux.

4.2. Переписать в личный каталог командные процедуры для компиляции, компоновки и выполнения программ. Выполнить отладку и проверку работы командных процедур.

**5. ПОРЯДОК ВЫПОЛНЕНИЯ РАБОТЫ**

5.1. Загрузить операционную систему Linux и после появления приглашения login войти в систему.

5.2. Запустить эмулятор консольного окна и с помощью команды pwd вывести в консольное окно путь к начальному (домашнему) каталогу пользователя. Определить, какой командный интерпретатор выполняет вводимые с консоли команды.

5.3. Запустить поочередно остальные интерпретаторы, используя команды sh, csh, bash, ksh. Для каждого интерпретатора командой set вывести значения параметров среды и найти командные процедуры (скрипты) инициализации, устанавливающие значения этих параметров при входе пользователя в систему (скрипты .profile или .login) и при запуске очередного интерпретатора (скрипты типа .\*shrc). Проверить выполнение команд (help, history и др.), а затем командой exit завершить работу очередного интерпретатора.

5.4. Перейти в каталог lab3 и ознакомиться с содержимым находящихся в нем текстовых файлов hello.c, add.c и add.cpp, содержащих программы, написанные на языках C и C++ соответственно. Посредством команд gcc и g++ откомпилировать и выполнить эти программы.

5.5. Создать в каталоге lab3 подкаталог lab31. Сделать каталог lab31 текущим и выполнить копирование в него текстовых файлов hello.c, add.c и add.cpp.

5.6. С помощью редактора ed создать командную процедуру comf1. Выполнить командную процедуру comf1, используя команду sh, сначала с флагом -v, а затем с флагом -x. Если ошибок не обнаружено, выполнить запуск процедуры без использования флагов. Повторить выполнение командной процедуры comf1, предварительно сделав ее исполнимой посредством команды chmod.

5.7. Переписать из родительского каталога в текущий каталог командные процедуры comf2, comf3, comf4, comf5, clg и повторить для них пункт 5.6. Для проверки работы скриптов использовать копии файлов hello.c, add.c и add.cpp, находящихся в каталоге lab3. При отладке скрипта clg объяснить каждую из выполняемых в скрипте проверок. Продемонстрировать преподавателю работу скрипта clg.

**6. ОФОРМЛЕНИЕ ОТЧЕТА**

Отчет должен содержать:

* тексты командных процедур;
* примеры выполнения скриптов.

**7. КОНТРОЛЬНЫЕ ВОПРОСЫ**

7.1. Какие значения имеют переменные среды PS1 и PS2 в каждом из командных интерпретаторов системы Linux?

7.2. Какое значение имела переменная PATH среды при выполнении лабораторной работы?

7.3. Какие формальные параметры имеет скрипт?

7.4. Укажите назначение различных флажков команд сс и gcc.

7.5. Как работает в скриптах цикл с перечислением?

7.6. Укажите назначение различных флажков команды test?

7.7. Как завершить работу скрипта comf5?

7.8. Перечислите команды строкового редактора ed.

7.9. Как выполняется отладка командных скриптов?

7.10. Какое назначение скриптов инициализации оболочки системы Linux?

**БИБЛИОГРАФИЧЕСКИЙ СПИСОК**

1. Робачевский, А.М. Операционная система UNIX : учеб.пособие / А.М.Робачевский, С.А.Немнюгин, О.Л.Стесик .— 2-е изд.,перераб.и доп. — СПб. : БХВ-Петербург, 2005 .— 635с.

2. Магда, Ю.С. UNIX для студента / Ю.С.Магда .— СПб. : БХВ-Петербург, 2007 .— 480с.

3. Рочкинд, M.Д. Программирование для UNIX / М.Д.Рочкинд;пер.с англ.под общ.ред.В.В.Вшивцева .— 2-е изд. — СПб. : БХВ-Петербург, 2005 .— 704с.

4. Моли, Molay B. Unix/Linux:теория и практика программирования : пер.с англ. / Б.Моли .— М. : КУДИЦ-ОБРАЗ, 2004 .— 576с.

5. Современные операционные системы/ Э. С. Таненбаум, Х. Бос. – 4-е изд. – СПб. : Питер, 2017. – 1120 с.

6. 15. Таненбаум Э. С. Современные операционные системы – 3-е изд. – СПб. : Питер, 2012. – 1120 с.

7. Реймонд, Э.С. Искусство программирования для Unix / Э.С.Реймонд;пер.с англ.и ред.В.А.Швеца .— М.и др. : Вильямс, 2005 .— 543с.

8. Стивенс У. UNIX: взаимодействие процессов. – СПб. : Питер, 2002. – 576 с.

9. Митчелл М., Оулдем Д., Самьюэл А. Программирование для Linux. Профессиональный подход. - М.: "Вильямс", 2002. - 288 с.

10. К.Кристиан. Введение в операционную систему UNIX. - М.: Финансы и статистика, 1985.

11. Р.Готье. Руководство по операционной системе UNIX. - М.: Финансы и статистика, 1986.

12. М.Банахан, Э.Раттер. Введение в операционную систему UNIX. - М.: Радио и связь, 1986.

13. П.Браун. Введение в операционную систему UNIX. - М.: Мир, 1987.

Лабораторная работа № 4

Доступ к файловой системе через интерфейс системных вызовов

**1. ЦЕЛЬ И ЗАДАЧИ РАБОТЫ**

Ознакомление с интерфейсом системных функций системы Linux и получение практических навыков программирования ввода-вывода на низком уровне, а также навыков отладки консольных приложений на языке C как в командной строке, так и в интегрированной среде CodeBlocks.

**2. ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ**

*2.1. 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.

*2.2. 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.

*2.3.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.

*2.4. 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;

}

*2.5. 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;

}

*2.6. 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; }

*2.7. 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;

}

*2.8. Компиляция и отладка программ в среде CodeBlocks*

Code::Blocks — это бесплатная кроссплатформенная среда разработки на языке C/C++. На данный момент это лучшая бесплатная среда разработки на языке Си.

В качестве отладчика по умолчанию CodeBlocks использует GDB — самый популярный отладчик для языка Си. Поддерживается также отладчик MS CDB.

Имеются следующие основные возможности редактора кода: подсветка синтаксиса, сворачивание блоков кода, автодополнение кода.

Установка среды в Ubuntu выполняется по аналогии с установкой обычных приложений:

sudo apt install codeblocks

После установки среды CodeBlocks рекомендуется поместить значок этого приложения в панель "Избранное" (рис. 1).

Отладка приложений на языке Си может выполняться двумя способами - с использованием и без использования проектов. В последнем случае находятся только ошибки компиляции, однако этот способ наиболее простой. Рассмотрим этот способ на примере отладки программы read-file.c. Загрузить этот файл в окно редактора можно при запуске среды из командной строки:

codeblocks read-file.c

Если среда уже запущена, для загрузки этого файла в окно редактора выполняем команды меню File -> Open -> read-file.c (рис. 2). После этого нажимаем кнопку Open (справа сверху). Далее уменьшаем размер шрифта в окне редактора посредством его масштабирования - при нажатой клавише Ctrl крутим колесико мыши (рис. 3).

