

Министерство образования и науки Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего профессионального образования
**Пермский национальный исследовательский
политехнический университет**

Кузнецов Д. Б., Сорока Д. П.

Исследование операционной системы minix

**Методические указания к выполнению лабораторных работ по дисциплине
"Операционные системы" для студентов специальностей РИС, АСУ и ЭВТ**

Пермь, 2016

Утверждено на заседании кафедры ИТАС _____ г.
Протокол №____/ ПГТУ. – Пермь: 2016.

Введение

Предлагаемый лабораторный практикум предназначен для освоения основных алгоритмов функционирования операционных системы (ОС).

Требования к аппаратному обеспечению

Вариант 1: Выполнение работы на локальном компьютере

Компьютер должен иметь процессор с частотой от 1000 МГц, ОЗУ от 512 МБ, свободно дисковое пространство от 20 ГБ. Рекомендуется использование процессоров с аппаратной виртуализацией.

Вариант 2: Выполнение работы на удаленном сервере

Сервер должен иметь процессор с частотой от 300 МГц, ОЗУ от 512 МБ, свободно дисковое пространство от 400 МБ.

Локальное рабочее место должно быть оборудовано терминалом, подключенным к серверу. Параметры сервера кафедры ИТАС указаны в Приложении 3.

Требования к системному программному обеспечению

Рассматриваемое ниже системное программное обеспечение должно быть установлено на локальном компьютере (Вариант 1 аппаратного обеспечения) или на сервере (Вариант 2 аппаратного обеспечения).

Работы выполняются в операционной системе minix. Для запуска minix рекомендуется использовать выдаваемый отдельно образ виртуальной машины, запускаемый в среде VirtualBox. Также возможно использование других сред виртуализации (kvm, vmware) или установка на физическую аппаратуру. В процессе работы для доступа к файлам и программам minix можно использовать системы терминального доступа (ssh, putty) и файловые менеджеры (winscp, filezilla, nautilus). Системное программное обеспечение, используемое в рамках minix предустановлено в рамках выдаваемого образа виртуальной машины. Для справки. Кроме базовой системы minix 3.3.0. Установлены исходные коды системы (/usr/src/) и пакеты: bash, binutils, clang, curl, nano, openssh...

Требования к прикладному программному обеспечению

Для удобства программирования на Си, при работе через ssh, можно использовать текстовые редакторы и интегрированные системы: notepad++, gedit, eclipse.

Требования к методическому обеспечению

Для выполнения работ требуется наличие данного методического материала и конспекта лекций по дисциплине “Операционные системы”.

Лабораторная работа №1 Знакомство с minix

Цель работы

Изучение основ Minix. Знакомство с процессом установки. Подготовка к работе.

Краткие теоретические сведения

Установка системы

Первым делом для установки ОС Minix необходимо скачать образ ее дистрибутива с официального сайта: www.minix3.org. В данном методическом пособии рассматривается работа с версией 3.3.0.

С образом дистрибутива можно распорядиться двумя способами. Либо с помощью специальных утилит записать образ на диск CD-R(W), после чего установить систему непосредственно на ПК, либо использовать образ для установки системы на виртуальную машину, создаваемой с помощью специальных программ. В данном пособии будет использоваться второй способ, для создания виртуальной машины будет использоваться VirtualBox.

Первым делом необходимо скачать дистрибутив VirtualBox с сайта <https://www.virtualbox.org/>, затем произвести его установку, следуя указаниям мастера.

По окончании установки необходимо запустить VirtualBox, если не произошло его автоматического запуска.

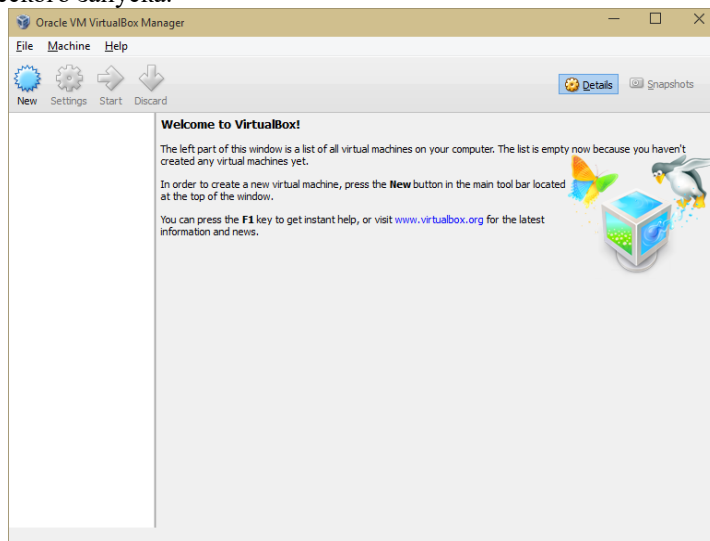


Рисунок 1. Главное окно VirtualBox

Создание новой виртуальной машины осуществляется с помощью нажатия на кнопку “New”, или команды меню Machine => Add. После выполнения одного из этих действий будет запущен мастер создания новой виртуальной машины. На первом шаге необходимо задать название для виртуальной машины, а также выбрать тип операционной системы, которая будет на нее установлена. При установке системы Minix в качестве типа системы и её версии необходимо выбрать значения *Other* и *Other/Unknown* соответственно. Название виртуальной машины задается в соответствии с предпочтениями пользователя.

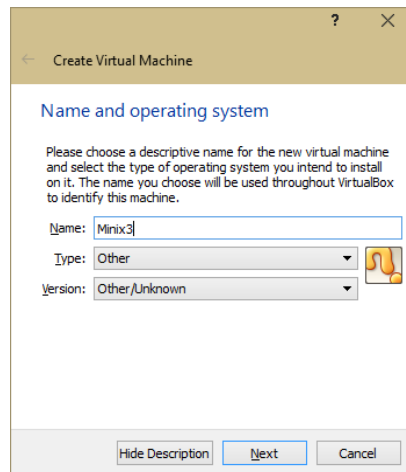


Рисунок 2 Шаг первый создания ВМ – определение названия и типа ОС

На следующем шаге необходимо выбрать количество оперативной памяти, доступной виртуальной машине. Рекомендуемое значение на этом шаге – 256 МБ.