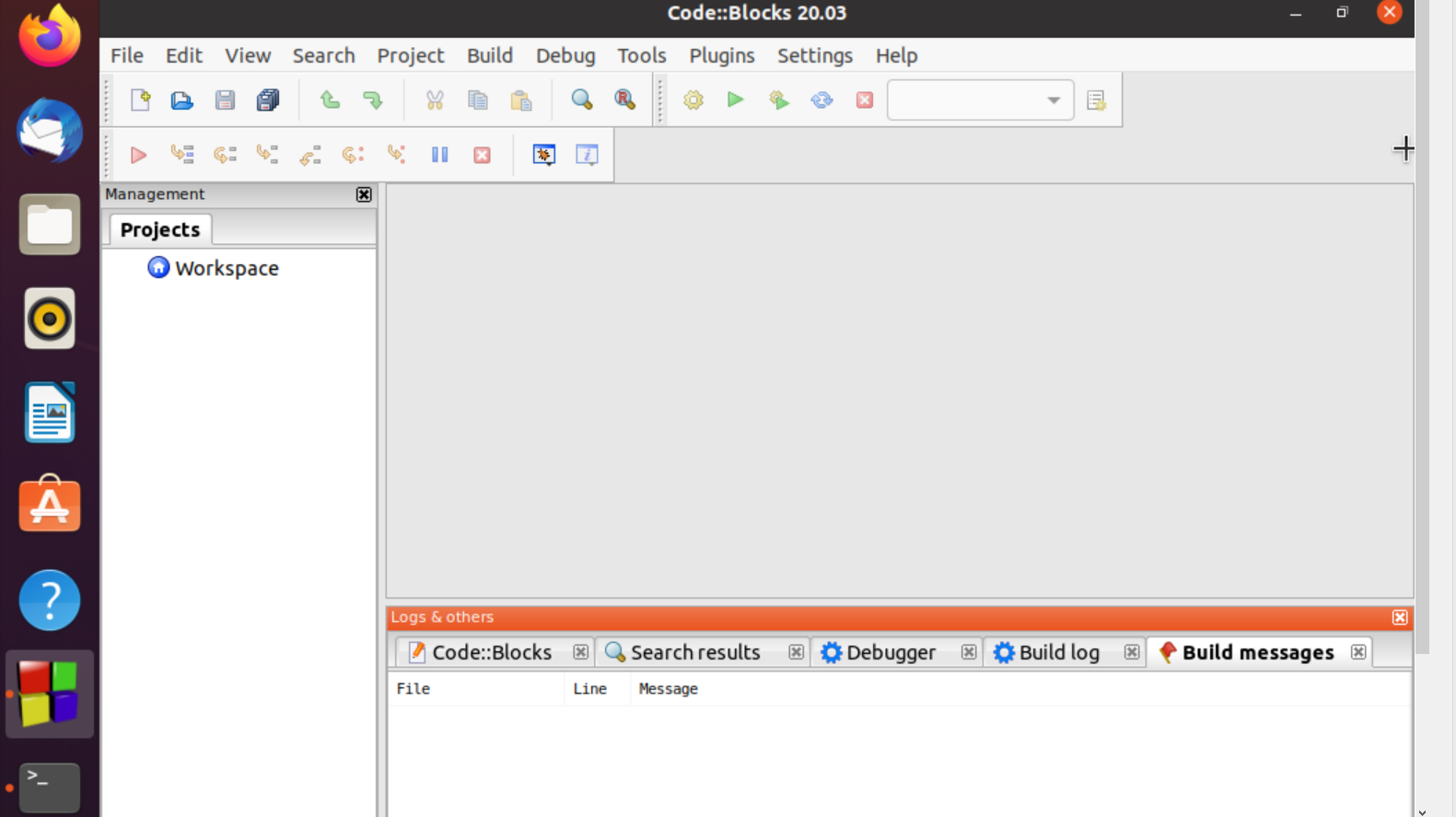


Рис. 1. Главное окно среды CodeBlocks

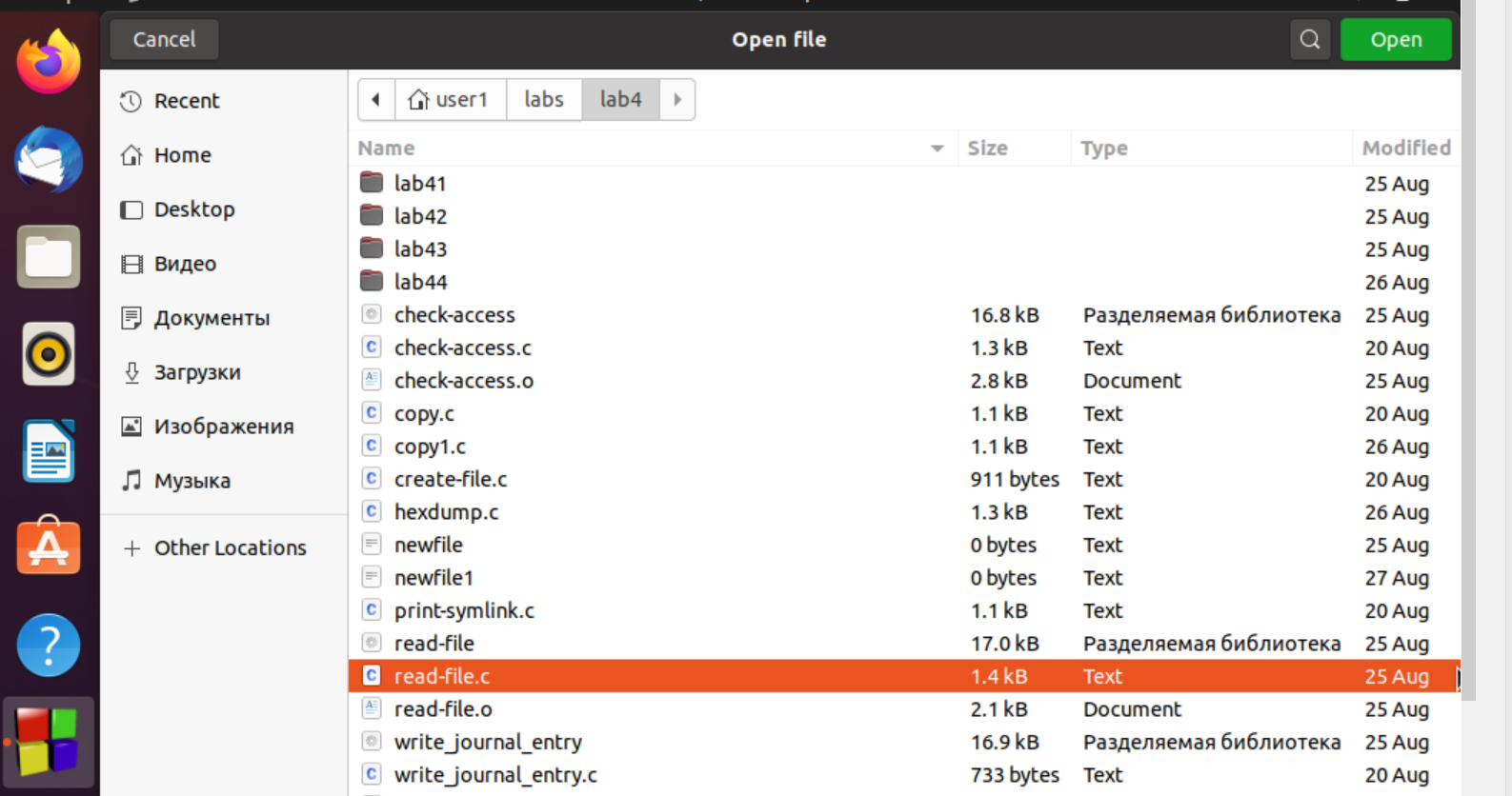


Рис. 2. Загрузка файла в окно редактирования

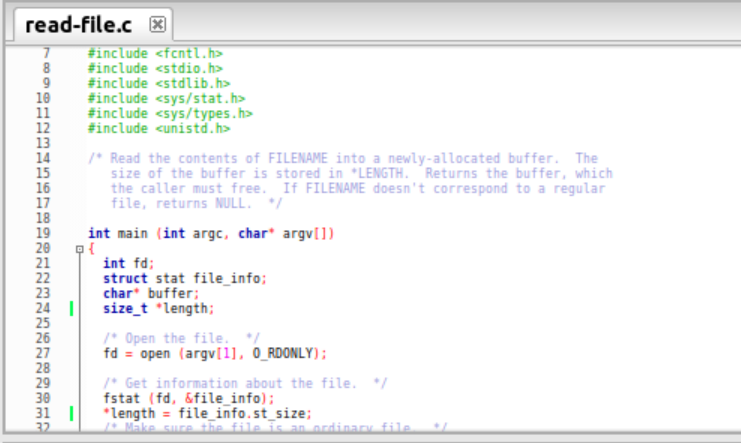


Рис. 3. Файл в окне редактирования

Для компиляции файла в окне редактора нажимаем Shift+Ctl+F9. Результаты компиляции появятся в окне Build log (рис. 4).

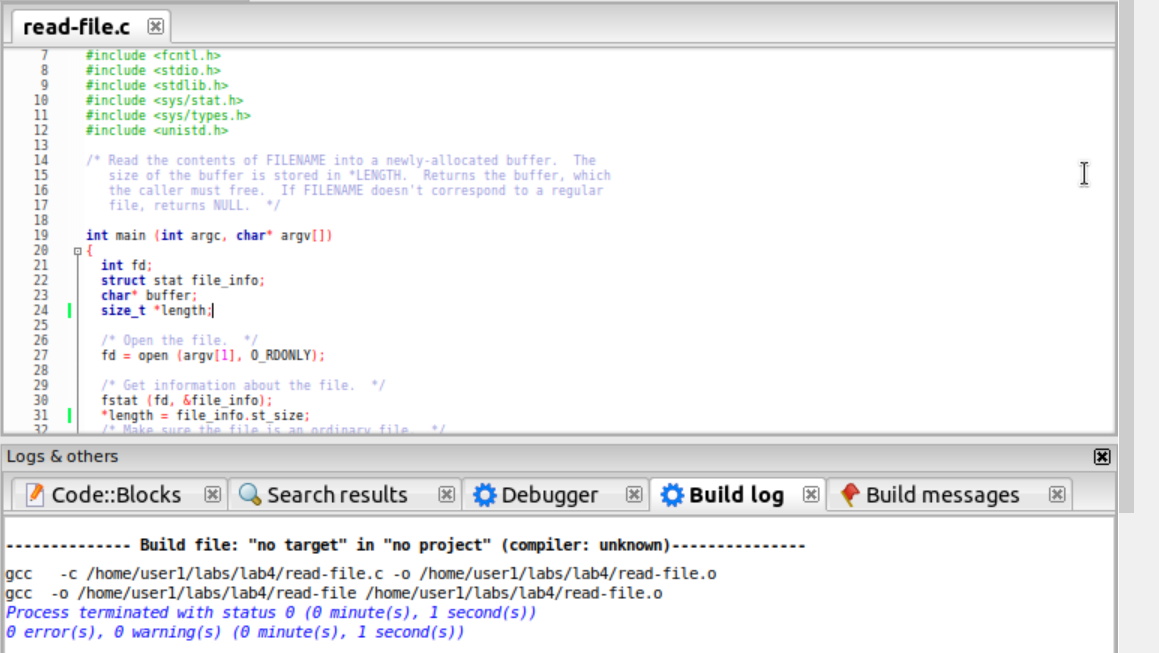


Рис. 4. Результаты компиляции программы

В этой программе необосновано используется указатель length, что усложнило синтаксис кода. Для упрощения кода откажемся от использования указателя - уберем звездочку в объявлении length (строка 24). С помощью Ctrl+S сохраним измененный код и с помощью Shift+Ctl+F9 запустим компиляцию. В окне Build messages появились сообщения об ошибках - в строках 31, 40 и 42 имеются лишние звездочки (рис. 5). Удалим их и сохраним изменения. После этого компиляция пройдет без ошибок.

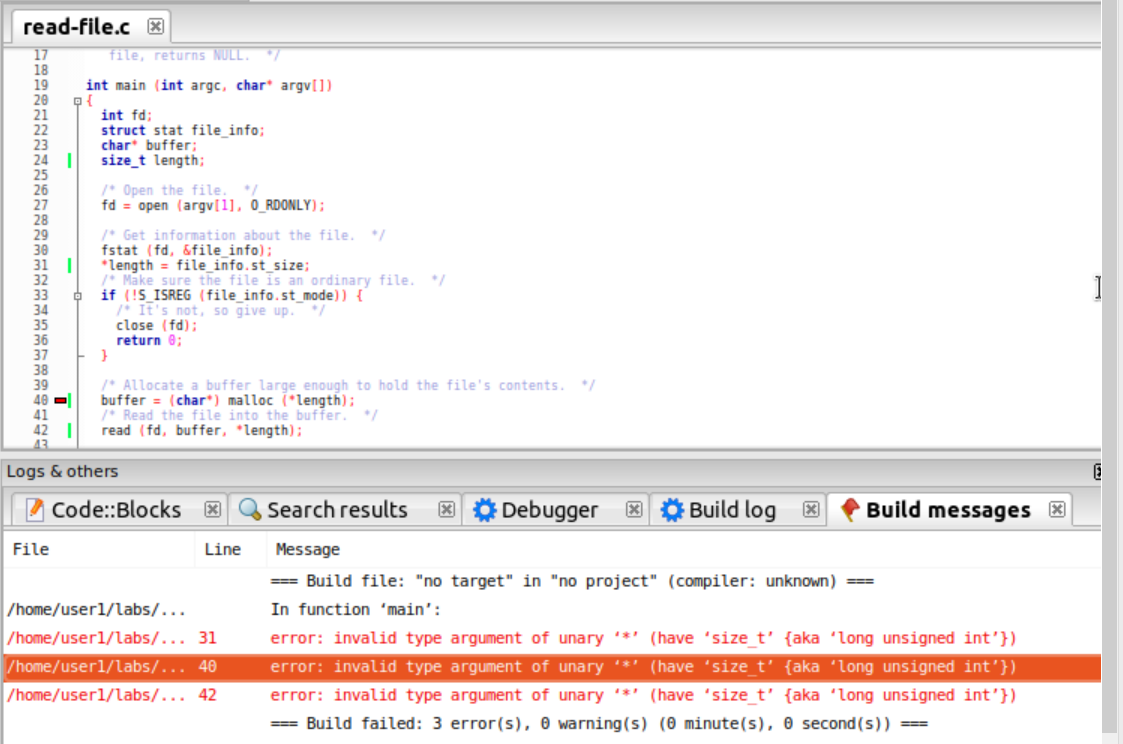


Рис. 5. Сообщения об ошибках

Рассмотрим теперь отладку программ с использованием проектов. В этом случае мы сможем компилировать, создавать загрузочный модуль и запускать его на выполнение.

Создание проекта требует выполнения ряда шагов. Рассмотрим их.

Выполняем следующую последовательность команд: File -> New -> Project. Появляется окно шаблонов для различных видов создаваемых приложений (рис. 6).



Рис. 6. Окно для выбора шаблона проекта

Выбирает тип проекта - Console Application - и нажимаем кнопку Go. Появится окно для выбора языка программирования. Выбираем язык C и нажимаем Next (рис. 7).



Рис. 7. Окно для выбора языка программирования

Выбираем каталог для размещения файла проекта и имя проекта (рис. 8).

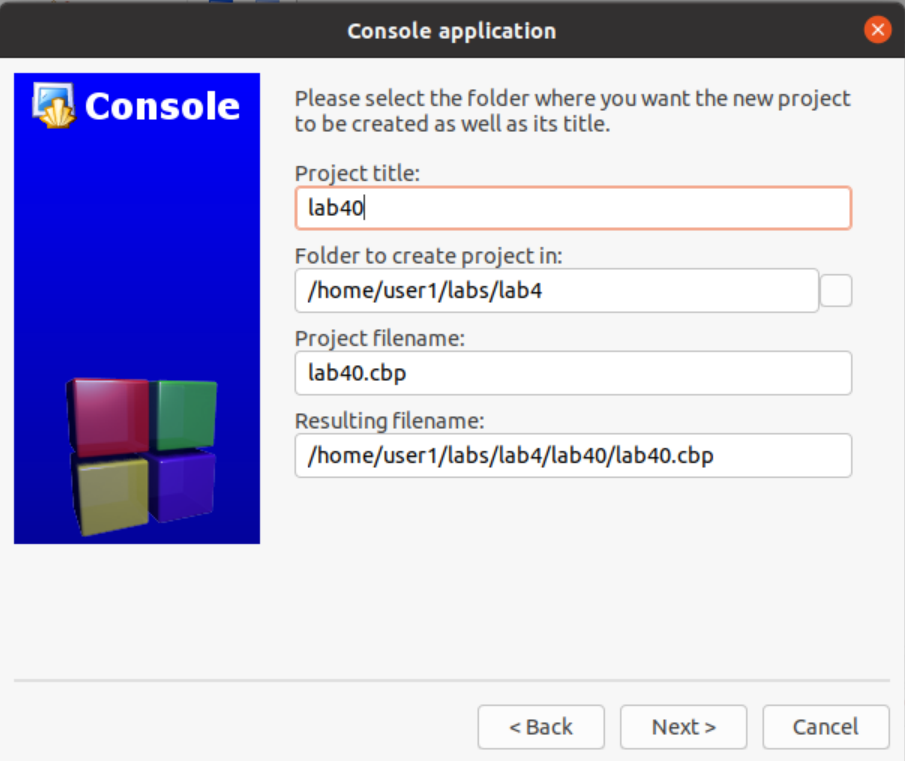


Рис. 8. Окно для выбора каталога и имени проекта

Далее выбираем компилятор и цель для проекта – создание Debug и/или Realease конфигурации (рекомендуется ограничиться конфигурацией Debug) (рис. 9).

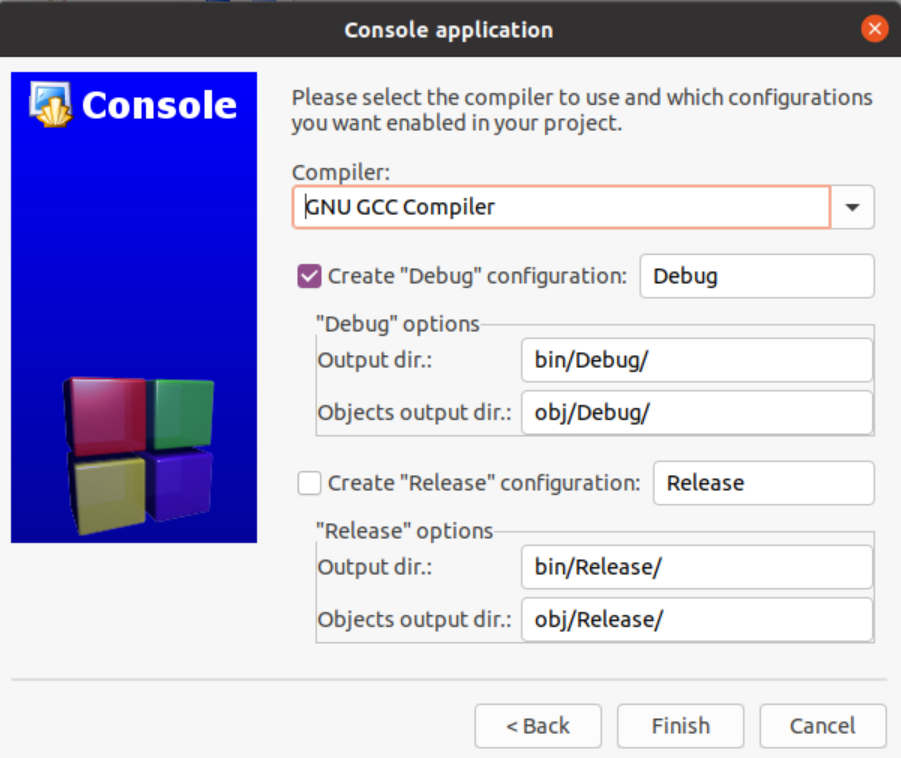


Рис. 9. Окно для выбора компилятора и цели проекта

При создании проекта в каталоге проекта создается файл main.c. Поскольку мы используем существующие исходные файлы программ, файл main.c удаляем из проекта. Для этого используем контекстное меню – перемещаем указатель мыши на имя файла, нажимаем правую клавишу мыши и выбираем опцию Remove file from project (рис. 10).

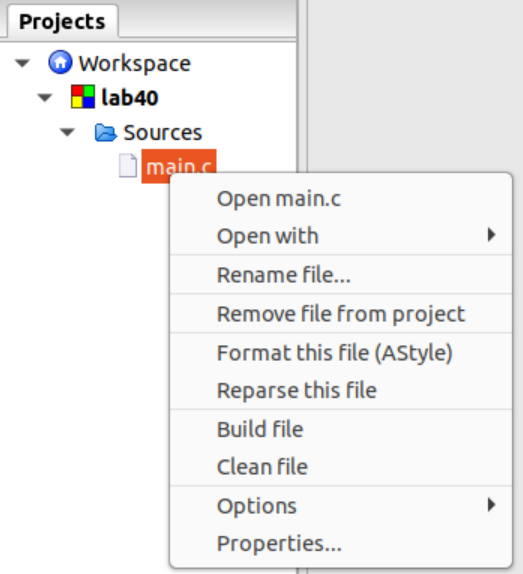


Рис. 10. Удаление файла main.c из проекта

Далее ставим указатель мыши на имя проекта, например, lab41, открываем контекстное меню и выбираем в нем опцию Add files (рис. 11). Находим в каталоге lab4 необходимый нам файл, например, файл check-access.c и нажимаем кнопку открыть (справа сверху). Файл появится в окне проектов. По двойному клику на имени файла загружаем этот файл в окно редактора (рис. 12).

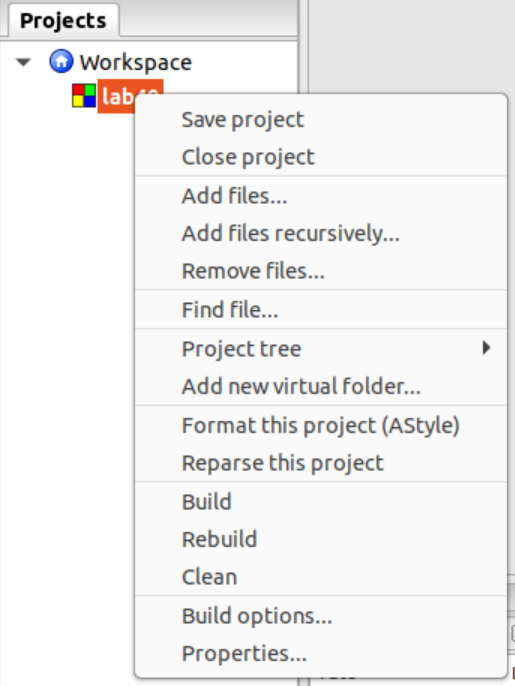


Рис. 11. Добавление файла в проект

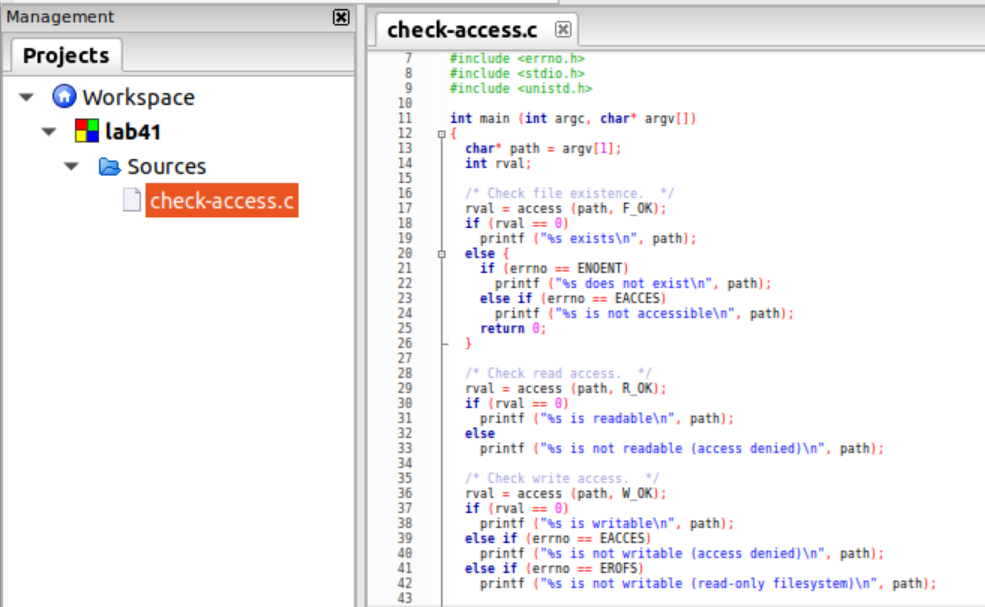


Рис. 12. Открытие файла в окне редактора

Некоторые программы требуют задание аргументов командной строки. Например, компиляция и выполнение программы check-access.c в терминале выполняется следующим образом:

gcc check-access.c

./a.out create-file.c

Другими словами, наша программа должна проверить права доступа у файла create-file.c. Чтобы указать этот файл, выберем в в меню Project опцию Set programs argument (рис. 13).