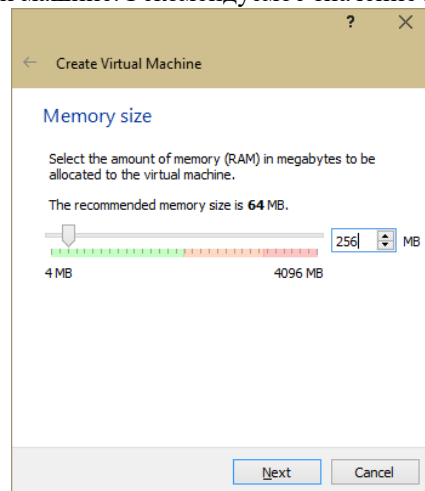


Рисунок 3 Шаг второй - определение размера RAM

Далее необходимо определить виртуальный жесткий диск, который будет использоваться виртуальной машиной. На этом шаге мастер создания ВМ предлагает 3 варианта:

- ☐ Не использовать виртуальный жесткий диск. Данный вариант используется в случае с так называемыми Live-CD – образами или непосредственно лазерными дисками, с которых загружается система, не требующая установки.
- ☐ Создать виртуальный жесткий диск. Наиболее часто используемый вариант, подразумевающий создание нового виртуального жесткого диска для дальнейшего использования.
- ☐ Использовать существующий жесткий диск. Вариант используется в том случае, если в распоряжении пользователя имеется образ жесткого диска с уже установленной (и, возможно, настроенной определенным образом) операционной системой.

В рамках установки системы Minix необходимо выбрать второй вариант.

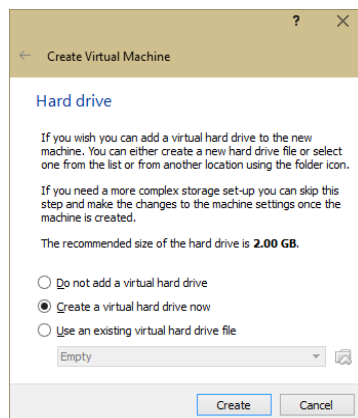


Рисунок 4 Шаг третий - выбор виртуального жесткого диска

При выборе второй опции запустится мастер создания виртуального жесткого диска. На первом этапе предлагается выбрать один из форматов виртуальных жестких дисков. Опция выбора существует для обеспечения совместимости с другими программами виртуализации. Для установки Minix будет использоваться стандартный формат VDI.

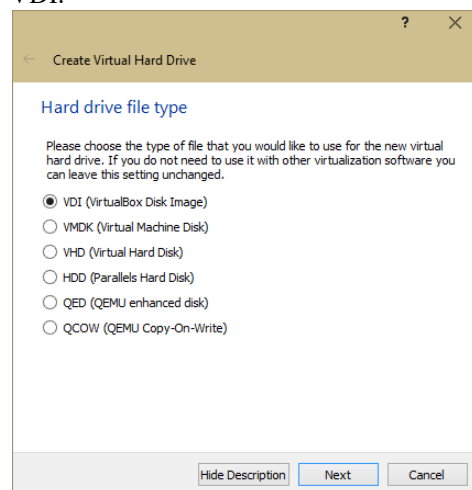


Рисунок 5 Выбор формата виртуального жесткого диска

Следующим шагом определяется, как будет выделяться место для виртуального жесткого диска. Существует два варианта:

- ☐ Динамическое определение размера. Файл, отвечающий за хранение такого жесткого диска, будет расти в размерах динамически, по мере использования виртуального жесткого диска системой. Стоит принять во внимание тот факт, что файл будет только расти (вплоть до обозначенного предела), но не уменьшаться. Преимуществом такого подхода является то, что файл в этом случае как правило занимает меньше места, поскольку место виртуального жесткого диска редко используется по максимуму. Недостатком же является то, что в случае если на реальном жестком диске закончится свободное место, виртуальному диску будет некуда расти, что может привести к ошибкам в работе.
- ☐ Фиксированный размер. Файл, отвечающий за хранение такого жесткого диска, будет сразу создан с тем размером, который будет определен для виртуального жесткого диска. Преимуществами такого

подхода как правило являются небольшой выигрыш в производительности (связанный с меньшей подверженности фрагментации) и невозможность возникновения ситуации, когда на реальном жестком диске закончилось место для растущего виртуального диска. Недостатком же является то, что зачастую не малая часть места, определенного под виртуальный жесткий диск, остается неиспользованной.

Для установки системы Minix рекомендуется использовать динамическое распределение.

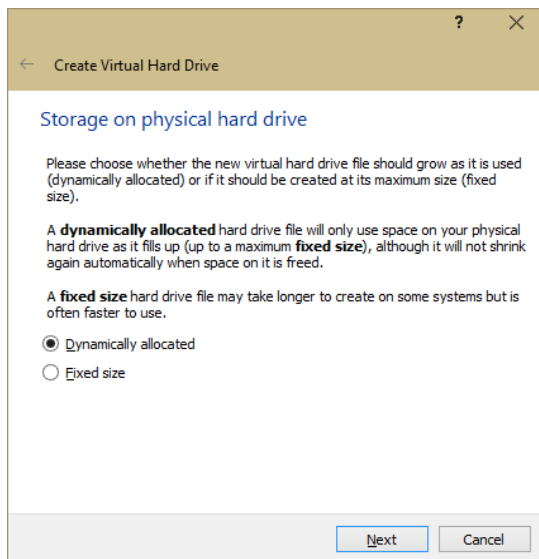


Рисунок 6 Определение типа хранения виртуального жесткого диска

Последним шагом при создании виртуального жесткого диска является определение места хранения файла виртуального диска и его размер. Рекомендуемый размер – 5 ГБ, место хранения отдается на усмотрение пользователя.

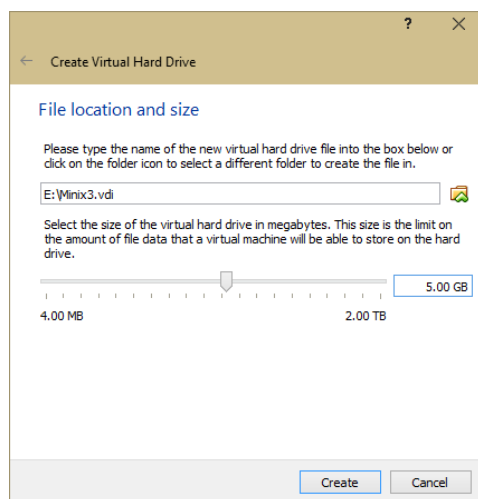


Рисунок 7 Определение места хранения файла виртуального жесткого диска и его размера

По завершению мастера создания виртуального жесткого диска виртуальная машина будет создана и отображена в главном окне программы.