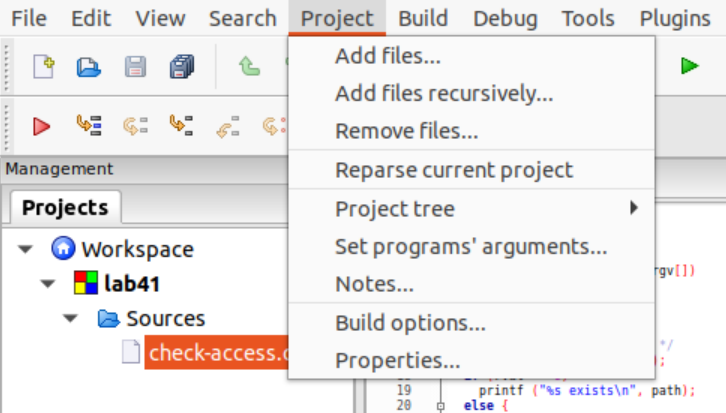


Рис. 13. Окно меню Project

При задании аргумента программы надо учесть тот факт, что при выполнении программы текущим каталогом будет каталог lab41, а файл create-file.c находится в lab4, который является родительским каталогом для lab41. Поэтому мы задаем имя ../create-file.c (рис. 14).

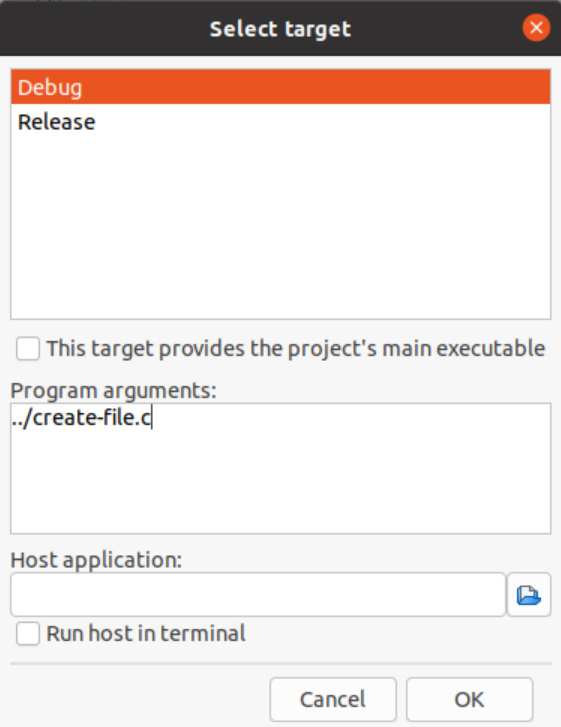


Рис. 14. Окно задания аргумента программы

При выполнении программы открывается окно, в которое помещаются результаты выполнения программы (рис. 15).

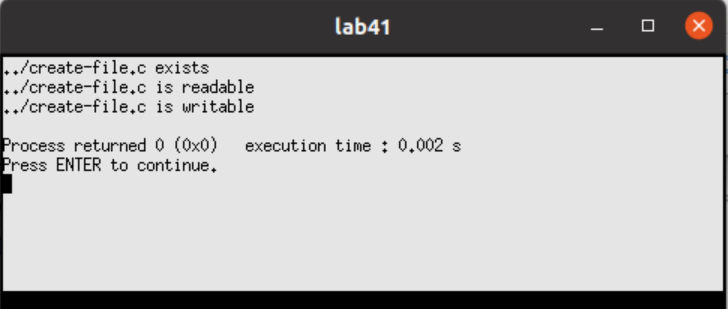


Рис. 15. Окно с результатами выполнения программы

На рис. 15 показано окно проектов, созданных при выполнении данной лабораторной работы.

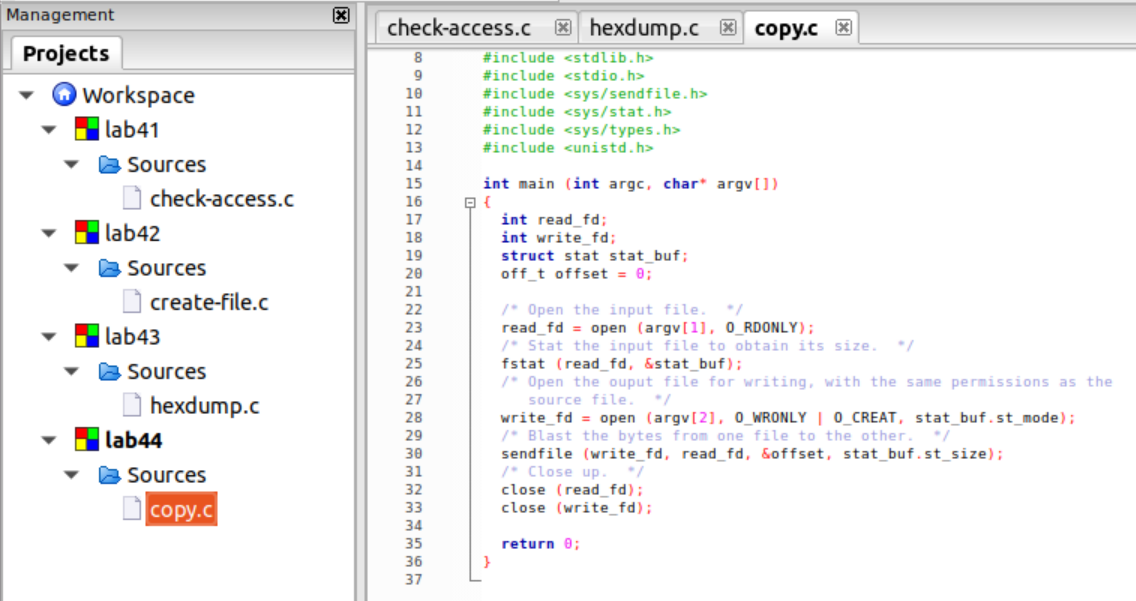


Рис. 16. Окно с проектами

**3. ОБОРУДОВАНИЕ**

Персональный компьютер типа IBM PC Pentium, операционная система Linux, файл МУ\_ЛР\_4.doc, содержащий методические указания к лабораторной работе, каталог lab4 c файлами check-access.c, copy.c, create-file.c, hexdump.c, print-symlink.c, read-file.c, write\_journal\_entry.c.

**4. ЗАДАНИЕ НА РАБОТУ**

Перекомпилировать и выполнить тестирование программ на языке Си, использующих системные вызовы для доступа к файловой системе как в командной строке, так и в интегрированной среде CodeBlocks.

**5. ОФОРМЛЕНИЕ ОТЧЕТА**

Отчет должен содержать:

* тексты программ;
* примеры компиляции и отладки программ как в командной строке, так и в интегрированной среде CodeBlocks.

**7. КОНТРОЛЬНЫЕ ВОПРОСЫ**

7.1. Какие системные и библиотечные функции ввода-вывода Вы знаете?

7.2. Как работает системная функция readv()?

7.3. Как обрабатываются ошибки, возникающие при выполнении системных функций?

7.4. Какие коды ошибок могут возникнуть при выполнении системного вызова access()?

7.5. Как создать файл посредством системного вызова open()?

7.6. Какая маска прав доступа к файлу установлена по умолчанию?

7.7. Чем отличаются форматы текстовых файлов в Linux и в DOS/Windows?

7.8. Какие поля содержит структура stat?

7.9. Какие системные вызовы можно использовать при работе с символическими ссылками?

7.10. Как используется системная функция sendfile()?

**БИБЛИОГРАФИЧЕСКИЙ СПИСОК**

1. Робачевский, А.М. Операционная система UNIX : учеб.пособие / А.М.Робачевский, С.А.Немнюгин, О.Л.Стесик .— 2-е изд.,перераб.и доп. — СПб. : БХВ-Петербург, 2005 .— 635с.

2. Магда, Ю.С. UNIX для студента / Ю.С.Магда .— СПб. : БХВ-Петербург, 2007 .— 480с.

3. Рочкинд, M.Д. Программирование для UNIX / М.Д.Рочкинд;пер.с англ.под общ.ред.В.В.Вшивцева .— 2-е изд. — СПб. : БХВ-Петербург, 2005 .— 704с.

4. Моли, Molay B. Unix/Linux:теория и практика программирования : пер.с англ. / Б.Моли .— М. : КУДИЦ-ОБРАЗ, 2004 .— 576с.

5. Галатенко, В.А. Программирование в стандарте POSIX : Курс лекций:Учеб.пособие / В.А.Галатенко; Под ред.В.Б.Бетелина .— М. : Интернет-Университет Информационных Технологий, 2004 .— 560с.

6. Хорвиц, Д. Unix-системы.От проектирования до сопровождения : пер.с англ. / Д.Хорвиц .— М. : DiaSoft, 2004 .— 608c.

7. Реймонд, Э.С. Искусство программирования для Unix / Э.С.Реймонд;пер.с англ.и ред.В.А.Швеца .— М.и др. : Вильямс, 2005 .— 543с.

8. Стивенс У. UNIX: взаимодействие процессов. – СПб. : Питер, 2002. – 576 с.