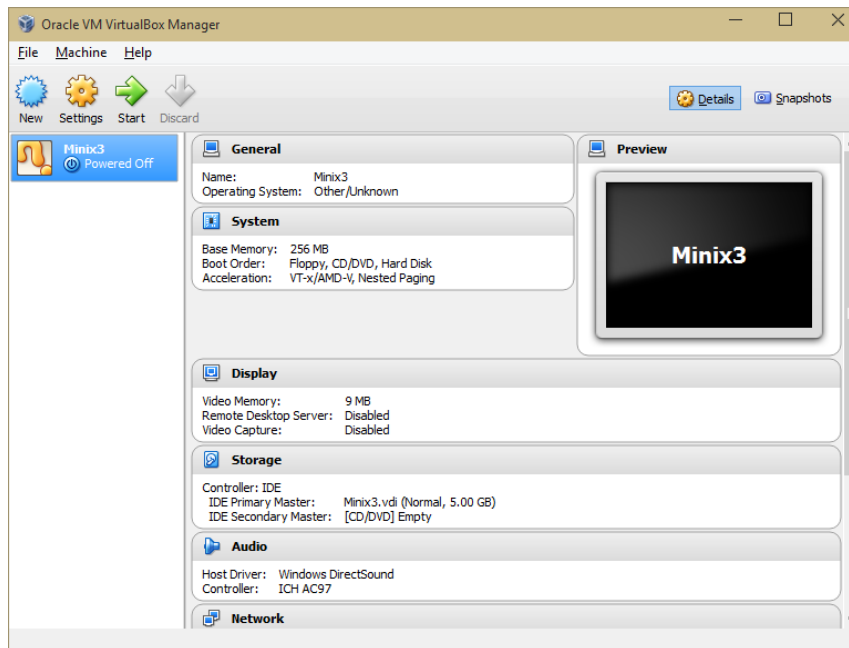


Рисунок 8 Созданная виртуальная машина в главном окне VirtualBox

После создания виртуальной машины необходимо произвести ее дальнейшую настройку, в соответствии с рекомендациями, приведенными на странице документации Minix:

<http://wiki.minix3.org/doku.php?id=usersguide:runningonvirtualbox>. Для этого:

1. С помощью кнопки “Settings”, команды Machine => Settings, сочетания клавиш Ctrl+S, или выпадающего меню открыть окно настроек виртуальной машины.
2. На вкладке System => Motherboard выбрать опцию Hardware Clock in UTC Time
3. На вкладке System => Acceleration выбрать опцию Enable VT-x/AMD-V

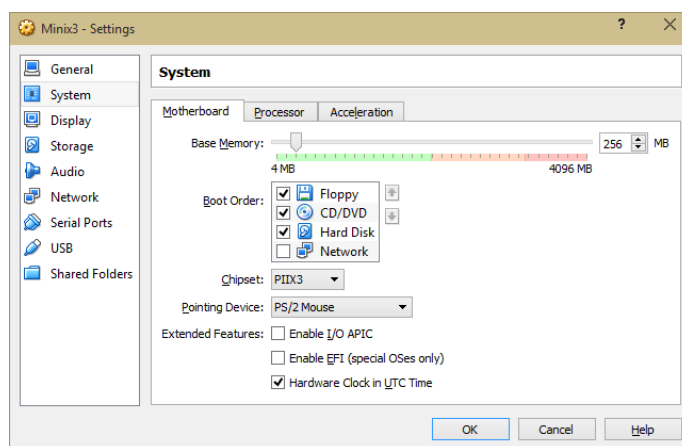


Рисунок 9 Опция Hardware Clock in UTC Time

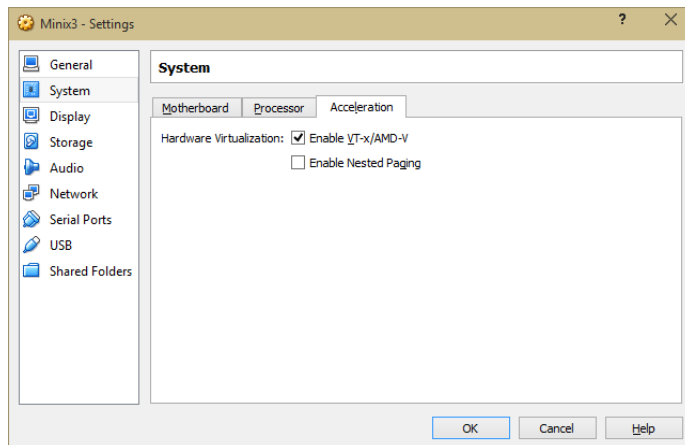


Рисунок 10 Опция Enable VT-x/AMD-V

После применения настроек можно запустить виртуальную машину. При первом запуске VM VirtualBox покажет диалоговое окно, в котором пользователю предлагается выбрать оптический привод или образ диска, с которого необходимо произвести загрузку. В этом диалоговом окне необходимо выбрать образ дистрибутива системы, скачанный с официального сайта Minix.

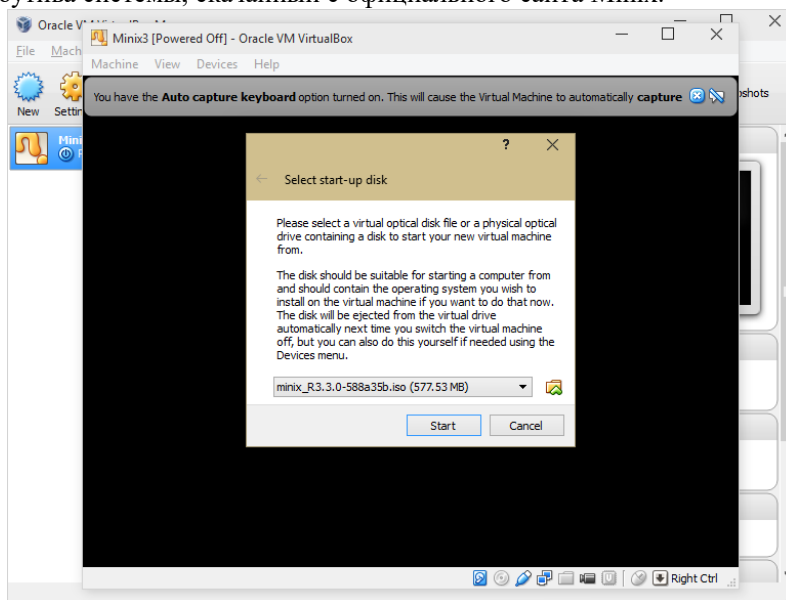


Рисунок 11 Диалоговое окно выбора загрузочного диска

После выбора диска виртуальная машина включится, и загрузится с диска, указанного при запуске, а спустя 10 секунд (или по нажатию цифры 1) система Minix запустится с дистрибутива, при этом выдав сообщение о том, что большинство команд будет работать в штатном режиме, однако для «серьезного» использования необходимо провести установку. Для авторизации используется логин “root”, для запуска установки – команда “setup”.

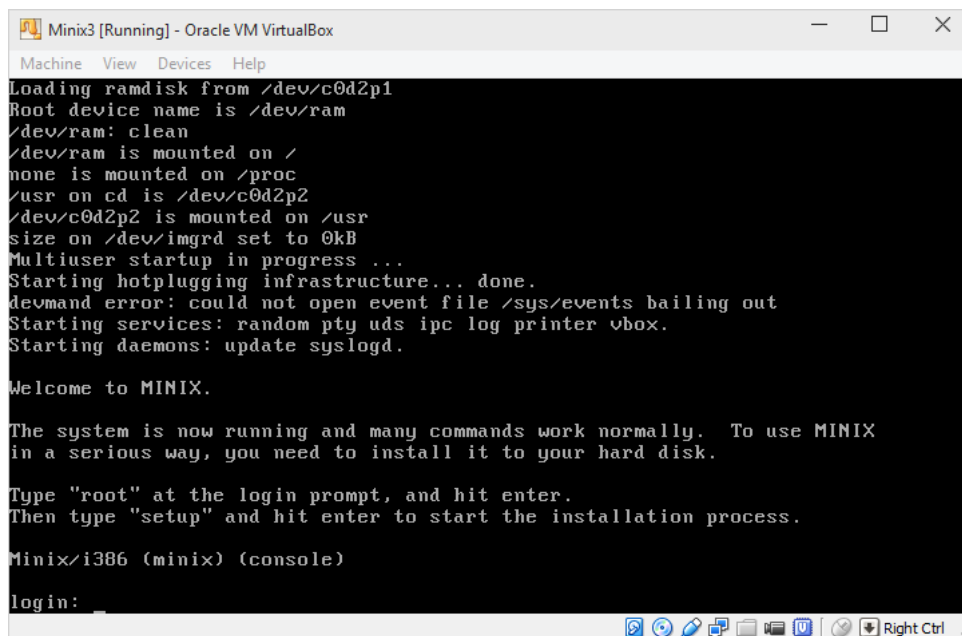


Рисунок 12 Окно с запущенной системой Minix3 - приглашение для входа по логину

При запуске установки будет показана краткая инструкция по использованию установщика, состоящая из 4 пунктов.

1. Если экран опустеет, необходимо нажать Ctrl+F3.
2. Если вы ошиблись при выборе вариантом, нажмите Ctrl+C для того чтобы начать заново.
3. Ответы по умолчанию, например, [y], могут быть выбраны простым нажатием клавиши Enter.
4. Если вы видите двоеточие (:), значит необходимо нажать клавишу Enter для продолжения.

Сразу после 4 пункта стоит двоеточие, а значит, для перехода к следующему шагу необходимо нажать Enter.

Далее по шагам, используемым мастером:

1. Выбор типа клавиатуры – Russian
2. Автоматический выбор без участия пользователя
3. Создание раздела жесткого диска. Сперва предполагается выбрать между автоматическим режимом и экспертным – выбирается автоматический (простым нажатием на клавишу Enter). Затем выбирается устройство, на которое будет произведена установка, затем – раздел устройства, после чего необходимо ввести в командную строку “yes” для подтверждения намерений установки в указанный раздел.
4. Шаг используется при переустановке системы поверх старой, соответственно, при чистой установке пропускается.
5. Необходимо указать в мегабайтах размер домашней директории. Размер директории по умолчанию зависит от размера раздела, на который будет производиться установка

6. Необходимо выбрать размер блока в килобайтах. Рекомендуется использовать значение по умолчанию.
7. Начнется копирование файлов системы на жесткий диск. Общий прогресс будет демонстрироваться специальным индикатором, а также количеством файлов, ожидающих копирования.
8. Необходимо выбрать Ethernet чип, используемый компьютером. В случае с виртуальной машиной, эмулируется чип AMD Lance, поэтому выбираем пункт 9 (выбран по умолчанию)
9. На этом шаге определяется, конфигурация сети: автоматически с помощью DHCP или вручную. Выбирается первый вариант – автоматически.

По завершению установки система предложит выполнить команду “reboot” для перезагрузки системы, однако до следующего запуска системы необходимо отключить образ дистрибутива от VM, в связи с чем используется команда “shutdown now” для выключения системы. Стоит принять во внимание тот факт, что команда не выключит саму VM, а лишь подготовит систему к отключению, поэтому саму VM необходимо будет выключить, используя средства VirtualBox (Machine => Close => Power Off).

После выключения VM необходимо зайти в ее настройки, вкладку Storage, выбрать образ дистрибутива и отключить его.

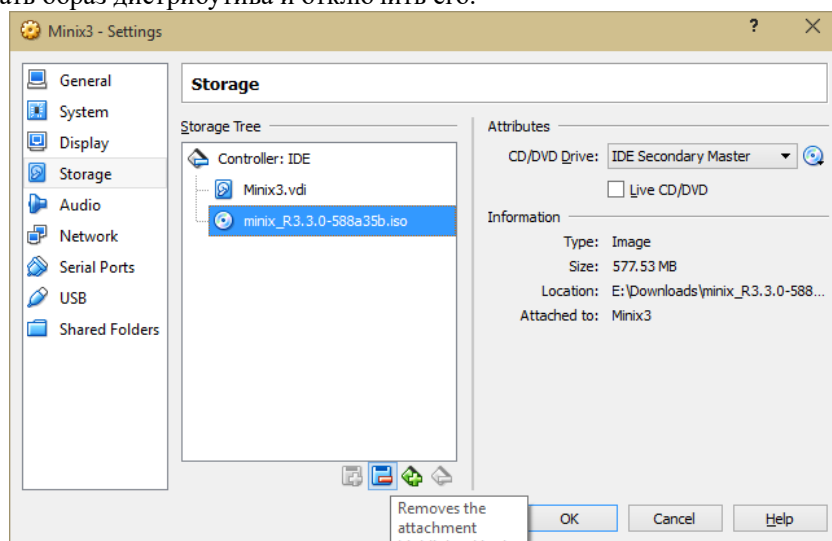


Рисунок 13 Образ дистрибутива в настройках VM

После этого можно вновь запустить виртуальную машину. Система Minix будет запущена автоматически.