9. Митчелл М., Оулдем Д., Самьюэл А. Программирование для Linux. Профессиональный подход. - М.: "Вильямс", 2002. - 288 с.

10. К.Кристиан. Введение в операционную систему UNIX. - М.: Финансы и статистика, 1985.

11. Р.Готье. Руководство по операционной системе UNIX. - М.: Финансы и статистика, 1986.

12. М.Банахан, Э.Раттер. Введение в операционную систему UNIX. - М.: Радио и связь, 1986.

13. П.Браун. Введение в операционную систему UNIX. - М.: Мир, 1987.

Лабораторная работа № 5

Работа с процессами в командной и программной средах

операционной системы Linix

**1. ЦЕЛЬ И ЗАДАЧИ РАБОТЫ**

Ознакомление с командами и системными функциями операционной системы Linux, предназначенными для работы с процессами, получение практических навыков программирования и отладки программ процессов на низком уровне в консольной среде.

**2. ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ**

*2.1. 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.

*2.2. 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.

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;

}

*2.3. 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 \*/

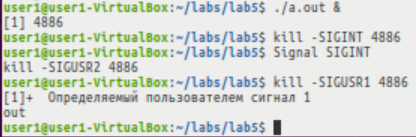
while(1)

pause();

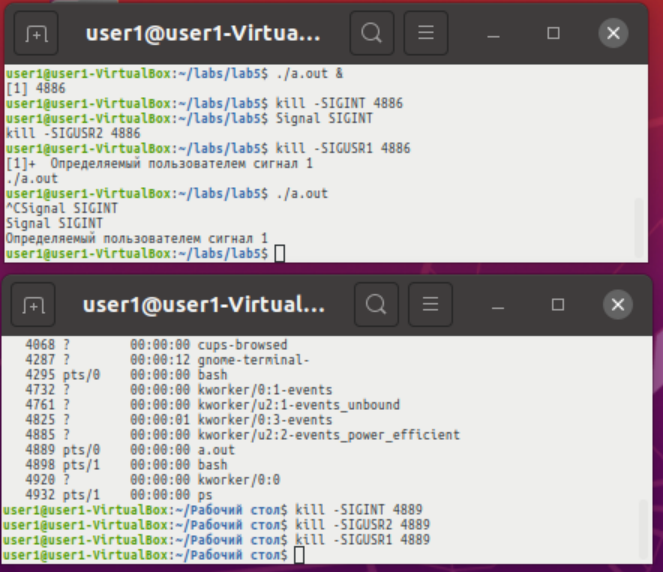
}

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:



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:



*2.4. 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.

*2.5. 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 \*/

}

*2.6. Отладка программ с помощью отладчика GDB*

При отладке консольных программ на языке Си удобно использовать отладчик GDB. В качестве примера будем использовать рассмотренную ранее программу clock.c:

#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(;;)

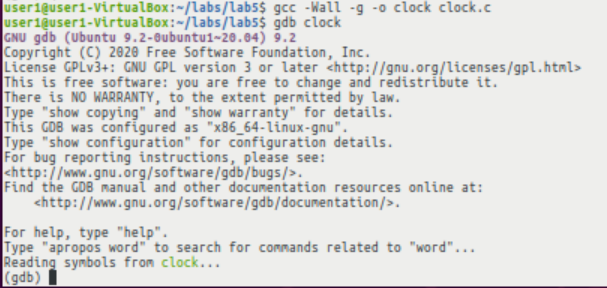
pause();

return();

}

Для комфортной отладки сначала нужно собрать программу с правильными флагами. Во-первых, необходимо указать флаг -g. Иначе программа будет собрана без отладочных символов и отлаживать ее придется в ассемблерном коде (что, впрочем, вполне выполнимо для программ, написанных на чистом Си). Во-вторых, рекомендуется отключить оптимизацию программы при помощи флага -o 0, иначе некоторые переменные окажутся, что называется, optimized out, и вы не сможете посмотреть их значения при отладке. Если при компиляции возникли предупреждения (warning), их практически все (all) можно отключить с помощью опции - Wall.

Ниже представлен пример компиляции программы и запуска ее в отладчике GDB:



Появилось приглашение (gdb) отладчика к вводу команд. Для выполнения команды в gdb можно вводить либо всю команду целиком, либо только первые несколько букв. То есть, h, he и help или r, ru и run — это все одни и те же команды. Обычно используют короткие версии, поскольку соответствующие длинные версии очевидны. Однако при первом использовании команды рекомендуем указавать ее полное имя. Если мы просто нажмем клавишу Enter без ввода имени команды, то выполнится еще раз предыдущая команда.

После запуска отладчика программа clock будет находиться в остановленном состоянии и мы не сможем выполнить операцию, например, step (выполнить один шаг с заходом в тело вызываемой функции) или операцию next (выполнить шаг без захода в функцию).

Чтобы перевести программу в состояние исполнения, необходимо воспользоваться командой

(gdb) run

По этой команде программа выполнит код до конца и снова остановится. Поэтому перед выполнением команды run необходимо установить точку останова, например, остановиться при входе в тело функции main:

(gdb) break main

Breakpoint 1 at 0x4005ab: file 1.c, line 7.

(gdb) run

При отладке удобно использовать номера строк программы, которые можно вывести на экран по команде

(gdb) list

Для того, чтобы посмотреть, на какой мы сейчас строке, нужно написать f. Чтобы узнать, какие переменные (локальные) сейчас инициализированны в программе, нужно написать команду info locals. Чтобы вывести только одну переменную, нужно написать print имя\_переменной. Мы можем также изменить переменную с помощью set.

Мы можем также следить за переменными с помощью watch имя\_переменной. Чтобы посмотреть, какие в данный момент есть breakpoints (точки останова), нужно написать info breakpoints. Удаляем точку останова командой del breakpoint\_num.

Чтобы дойти до следующей точки останова, нужно выполнить команду continue. Чтобы продолжить выполнение функции и остановить программу, когда она (функция) завершится, нужно написать finish или fin. Отметим, что нельзя использовать finish в главном методе.

Чтобы завершить выполнение программы, нужно написать kill.

*2.7. Отладка программ с помощью отладчика GDB TUI*

Команда gdbtui позволяет запустить gdb в режиме TUI (текстовый интерфейс пользователя). Это дает пользовательский интерфейс, в котором экран будет разделен на несколько окон, например, в одном будет отображаться исходный код, в другом - окно управления, в котором можно вводить команды gdb и т. Д. При запуске отладки в окне исходного кода будет отмечена строка, в которой является текущей..

Вы можете запустить gdb в режиме TUI (Text User Interface) двумя способами: либо использовать команду gdbtui либо указать опцию -tui в команде gdb. При отладке в режиме TUI в верхнем окне будет отображаться исходный код программы (в том месте, которое сейчас выполняется), а ниже будет находиться консольное окно для ввода команд отладчика. Экран иногда будет портиться, что можно исправить: ввод Ctrl-L приведет к перерисовке экрана терминала.

В режиме текстового интерфейса пользователя -tui клавиши меняются так, что стрелки не работают нормально. Тем не менее, режим TUI показывает позицию исходного кода в верхней части терминала, что, по мнению большинства, полезно. Работу в режиме текстового интерфейса пользователя поддерживают функции библиотеки ReadLine.

**3. ОБОРУДОВАНИЕ**

Персональный компьютер типа IBM PC Pentium, операционная система Linux, файл МУ\_ЛР\_5.doc, содержащий методические указания к лабораторной работе, каталог lab5 c файлами clock.c, fork.c, fork-exec.c, new-signal.c, old-signal.c, parent-child.c, print-pid.c, system.c, timeout.c, zombie.c.

**4. ЗАДАНИЕ НА РАБОТУ**

4.1. Ознакомиться с командами интерпретатора команд, предназначенными для работы с процессами.

4.2. Перекомпилировать и выполнить тестирование и отладку программ на языке Си, использующих библиотечные функции и системные вызовы для создания и управления процессами.

**5. ПОРЯДОК ВЫПОЛНЕНИЯ РАБОТЫ**

5.1. Загрузить операционную систему Linux и после появления приглашения login войти в систему.

5.2. Ознакомиться с командами интерпретатора команд, предназначенными для работы с процессами. Проверить их работу в консольной среде.

5.3. Запустить эмулятор консольного окна. Перейти в каталог lab5 и ознакомиться с содержимым находящихся в нем файлов clock.c, fork.c, fork-exec.c, new-clock.c, new-clock.c, new-signal.c, old-signal.c, parent-child.c, print-pid.c, system.c, timeout.c, zombie.c, содержащих программы, написанные на языке Cи.

5.4. Посредством команд gcc откомпилировать и выполнить эти программы. На различных примерах проверить правильность работы программ.

5.5. Получить навыки отладки программ с помощью отладчиков gdb и gdbtui.

**6. ОФОРМЛЕНИЕ ОТЧЕТА**

Отчет должен содержать:

* тексты программ и результаты их выполнения;
* примеры отладки программ.

**7. КОНТРОЛЬНЫЕ ВОПРОСЫ**

7.1. Какие стили опций можно использовать в команде ps?

7.2. Как работают команды top, kpm, xsysinfo, xosview?

7.3. В каких состояниях может находиться процесс?

7.4. Как используется в программах сигнал SIGALRM?

7.5. Что наследует потомок, созданный вызовом fork()?

7.6. Что называется диспозицией сигнала? Как она задается?

7.7. Какое определение имеет библиотечная функция signal()?

7.8. Какие поля содержит структура sigaction?

7.9. Какие системные вызовы работают с переменной типа sigset\_t?

7.10. Какой макрос извлекает номер сигнала из кода завершения процесса?

**БИБЛИОГРАФИЧЕСКИЙ СПИСОК**

1. Робачевский, А.М. Операционная система UNIX : учеб.пособие / А.М.Робачевский, С.А.Немнюгин, О.Л.Стесик .— 2-е изд.,перераб.и доп. — СПб. : БХВ-Петербург, 2005 .— 635с.

2. Магда, Ю.С. UNIX для студента / Ю.С.Магда .— СПб. : БХВ-Петербург, 2007 .— 480с.

3. Рочкинд, M.Д. Программирование для UNIX / М.Д.Рочкинд;пер.с англ.под общ.ред.В.В.Вшивцева .— 2-е изд. — СПб. : БХВ-Петербург, 2005 .— 704с.

4. Моли, Molay B. Unix/Linux:теория и практика программирования : пер.с англ. / Б.Моли .— М. : КУДИЦ-ОБРАЗ, 2004 .— 576с.

5. Галатенко, В.А. Программирование в стандарте POSIX : Курс лекций:Учеб.пособие / В.А.Галатенко; Под ред.В.Б.Бетелина .— М. : Интернет-Университет Информационных Технологий, 2004 .— 560с.

6. Хорвиц, Д. Unix-системы.От проектирования до сопровождения : пер.с англ. / Д.Хорвиц .— М. : DiaSoft, 2004 .— 608c.

7. Реймонд, Э.С. Искусство программирования для Unix / Э.С.Реймонд;пер.с англ.и ред.В.А.Швеца .— М.и др. : Вильямс, 2005 .— 543с.

8. Стивенс У. UNIX: взаимодействие процессов. – СПб. : Питер, 2002. – 576 с.

9. Митчелл М., Оулдем Д., Самьюэл А. Программирование для Linux. Профессиональный подход. - М.: "Вильямс", 2002. - 288 с.

Лабораторная работа № 6

Создание неименованных каналов и приложений типа клиент-сервер в системе Linux

**1. ЦЕЛЬ И ЗАДАЧИ РАБОТЫ**

Ознакомление с неименованными каналами системы Linux, предназначенными для обмена данными между родственными процессами, а также получение практических навыков создания сервера файлов в рамках приложений типа клиент-сервер.

**2. ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ**

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 неименованные каналы.

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.

*2.1. 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.

*2.2. 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;

}

*2.3. 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.

*2.4. 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.

*2.5. Двусторонние неименованные каналы*

В некоторых UNIX-системах реализованы двусторонние каналы. В UNIX SVR4 это обеспечивается самой функцией pipe, а во многих других ядрах – функцией soketpair. При использовании pipe двусторонний канал получается из объединения двух односторонних, т.е. вызовом pipe() создается фактически два односторонних, не зависимых друг от друга канала. Все данные, записываемые в fd[1], будут доступны для чтения из fd[0], а данные, записываемые в fd[0], будут доступны для чтения из fd[1].

Программа в листинге 7 иллюстрирует использование одного двустороннего канала для двусторонней передачи информации. В этой программе сначала создается двусторонний канал, а затем делается системный вызов fork(). Породивший процесс записывает символ p в канал, а затем считывает из канала данные. Дочерний процесс ждет три секунды, считывает символ из канала, а потом записывает туда символ с. Задержка чтения для дочернего процесса позволяет породившему процессу вызвать read() первым – таким образом мы можем узнать, не будет ли породившим процессом считан обратно только что записанный им символ.

**Листинг 7. (*fduplex.c*) Двусторонняя связь через двусторонний канал**

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

int

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

{

int fd[2], n;

char c;

pid\_t childpid;

pipe(fd); /\* assumes a full-duplex pipe (e.g., SVR4) \*/

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

sleep(3);

if ( (n = read(fd[0], &c, 1)) != 1){

printf(child: read returned %d", n);

exit(EXIT\_FAILURE);

}

printf("child read %c\n", c);

write(fd[0], "c", 1);

exit(0);

}

/\* parent \*/

write(fd[1], "p", 1);

if ( (n = read(fd[1], &c, 1)) != 1){

printf(child: read returned %d", n);

exit(EXIT\_FAILURE);

}

printf("parent read %c\n", c);

exit(0);

}

*2.6. Клиент-серверное приложение с использованием popen()*

Для создания клиент-серверных приложений можно использовать и рассмотренную ранее библиотечную функцию popen(), которая создает канал и запускает другой процесс, записывающий данные в этот канал или считывающий их из него. В листинге 8 приведен пример такого приложения.

**Листинг 8. (*mainpopen.c*) Клиент-сервер с использованием popen()**

#include <stdio.h>

#include <string.h>

#define MAXLINE 80

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

{

size\_t n;

char buff[MAXLINE], command[MAXLINE];

FILE \*fp;

/\* считывание полного имени файла \*/

fgets(buff, MAXLINE, stdin);

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

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

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

snprintf(command, sizeof(command), "cat %s", buff);

fp = popen(command, "r");

/\* копирование из канала в стандартный вывод \*/

while (fgets(buff, MAXLINE, fp) != NULL)

fputs(buff, stdout);

fclose(fp);

exit(0);

}

Полное имя файла считывается из стандартного потока ввода, как и в программе в листинге 5. Формируется командная строка, которая передается popen(). Вывод интерпретатора команд или команды cat копируется в стандартный поток вывода. Одним из отличий этой реализации от приведенной в листинге 5, является отсутствие возможности формировать собственные сообщения об ошибках. Теперь мы полностью зависим от программы cat, а выводимые ею сообщения не всегда адекватны.

**3. ОБОРУДОВАНИЕ**

Персональный компьютер типа IBM PC Pentium, операционная система Linux, файл МУ\_ЛР\_unixSystems.doc, содержащий сборник методических указаний к лабораторным работам, каталог lab6 c файлами pipe.c, dup2.c, popen.c и подкаталог pipe c файлами mainpipe.c, client.c, server.c, fduplex.c, mainpopen.c.

**4. ЗАДАНИЕ НА РАБОТУ**

4.1. Ознакомиться с различными способами создания и использования неименованных каналов в обычных и клиент-серверных приложениях.

4.2. Перекомпилировать и выполнить тестирование обычных и клиент-серверных программ на языке Си, использующих неименованные каналы.

**5. ПОРЯДОК ВЫПОЛНЕНИЯ РАБОТЫ**

5.1. Загрузить операционную систему Linux и после появления приглашения login войти в систему.

5.2. Запустить эмулятор консольного окна. Перейти в каталог lab6 и ознакомиться с содержимым находящихся в нем файлов pipe.c, dup2.c, popen.c, а также файлов mainpipe.c, client.c, server.c, fduplex.c, mainpopen.c в подкаталоге pipe.

5.4. Посредством команд gcc откомпилировать и выполнить данные программы. На различных примерах проверить правильность их работы.

**6. ОФОРМЛЕНИЕ ОТЧЕТА**

Отчет должен содержать:

* тексты программ;
* результаты выполнения программ.

**7. КОНТРОЛЬНЫЕ ВОПРОСЫ**

7.1. Какие объекты IPC используются в различных системах?

7.2. Какое назначение функции fdopen?

7.3. Приведите примеры использования вызова dup2().

7.4. Напишите программу реализации программного канала между двумя дочерними процессами.

7.5. Какое назначение файлового сервера?

7.6. Приведите прототипы функций sprintf() и snprintf() и укажите их назначение.

7.7. В каких случаях процессы превращаются в зомби?

7.8. Как создается двусторонний канал?

7.9. Какими способами можно создать программный канал с утилитой или между двумя утилитами?

7.10. Приведите прототип функции popen().

**БИБЛИОГРАФИЧЕСКИЙ СПИСОК**

1. Робачевский, А.М. Операционная система UNIX : учеб.пособие / А.М.Робачевский, С.А.Немнюгин, О.Л.Стесик .— 2-е изд.,перераб.и доп. — СПб. : БХВ-Петербург, 2005 .— 635с.

2. Магда, Ю.С. UNIX для студента / Ю.С.Магда .— СПб. : БХВ-Петербург, 2007 .— 480с.

3. Рочкинд, M.Д. Программирование для UNIX / М.Д.Рочкинд;пер.с англ.под общ.ред.В.В.Вшивцева .— 2-е изд. — СПб. : БХВ-Петербург, 2005 .— 704с.

4. Моли, Molay B. Unix/Linux:теория и практика программирования : пер.с англ. / Б.Моли .— М. : КУДИЦ-ОБРАЗ, 2004 .— 576с.

5. Галатенко, В.А. Программирование в стандарте POSIX : Курс лекций:Учеб.пособие / В.А.Галатенко; Под ред.В.Б.Бетелина .— М. : Интернет-Университет Информационных Технологий, 2004 .— 560с.

6. Хорвиц, Д. Unix-системы.От проектирования до сопровождения : пер.с англ. / Д.Хорвиц .— М. : DiaSoft, 2004 .— 608c.

7. Реймонд, Э.С. Искусство программирования для Unix / Э.С.Реймонд;пер.с англ.и ред.В.А.Швеца .— М.и др. : Вильямс, 2005 .— 543с.

8. Стивенс У. UNIX: взаимодействие процессов. – СПб. : Питер, 2002. – 576 с.

9. Митчелл М., Оулдем Д., Самьюэл А. Программирование для Linux. Профессиональный подход. - М.: "Вильямс", 2002. - 288 с.

Лабораторная работа № 7

Создание именованных каналов и приложений типа клиент-сервер в системе Linux

**1. ЦЕЛЬ И ЗАДАЧИ РАБОТЫ**

Ознакомление с именованными каналами системы Linux, предназначенными для обмена данными между родственными и неродственными процессами, а также получение практических навыков создания сервера файлов в рамках приложений типа клиент-сервер.

**2. ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ**

*2.1. 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. (*mainfifo.c*) Функция main() для приложения типа клиент-сервер**

#include <sys/stat.h> // mkfifo

#include <stdlib.h> // exit

#include <stdio.h> // fgets, stdin, printf

#include <fcntl.h> // O\_RDONLY, O\_WRONLY

#include <string.h> // strlen

#include <errno.h> // errno, EEXIST

#include <wait.h> // waitpid

#include <unistd.h> // read, write, fork

#define FIFO1 "/tmp/fifo.1"

#define FIFO2 "/tmp/fifo.2"

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

#define MAXLINE 80

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))

printf("can't create %s", FIFO1);

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

unlink(FIFO1);

printf("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. (*mainfifo.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. (*mainfifo.c*) Функция server() для приложения типа клиент-сервер**

void

server(int readfd, int writefd)

{

int fd;

size\_t n;

char buff[MAXLINE+1];

/\* read pathname from IPC channel \*/

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

printf("end-of-file while reading pathname");

exit(EXIT\_FAILURE);

}

buff[n] = '\0'; /\* null terminate pathname \*/

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

/\* error: must tell client \*/

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

strerror(errno));

n = strlen(buff);

write(writefd, buff, n);

} else {

/\* open succeeded: copy file to IPC channel \*/

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 клиента. Можно было сделать и по другому: запускать только клиентскую программу, 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.