□ **Настройка сети**

Приведенные далее настройки скорее всего будут работать только в том случае, если у вас есть роутер, через который осуществляется выход в сеть Internet, а также – DHCP сервер, запущенный на этом роутере. В противном случае работоспособность описанного метода не гарантируется – экспериментируйте. Для того, чтобы продолжить работу с системой, необходимо настроить сеть, дабы система могла скачивать из сети интернет обновления и новые программы. Для этого:

Выключите виртуальную машину.

Откройте окно настроек виртуальной машины.

Выберите вкладку Network (Сеть)

В выпадающем списке «Attached to» выберите Bridged Adapter

Нажмите «ОК»

Теперь виртуальную машину можно включить. После загрузки системы, IP адрес, выделенный для системы, будет указан перед вводом логина и пароля.

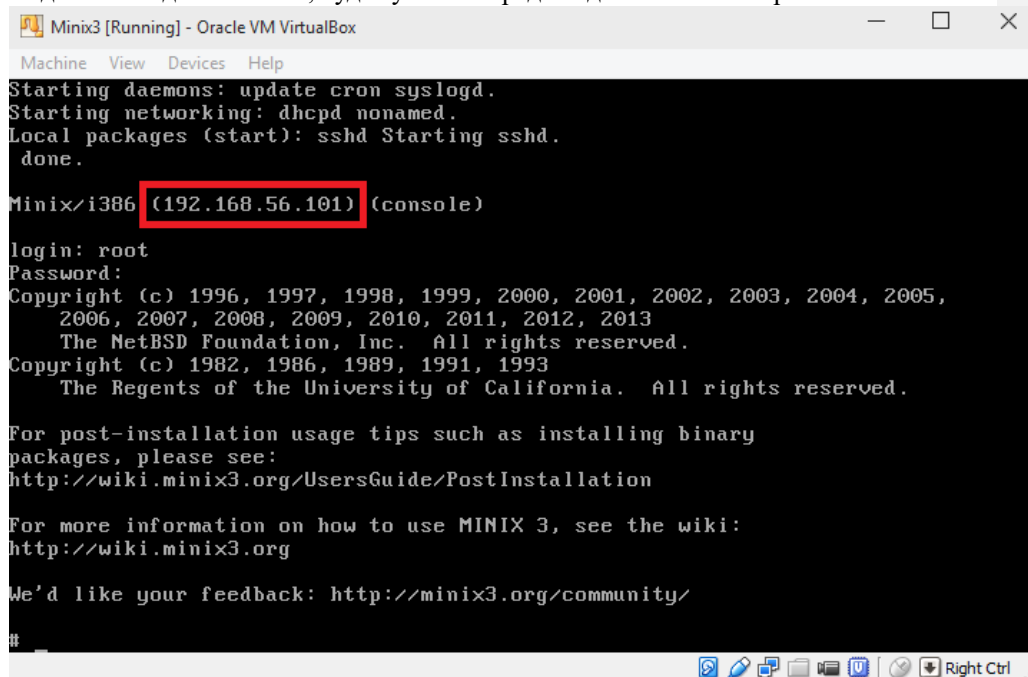


Рисунок 14 IP адрес системы - 192.168.56.101

□ Создание пользователя для работы с системой

Первым делом, после установки системы необходимо изменить пароль, используемый при входе в систему, а еще лучше – создать нового пользователя, используя которого вы будете работать с системой.

Для того, чтобы изменить пароль root, введите команду *passwd* после чего нужно будет ввести пароль для доступа к аккаунту (один раз чтобы задать его, второй – чтобы подтвердить). Обратите внимание, Minix пытается заботиться о вашей безопасности, в связи с чем может начать ругаться на слишком короткие и простые пароли. Возможно, вам придется вводить пароль более двух раз. После того, как вы создадите пароль для root, стоит создать пользователя, под которым вы будете работать. Для этого введите команду *useradd -m _логин_*, где *useradd* – команда для создания пользователя, *-m* дает команду системе автоматически создать домашний каталог для пользователя, а *_логин_* - имя пользователя, которое будет использоваться при входе в систему. Команда для создания пользователя *student* будет выглядеть следующим образом:

```
useradd -m student
```

После того, как пользователь будет создан, он будет заблокирован до тех пор, пока не будет установлен пароль для этого пользователя. Для этого используется

команда *passwd _логин_*. Для создания пароля для пользователя *student* команда будет выглядеть следующим образом:

```
passwd student
```

После того, как пароль будет создан, можно будет войти в систему используя созданного пользователя. Это можно сделать прямо из пользователя *root*, введя команду *su _логин_*. Например, для пользователя *student*:

```
su student
```

Обратите внимание, при входе в систему как *student* из пользователя *root* система не станет спрашивать пароль от пользователя *student*. Это связано с тем, что в системах вроде *Minix* пользователь *root* подобен богу, и может творить что захочет – вот почему так важно защищать пользователя *root* хорошим паролем, а также избегать повседневного использования этого пользователя – мало ли кто сядет за компьютер пока вы отойдете.

Другим способом зайти под *student* является сперва выйти из пользователя *root* командой *exit*, после чего система сама предложит ввести новое имя пользователя и его пароль.

Обратите внимание, как меняется спецсимвол, обозначающий ввод команд от имени пользователя *root* (#) и любого другого пользователя (\$).

Для того, чтобы удалить пользователя, необходимо войти в систему как пользователь *root* и использовать команду *userdel _логин_*.

Подробнее: <http://wiki.minix3.org/doku.php?id=usersguide:managinguseraccounts>

□ Установка необходимых пакетов

В системе *Minix* для установки и удаления программ, а также для обновления системы используется команда *pkgin*. После установки системы рекомендуется обновить систему, для этого используется команда *pkgin upgrade*. Помимо этого, стоит обновить и те приложения, которые поставляются «из коробки» командой *pkgin update*.

Затем, необходимо установить некоторые программы, которые понадобятся в будущем: **bash**, **binutils**, **clang**, **curl**, **nano**, **openssh**. Программы устанавливаются с помощью команды *pkgin install _программа_*.