*2.4. Один сервер и несколько клиентов*

Преимущества канала FIFO проявляется более явно в том случае, когда сервер представляет собой некоторый длительно функционирующий процесс (например, демон), не являющийся родственным клиенту. Демон создает именованный канал с вполне определенным известным именем, открывает его на чтение, а запускаемые впоследствии клиенты открывают его на запись и отправляют демону команды и необходимые данные. Односторонняя связь в этом направлении (от клиента к серверу) легко реализуется с помощью FIFO, однако необходимость отправки данных в обратную сторону (от сервера к клиенту) усложняет задачу.

Пусть сервер создает канал с именем /tmp/fifo.serv. Из этого канала он считывает запросы клиентов. Каждый клиент при запуске создает свой собственный канал, полное имя которого определяется его идентификатором процесса. Сообщение посылаемое клиентом серверу состоит из идентификатора процесса клиента, символа пробела и имени файла, содержимое которого клиент хочет получить от файлового сервера.

В листинге 7 приведен пример программы сервера.

**Листинг 7. (*fifocliserv/mainserver.c*) Сервер для нескольких клиентов**

#include "fifo.h"

void server(int, int);

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

{

int readfifo, writefifo, dummyfd, fd;

char \*ptr, buff[MAXLINE], fifoname[MAXLINE];

pid\_t pid;

ssize\_t n;

/\* создание FIFO сервера с известным именем, ОК, если уже существует \*/

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

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

/\* открытие FIFO-сервера на чтение \*/

readfifo = open(SERV\_FIFO, O\_RDONLY, 0);

dummyfd = open(SERV\_FIFO, O\_WRONLY, 0); /\* не используется \*/

while ( (n = Readline(readfifo, buff, MAXLINE)) > 0) {

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

n--; /\* удаление newline после readline() \*/

buff[n] = '\0'; /\* полное имя, завершаемое 0 \*/

if ( (ptr = strchr(buff, ' ')) == NULL) {

err\_msg("bogus request: %s", buff);

continue;

}

\*ptr++ = 0; /\* идентификатор процесса, указатель на имя файла \*/

pid = atol(buff);

snprintf(fifoname,sizeof(fifoname),"/tmp/fifo.%ld",(long)pid);

if ( (writefifo = open(fifoname, O\_WRONLY, 0)) < 0) {

err\_msg("cannot open: %s", fifoname);

continue;

}

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

/\* ошибка, нужно сообщить клиенту \*/

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

strerror(errno));

n = strlen(ptr);

write(writefifo, ptr, n);

close(writefifo);

} else {

/\* успешное открытие, копируем файл \*/

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

write(writefifo, buff, n);

close(fd);

close(writefifo);

}

}

}

Дескриптор readfifo используется сервером для приема запросов от клиентов через канал FIFO, а дескриптор dummyfd, открытый для записи в канал, не предназначен для передачи данных. Он решает одну из проблем, связанных с блокировкой канала. Если канал не открыть на запись, то при завершении работы очередного клиента (при выполнении клиентом функции close()) этот канал будет опустошаться и сервер будет получать от функции read() код возврата 0, означающий конец файла. И в этом случае пришлось бы каждый раз закрывать канал вызовом close(), а затем заново открывать его с флагом O\_RDONLY, что приводило бы к блокированию демона до подключения следующего клиента. При использовании dummyfd мы всегда будем иметь дескриптор, открытый на запись, поэтому функция read() не будет сигнализировать о конце файла при отсутствии клиентов. Вместо этого сервер просто будет блокироваться при вызове read(), ожидая подключения следующего клиента. Этот трюк упрощает код программы-сервера и уменьшает количество вызовов open() для канала сервера.

При анализе запроса клиента функция strchr() возвращает указатель на пробел в запросе, после которого начинается имя файла.

Текст программы-клиента приведен в листинге 8.

**Листинг 8. (*fifocliserv/mainclient.c*) Клиент, подключаемый к серверу через канал FIFO**

#include "fifo.h"

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

{

int readfifo, writefifo;

size\_t len;

ssize\_t n;

char \*ptr, fifoname[MAXLINE], buff[MAXLINE];

pid\_t pid;

/\* создание FIFO с включением в его имя PID \*/

pid = getpid();

snprintf(fifoname, sizeof(fifoname), "/tmp/fifo.%ld", (long) pid);

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

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

/\* инициализация буфера PID и пробелом \*/

snprintf(buff, sizeof(buff), "%ld ", (long) pid);

len = strlen(buff);

ptr = buff + len;

/\* считывание полного имени \*/

Fgets(ptr, MAXLINE - len, stdin);

len = strlen(buff); /\* fgets() гарантирует завершающий 0 \*/

/\* открытие FIFO сервера и запись в него полного имени и PID \*/

writefifo = Open(SERV\_FIFO, O\_WRONLY, 0);

Write(writefifo, buff, len);

/\* открытие созданного FIFO; блокирование до открытия его сервером \*/

readfifo = Open(fifoname, O\_RDONLY, 0);

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

while ( (n = Read(readfifo, buff, MAXLINE)) > 0)

Write(STDOUT\_FILENO, buff, n);

Close(readfifo);

Unlink(fifoname);

exit(0);

}

Отметим, что сервер не удаляет созданный им канал клиента; клиент должен самостоятельно позаботиться об этом после приема от сервера символа конца файла.

Для проверки работы клиент-серверного приложения сервер может быть запущен в одном терминальном окне, а клиент – в другом. Однако мы можем также связаться с сервером из интерпретатора команд, поскольку каналы FIFO обладают именами в файловой системе. Пример такого сценария:

Pid=$$

mkfifo /tmp/fifo.$Pid

echo "$Pid /etc/inet/ntp.conf" > /tmp/fifo.serv

cat < /tmp/fifo.$Pid

rm /tmp/fifo.$Pid

*2.5.Ограничения программных каналов и FIFO*

На программные каналы и каналы FIFO системой накладываются всего два ограничения:

* OPEN\_MAX – максимальное количество дескрипторов, которые могут быть одновременно открыты некоторым процессом (Posix устанавливает для этой величины ограничение снизу - 16);
* PIPE\_BUF – максимальное количество данных, для которого гарантируется атомарность операции записи (Posix требует по меньшей мере 512 байт).

Значение OPEN\_MAX можно узнать, вызвав функцию sysconf(). Обычно его можно изменить из интерпретатора команд с помощью ulimit. Оно может быть изменено и самим процессом с помощью вызова функции setrlimit().

Значение PIPE\_BUF обычно определено в заголовочном файле <limits.h>, но с точки зрения стандарта Posix оно представляет собой переменную, зависимую от полного имени файла (для FIFO файлов). Это связано с тем, что разные имена файлов могут относиться к разным файловым системам и эти файловые системы могут иметь различные характеристики. Поэтому значение PIPE\_BUF можно получить в момент выполнения программы, вызвав pathconf() либо fpathconf(). В листинге 9 приведена программа, выводящая текущие значения этих двух ограничений.

**Листинг 9. (*pipe/pipeconf.c*) Определение значений PIPE\_BUF и OPEN\_MAX**

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

{ if (argc != 2) err\_quit("usage: pipeconf <pathname>");

printf("PIPE\_BUF = %ld, OPEN\_MAX = %ld\n",

pathconf(argv[1], \_PC\_PIPE\_BUF), sysconf(\_SC\_OPEN\_MAX));

exit(0);

}

Имеется также команда getconf, определенная стандартом Posix.2 и выводящая значения большинства ограничений системы. Например:

getconf OPEN\_MAX

getconf PIPE\_BUF /home/ovp

**3. ОБОРУДОВАНИЕ**

Персональный компьютер типа IBM PC Pentium, операционная система Linux, файл МУ\_ЛР\_unixSystems.doc, содержащий сборник методических указаний к лабораторным работам, каталог lab7, подкаталог pipe c файлами mainfifo.c, client.c, server.c, server\_main.c, fifo.h, client\_main.c, pipeconf.c и подкаталог fifocliserv с файлами mainserver.c и mainclient.c.

**4. ЗАДАНИЕ НА РАБОТУ**

4.1. Ознакомиться с различными способами создания и использования именованных каналов в обычных и клиент-серверных приложениях.

4.2. Перекомпилировать и выполнить тестирование обычных и клиент-серверных программ на языке Си, использующих именованные каналы.

**5. ПОРЯДОК ВЫПОЛНЕНИЯ РАБОТЫ**

5.1. Загрузить операционную систему Linux и после появления приглашения login войти в систему.

5.2. Запустить эмулятор консольного окна. Перейти в каталог lab7 и ознакомиться с содержимым находящихся в подкаталогах pipe и fifocliserv файлов mainfifo.c, client.c, server.c, server\_main.c, fifo.h, client\_main.c, pipeconf.c, mainserver.c и mainclient.c, содержащих программы, написанные на языке Cи.

5.4. Посредством команд сс или gcc откомпилировать и выполнить эти программы. На различных примерах проверить правильность работы программ.

**6. ОФОРМЛЕНИЕ ОТЧЕТА**

Отчет должен содержать:

* тексты основных программ;
* примеры тестирования программ.

**7. КОНТРОЛЬНЫЕ ВОПРОСЫ**

7.1. Какие объекты IPC используются в различных системах UNIX?

7.2. Как можно создать и проверить FIFO канал в командной среде?

7.3. Можно ли использовать функцию lseek() для FIFO-файла?

7.4. Как удаляется FIFO-файл из файловой системы?

7.5. Какое назначение файлового сервера?

7.6. Приведите прототипы функций sprintf() и snprintf() и укажите их назначение.

7.7. Как влияет наличие или отсутствие флага O\_NONBLOCK на открытие и чтение данных из пустого программного канала или FIFO?

7.8. Как можно включить флаг O\_NONBLOCK у канала?

7.9. Какими способами можно создать именованный канал с утилитой или между двумя утилитами?

7.10. Какие ограничения накладываются системой на программные каналы и каналы FIFO?

**БИБЛИОГРАФИЧЕСКИЙ СПИСОК**

1. Стивенс У. UNIX: взаимодействие процессов. – СПб. : Питер, 2002. – 576 с.

2. Митчелл М., Оулдем Д., Самьюэл А. Программирование для Linux. Профессиональный подход. - М.: "Вильямс", 2002. - 288 с.

3. Робачевский А.М. Операционная система UNIX. - СПб.: БХВ - Санкт-Петербург, 2005. - 528 с.

4. Дансмур М., Дейвис Г. Операционная система UNIX и программирование на языке Си. – М.: "Радио и связь", 1989. – 192 с.

5. Богатырев А. Хрестоматия по программированию на Си в UNIX. Электронное издание, www.opennet.ru. – 1996 г.

6. Б.Керниган, Д.Ритчи, А.Фьюер. Язык программирования Си. Задачи по языку Си. - М.: Финансы и статистика, 1985.

Лабораторная работа № 8

Создание и синхронизация потоков в системе Linux

**1. ЦЕЛЬ И ЗАДАЧИ РАБОТЫ**

Ознакомление с потоками системы Linux, предназначенными для параллельного выполнения программного кода процесса, а также получение практических навыков создания многопотоковых программ и синхронизации потоков.

**2. ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ**

*2.1. 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:

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

**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.

*2.2. 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.

*2.3. 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.

*2.4. 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;

}

*2.5. 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.

**3. ОБОРУДОВАНИЕ**

Персональный компьютер типа IBM PC Pentium, операционная система Linux, файл МУ\_ЛР\_unixSystems.doc, содержащий сборник методических указаний к лабораторным работам, каталог lab8 c файлами thread-create.c, thread-create2.c, primes.c, critical-section.c, tsd.c, job-queue1.c, job-queue2.c, job-queue3.c.

**4. ЗАДАНИЕ НА РАБОТУ**

4.1. Ознакомиться с различными способами создания и использования программных потоков.

4.2. Перекомпилировать и выполнить тестирование программ на языке Си, использующих программные потоки.

**5. ПОРЯДОК ВЫПОЛНЕНИЯ РАБОТЫ**

5.1. Загрузить операционную систему Linux и после появления приглашения login войти в систему.

5.2. Запустить эмулятор консольного окна. Перейти в каталог lab8 и ознакомиться с содержимым находящихся в нем файлов thread-create.c, thread-create2.c, primes.c, critical-section.c, tsd.c, job-queue1.c, job-queue2.c, job-queue3.c, содержащих программы, написанные на языке Cи.

5.4. Посредством команд сс или gcc откомпилировать и выполнить эти программы. На различных примерах проверить правильность работы программ.

**6. ОФОРМЛЕНИЕ ОТЧЕТА**

Отчет должен содержать:

* тексты основных программ;
* примеры тестирования программ.

**7. КОНТРОЛЬНЫЕ ВОПРОСЫ**

7.1. Какой стандарт работы с потоками реализован в Linux?

7.2. Какой тип имеет потоковая функция?

7.3. Приведите команду компиляции многопотоковой программы.

7.4. Напишите программу реализации канала (именованного или неименованного) между двумя дополнительными потоками программы или между основным и дополнительным потоком.

7.5. Какие атрибуты имеются у потоков?

7.6. Как происходит возврат значений потоковыми функциями?

7.7. В каких случаях необходима синхронизация потоков?

7.8. Когда необходим исключающий и когда – обычный семафор?

7.9. Приведите программу, использующую функцию sem\_getvalue()?

7.10. Чем отличаются семафоры по стандартам POSIX и System V?

**БИБЛИОГРАФИЧЕСКИЙ СПИСОК**

1. Митчелл М., Оулдем Д., Самьюэл А. Программирование для Linux. Профессиональный подход. - М.: "Вильямс", 2002. - 288 с.

2. Стивенс У. UNIX: взаимодействие процессов. – СПб. : Питер, 2002. – 576 с.

3. Робачевский А.М. Операционная система UNIX. - СПб.: БХВ - Санкт-Петербург, 2005. - 528 с.

4. Дансмур М., Дейвис Г. Операционная система UNIX и программирование на языке Си. – М.: "Радио и связь", 1989. – 192 с.

5. Богатырев А. Хрестоматия по программированию на Си в UNIX. Электронное издание, www.opennet.ru. – 1996 г.

Лабораторная работа № 9

Семафоры стандарта Posix для синхронизации процессов и потоков в системе Linux

**1. ЦЕЛЬ И ЗАДАЧИ РАБОТЫ**

Ознакомление с семафорами системы Linux, предназначенными для синхронизации параллельного выполнения модулей приложения, а также получение практических навыков создания и использования семафоров.

**2. ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ**

*2.1. 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, FIFO, очереди сообщений);
* объекты для синхронизации (взаимные исключения, условные переменные, блокировки чтения-записи, семафоры);
* разделяемая память (неименованная и именованная).

Объекты IPC (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. Рассмотрим вначале семафоры Posix.

Для работы с семафорами Posix могут применяться следующие три операции.

1. Создание семафора. При этом необходимо указать начальное значение (value) семафора – обычно число ресурсов, выделением которых процессам и потокам будет управлять семафор. Если ресурс один, то семафор называется бинарным, а в качестве начального значения указывается 1, но может быть и 0.

2. Ожидание освобождения ресурса (wait). При этом производится проверка значения семафора и процесс блокируется, если значение оказывается меньшим или равным 0. Если же в момент выдачи запроса wait семафор открыт (значение семафора 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 изменения значения семафора до величины, превосходящей 0, 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.

**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 вместо взаимных исключений. В листинге 2 приведен фрагмент программы на псевдокоде с двумя бинарными семафорами для решения упрощенной версии задачи потребитель – производитель.

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.

**Листинг 2.** Псевдокод для задачи потребитель – производитель.

|  |  |
| --- | --- |
| **Производитель (producer)**  Инициализация семафора get значением 0;  Инициализация семафора put значением 1;  for ( ; ; ) {  sem\_wait (&put);  sem\_post(&get);  } | **Потребитель (consumer)**  for ( ; ; ) {  sem\_wait (&get);  sem\_post(&put);  } |

*2.2. 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 соответствовать существующим требованиям к именам файлов (не превышать в длину PATH\_MAX байтов, включая завершающий символ с кодом 0). 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 \*/

*2.3. 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.

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).

**Листинг 3. (*semcreate.c*) Создание именованного семафора**

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

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);

}

Отметим, что при компиляции программ с семафорами Posix необходимо подключать библиотеку для работы с потоками с помощью опции - lpthread.

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 semunlink.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 6. (*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);

}

Выполним теперь тестирование созданного командного интерфейса для именованных семафоров Posix. Для начала мы создадим именованный семафор и выведем его значение, полученное им при инициализации

semcreate /tmp/test1

ls -l /tmp/test1

semgetvalue /tmp/test1

Далее закроем семафор и прервем с помощью <ctrl/c> работу программы, установившей блокировку:

semwait /tmp/test1

<ctrl/c>

semgetvalue /tmp/test1

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.

Следующий пример показывает, что отрицательное значение семафора используется для хранения информации о количестве процессов, ожидающих разблокирования семафора:

semgetvalue /tmp/test1

semwait /tmp/test1 &

semgetvalue /tmp/test1

semwait /tmp/test1 &

semgetvalue /tmp/test1

sempost /tmp/test1

sempost /tmp/test1

**3. ОБОРУДОВАНИЕ**

Персональный компьютер типа IBM PC Pentium, операционная система Linux, файл МУ\_ЛР\_unixSystems.doc, содержащий сборник методических указаний к лабораторным работам, каталог lab9, подкаталог pxsem с файлами semcreate.c, semunlink.c, semgetvalue.c, semwait.c, sempost.c и подкаталог svsem с файлами semcreate.c, semrmid.c, semsetvalue.c, semgetvalue.c, semops.c.

**4. ЗАДАНИЕ НА РАБОТУ**

4.1. Ознакомиться с семафорами Posix, а также с программами, реализующими командный интерфейс семафоров.

4.2. Перекомпилировать и выполнить тестирование программ на языке Си, реализующих командный интерфейс семафоров Posix.

**5. ПОРЯДОК ВЫПОЛНЕНИЯ РАБОТЫ**

5.1. Загрузить операционную систему Linux и после появления приглашения login войти в систему.

5.2. Запустить эмулятор консольного окна. Перейти в каталог lab9/pxsem и ознакомиться с содержимым находящихся в нем программ semcreate.c, semunlink.c, semgetvalue.c, semwait.c, sempost.c, реализующих командный интерфейс семафоров Posix. Перекомпилировать исходные файлы и создать одноименные исполняемые файлы. Выполнить тестирование работы семафоров Posix из командной строки.

**6. ОФОРМЛЕНИЕ ОТЧЕТА**

Отчет должен содержать:

* тексты основных программ;
* примеры тестирования программ.

**7. КОНТРОЛЬНЫЕ ВОПРОСЫ**

7.1. Какие операции выполняют семафоры Posix?

7.2. Чем отличается бинарный семафор от взаимного исключения?

7.3. Приведите пример программы, реализующей механизм взаимодействия процессов типа производитель-потребитель.

7.4. У каких семафоров создание и инициализация семафора реализуется посредством атомарной операции?

7.5. Как используется в программах библиотечная функция getopt()?

7.6. Как доказать, что именованный семафор Posix обладает живучестью ядра? Какой живучестью обладает канал FIFO?

**БИБЛИОГРАФИЧЕСКИЙ СПИСОК**

1. Робачевский, А.М. Операционная система UNIX : учеб.пособие / А.М.Робачевский, С.А.Немнюгин, О.Л.Стесик .— 2-е изд.,перераб.и доп. — СПб. : БХВ-Петербург, 2005 .— 635с.

2. Магда, Ю.С. UNIX для студента / Ю.С.Магда .— СПб. : БХВ-Петербург, 2007 .— 480с.

3. Рочкинд, M.Д. Программирование для UNIX / М.Д.Рочкинд;пер.с англ.под общ.ред.В.В.Вшивцева .— 2-е изд. — СПб. : БХВ-Петербург, 2005 .— 704с.

4. Моли, Molay B. Unix/Linux:теория и практика программирования : пер.с англ. / Б.Моли .— М. : КУДИЦ-ОБРАЗ, 2004 .— 576с.

5. Галатенко, В.А. Программирование в стандарте POSIX : Курс лекций:Учеб.пособие / В.А.Галатенко; Под ред.В.Б.Бетелина .— М. : Интернет-Университет Информационных Технологий, 2004 .— 560с.

6. Хорвиц, Д. Unix-системы.От проектирования до сопровождения : пер.с англ. / Д.Хорвиц .— М. : DiaSoft, 2004 .— 608c.

7. Реймонд, Э.С. Искусство программирования для Unix / Э.С.Реймонд;пер.с англ.и ред.В.А.Швеца .— М.и др. : Вильямс, 2005 .— 543с.

8. Стивенс У. UNIX: взаимодействие процессов. – СПб. : Питер, 2002. – 576 с.

9. Митчелл М., Оулдем Д., Самьюэл А. Программирование для Linux. Профессиональный подход. - М.: "Вильямс", 2002. - 288 с.