Например, для того, чтобы установить перечисленные выше программы, необходимо ввести команду:

```
pkgin install bash binutils clang curl nano openssh
```

Система после этого просканирует базу данных приложений, определив необходимые для установки пакеты, и выдаст запрос на подтверждение действий, указав при этом, сколько данных нужно скачать и сколько места займут установленные пакеты.

После установки указанных пакетов необходимо перезапустить систему командой *reboot*.

Подробнее: <http://wiki.minix3.org/doku.php?id=usersguide:installingbinarypackages>

□ Подключение к терминалу через ssh

После установки пакета *openssh* к системе можно будет подключаться удаленно, используя программы, работающие через ssh протокол (*Putty* для использования удаленного терминала, *WinSCP* для передачи файлов). Рассмотрим подключение к системе на примере использования программы *Putty*.

Для подключения к системе вам необходимо знать IP-адрес, к которому необходимо подключаться, а также порт, который используется протоколом (по умолчанию – 22). При работе внутри локальной сети используется IP адрес,

показываемый при запуске системы (перед запросом авторизации). Этот IP адрес вводится в окне Putty, после чего кнопкой **Open** открывается соединение.

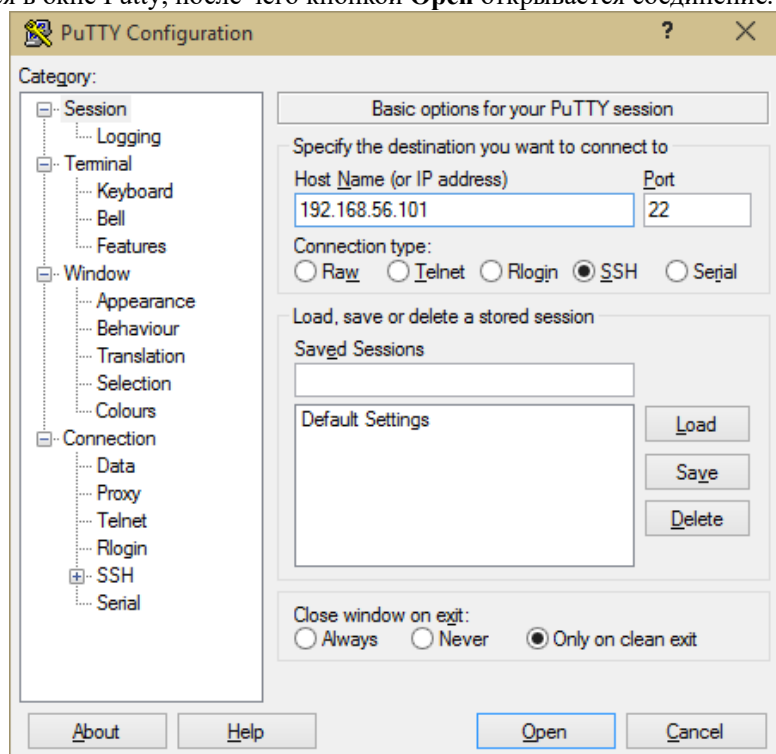
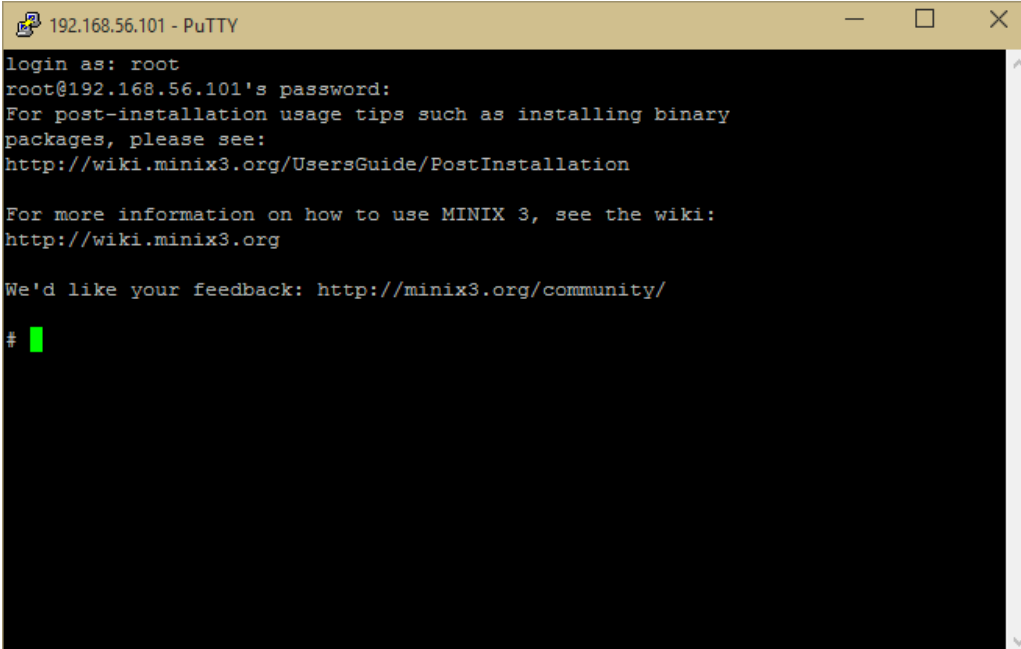


Рисунок 15 Окно соединения Putty

При первом соединении Putty выдаст предупреждающее сообщение (смысл которого – вдруг вы подключаетесь не туда куда планировали?), на которое нужно ответить утвердительно. После этого откроется окно терминала, с предложением ввести логин и пароль для авторизации в системе.

A screenshot of a PuTTY terminal window. The title bar at the top reads "192.168.56.101 - PuTTY". The terminal content shows a login prompt "login as: root", followed by "root@192.168.56.101's password:". Below this, there are two informational messages: "For post-installation usage tips such as installing binary packages, please see: http://wiki.minix3.org/UsersGuide/PostInstallation" and "For more information on how to use MINIX 3, see the wiki: http://wiki.minix3.org". A third message says "We'd like your feedback: http://minix3.org/community/". At the bottom, there is a prompt character "# " followed by a green cursor block.

```
login as: root
root@192.168.56.101's password:
For post-installation usage tips such as installing binary
packages, please see:
http://wiki.minix3.org/UsersGuide/PostInstallation

For more information on how to use MINIX 3, see the wiki:
http://wiki.minix3.org

We'd like your feedback: http://minix3.org/community/

# 
```

Рисунок 16 Терминал Putty

Терминал Putty несколько удобнее в использовании, например, позволяет использовать команды «Копировать-Вставить», не блокирует мышь в отличие от VirtualBox.

Задание к работе

Выполнить установку и первичную настройку системы Minix в соответствии с теоретической частью. Отчет должен содержать скриншоты процесса установки с введенными пользователем данными и пояснениями к ним, скриншот подключения к системе через терминал Putty и выполнения в этом терминале команды *whoami*.

Лабораторная работа №2

Рекурсивная сборка программного обеспечения с использованием Makefile

Цель работы

Изучение основных навыков работы с инструментами компиляции программ, функционала *make*

Краткие теоретические сведения

Make — утилита, автоматизирующая процесс преобразования файлов из одной формы в другую. Чаще всего это компиляция исходного кода в объектные файлы и последующая компоновка в исполняемые файлы или библиотеки.

Утилита использует специальные *make*-файлы, в которых указаны зависимости файлов друг от друга и правила для их удовлетворения. На основе информации о времени последнего изменения каждого файла *make* определяет и запускает необходимые программы.

Программа *make* выполняет команды согласно правилам, указанным в специальном файле. Этот файл называется *make*-файл (*makefile*, *мейкфайл*). Как правило, *make*-файл описывает, каким образом нужно компилировать и компоновать программу.

make-файл состоит из правил и переменных. Правила имеют следующий синтаксис:

```
цель1 цель2 ...: реквизит1 реквизит2 ...
    команда1
    команда2
...
```

Правило представляет собой набор команд, выполнение которых приведёт к сборке файлов-целей из файлов-реквизита.

Правило сообщает *make*, что файлы, получаемые в результате работы команд (*цели*) являются зависимыми от соответствующих файлов-реквизита. *make* никак не проверяет и не использует содержимое файлов-реквизита, однако, указание списка файлов-реквизита требуется только для того, чтобы *make* убедилась в наличии этих файлов перед началом выполнения команд и для отслеживания зависимостей между файлами.

Обычно цель представляет собой имя файла, который генерируется в результате работы указанных команд. Целью также может служить название некоторого действия, которое будет выполнено в результате выполнения команд (например, цель *clean* в *make*-файлах для компиляции программ обычно удаляет все файлы, созданные в процессе компиляции).

Строки, в которых записаны команды, должны начинаться с символа табуляции.

Рассмотрим следующий пример: существует каталог *v0*, содержащий в себе файл *prog.c*. В файле *prog.c* содержится исходный код программы, написанной на C:

```
//включаем header "myv0lib.h" в программу
#include "myv0lib.h"
//основная функция, исполняемая при запуске
int main()
{
    int x,y; //объявили x и y, типа int
    char s[64]; //объявили массив s длиной 64
```



```
//функции, вызываемые далее, хранятся в библиотеке myv0lib, описаны
соответствующим header'ом
x=myv0f1(6); //в переменную x сохраняем результат выполнения функции
myv0f1 от значения 6
y=myv0f2(7,8); //в переменную y сохраняем результат выполнения функции
myv0f2 от значений 7, 8
sprintf(s, "x=%d y=%d\n",x,y); //в переменную s записывается строка
приведенного в кавычках формата, символы %d заменяются аргументами x, y
write(1,s,sizeof(s)); //вывод на экран строки
return 0; //завершение программы с результатом 0 (как правило,
означающим стандартное закрытие)
}
```

В папке v0 так же содержится папка *libs*, содержащая в себе следующие файлы:

- myv0f1.c – файл на языке C, описывающий функцию myv0f1.c;
- myv0f2.c – файл на языке C, описывающий функцию myv0f2.c;
- myv0lib.h – header файл, описывающий библиотеку myv0lib;

Содержимое этих файлов можно посмотреть в приложении А.

Необходимо скомпилировать с помощью clang программу, созданную с помощью вышеуказанных файлов. Для этого необходимо создать Makefile для библиотеки myv0lib и программы prog, после чего выполнить их с помощью утилиты make. Сперва необходимо создать Makefile для компиляции библиотеки. Для этого, сперва нужно перейти в каталог *libs* и выполнить в нем команду *nano Makefile* для того, чтобы создать необходимый файл. Затем, файл необходимо заполнить следующим образом:

```
#Libs Make
all: libmyv0
myv0f1.o: myv0f1.c
clang -c -o myv0f1.o myv0f1.c
myv0f2.o: myv0f2.c
clang -c -o myv0f2.o myv0f2.c
libmyv0: myv0f1.o myv0f2.o
ar r ../libmyv0.a myv0f1.o myv0f2.o
ranlib ../libmyv0.a
clean:
rm -f *.o
```

При запуске этого файла утилитой *make* (утилита автоматически подхватывает файлы, названные как «Makefile» в той директории, в которой была запущена) будет выполнена следующая последовательность действий:

1. Сперва будет выполнена первая цель, описанная в файле, а именно – цель *all*.
2. Для достижения цели *all* необходим реквизит *libmyv0*, который в данный момент времени отсутствует. Соответственно, будет запущена цель *libmyv0*.
3. Для достижения цели *libmyv0* необходимы реквизиты *myv0f1.o* и *myv0f2.o*, которые так же отсутствуют. Соответственно, последовательно будут запущены цели *myv0f1.o* и *myv0f2.o*.
4. Для достижения цели *myv0f1.o* необходим реквизит *myv0f1.c*, который есть в папке, а значит цель можно достигнуть. Соответственно, будет выполнена команда *clang -c -o myv0f1.o myv0f1.c*, которая, используя

Примечание [Denis Sor1]: Добавит
ь инфу о том что должен быть TAB

компилятор *clang* создаст объект *myv0f1.o*, используя исходный код из файла *myv0f2.c*.

5. Аналогично 4 пункту, будет достигнута цель *myv0f2.o*.
6. После появления реквизитов из 4 и 5 пункта, можно достигнуть цель *libmyv0*. Соответственно, будет выполнена команда *ar r ../libmyv0.a myv0f1.o myv0f2.o*, которая заархивирует файлы *myv0f1.o myv0f2.o* в файл *../libmyv0.a*, а затем командой *ranlib* архив индексируется.
7. После выполнения цели из пункта 6 становится возможным достигнуть цели *all*, поскольку реквизит в наличии. Однако, в цели *all* не указано команд, которые необходимо выполнить, соответственно, скрипт просто завершится.

В ходе выполнения вышеописанной последовательности действий не была затронута цель *clean*. Эту цель можно выполнить командой *make clean*. Сама цель не требует реквизитов, и лишь удалит из каталога все файлы с расширением *.o*, удалив таким образом результаты компиляции файлов *myv0f1.c* и *myv0f2.c*. Теперь, необходимо скомпилировать саму программу *prog*, при этом указав что необходимо использовать библиотеку *libmyv0*. Для этого, сперва необходимо перейти к директории с файлом *prog.c*, и создать в этой директории *Makefile* со следующим содержанием:

```
#main Make
all: prog
prog: prog.c
    clang -oprog prog.c -lmyv0 -llibs/ -L.
clean:
    rm -f prog
```

Так, при запуске *make* будет выполнена компиляция и получен файл *prog*, который можно запустить командой

```
./prog
```

Однако, такой подход к компиляции – сперва библиотека, затем сама программа – не слишком удобен в случае, если необходимо компилировать несколько библиотек. Удобнее было бы сразу указать в *Makefile*, что необходимо компилировать библиотеку. Для этого, файл *Makefile* должен выглядеть следующим образом:

```
#main Make
all: libmyv0.a prog
libmyv0.a:
    cd libs && make
prog: prog.c
    clang -oprog prog.c -lmyv0 -llibs/ -L.
clean:
    cd libs && make clean
    rm -f prog libmyv0.a
```

Во-первых, теперь перед тем как выполнять цель *prog* необходимо выполнить цель *libmyv0.a*, в ходе которой будет запущен *make*-файл в директории с библиотекой, и уже после этого будет выполнена цель *prog*. Так же, была модифицирована цель *clean* для вызова цели *clean* из директории с библиотекой, и удаления скомпилированной библиотеки. Для того, чтобы проверить работу *make*-файла, сперва необходимо выполнить команду *make clean* для удаления всего, что было скомпилировано до этого, а затем – команду *make*, чтобы собрать программу с нуля и убедиться, что все работает.

Синтаксис, поддерживаемый утилитой *make*, так же позволяет использовать переменные. Например, чтобы объявить переменную, необходимо написать в *make*-файле до объявления целей:

X=

Для того, чтобы использовать переменную впоследствии, нужно будет написать *\${X}*. Тогда, при выполнении скрипта утилитой, вместо *\${X}* будет подставлено значение переменной *X*.

Например, можно модифицировать *make*-файл в директории с файлом *prog.c* следующим образом:

```
#main Make
CFLAG=-Ilibs/ -L. //объявили переменную
all: libmyv0.a prog
libmyv0.a:
    cd libs && make
prog: prog.c
    clang -oprog prog.c -lmyv0 ${CFLAG} //использовали переменную
clean:
    cd libs && make clean
    rm -f prog libmyv0.a
```

Более того, переменные можно объявлять в отдельных файлах. Например, создадим файл *Makefile.inc* со следующим содержанием:

CC=clang //объявили переменную

Изменим файл *Makefile* следующим образом:

```
#main Make
include Makefile.inc //подключили файл в котором происходит объявление
переменных
CFLAG=-Ilibs/ -L. //объявили переменную
all: libmyv0.a prog
libmyv0.a:
    cd libs && make
prog: prog.c
    ${CC} -oprog prog.c -lmyv0 ${CFLAG} //использовали переменную CC и
CFLAG
clean:
    cd libs && make clean
    rm -f prog libmyv0.a
```

Так же изменим *make*-файл, находящийся в папке *libs* следующим образом:

```
#Libs Make
include ../Makefile.inc //подключили файл в котором происходит объявление
переменных
all: libmyv0
myv0f1.o: myv0f1.c
    ${CC} -c -omyv0f1.o myv0f1.c //использовали переменную CC
myv0f2.o: myv0f2.c
    ${CC} -c -omyv0f2.o myv0f2.c //использовали переменную CC
libmyv0: myv0f1.o myv-f2.o
    ar r ../libmyv0.a myv0f1.o myv0f2.o
    ranlib ../libmyv0.a
clean:
    rm -f *.o
```

Подробнее про *make* можно почитать тут:

http://rus-linux.net/nlib.php?name=/MyLDP/algol/gnu_make/gnu_make_3-79_russian_manual.html

Задание к работе

- Создайте каталог MyProg, в нем – каталог libs. В каталоге MyProg создайте файл prog.c, в каталоге libs – файлы myv0f1.c, myv0f2.c, myv0lib.h. Содержимое файлов должно соответствовать приложению А.
- Создайте make-файлы для библиотеки в каталоге libs и программы в каталоге MyProg.
- Создайте собственную библиотеку в каталоге mylib, используя язык С, и модифицируйте программу prog таким образом, чтобы она использовала как библиотеку из папки libs, так и вашу библиотеку, выводя на экран результаты выполнения функций.
- Создайте make-файл для вашей библиотеки и модифицируйте make-файл в каталоге MyProg для автоматической компиляции вашей библиотеки и всего проекта.
- В случае, если при компиляции проекта clang выводит warning сообщения – модифицируйте проект таким образом, чтобы этих предупреждений не возникало.

Библиотека должна содержать в себе 3 функции:

- Первая функция получает значение int, умножает его на М и выводит результат.
- Вторая функция получает два значения int, и из большего из них вычитает 1.
- Третья функция получает два значения int, и к меньшему из них прибавляет М.

Каждая функция должна быть описана в отдельном файле. Значение М должно быть задано в header-файле, и соответствовать номеру студента в списке группы. Отчет должен содержать исходные коды созданных и модифицированных в рамках выполнения задания файлов (.c, .h, make-файлов), скриншот с результатом компиляции и скриншот с результатом работы программы.

Лабораторная работа №3 Системные вызовы

1. Содержание работы

Работа предназначена для ознакомления с принципами микроядерной структуры ядра minix и реализации вызовов.

В рамках представленного образа виртуальной машины реализован системный вызов записи значения в память:

1. Создан сервер операционной системы pg (файл /usr/src/minix/servers/pg/main.c и др.)
2. В сервер pg включена обработка запроса чтения значения из заполненного случайным образом массива
3. Разработан образец программы, выполняющей системные вызовы к серверу pg (файл /root/test_pg.c)

Необходимо:

1. в дополнение к вызову чтения в программе /root/test_pg.c, произвести вызов системного вызова записи значения по указанному индексу;
2. в дополнение к обработке вызова чтения, реализовать в сервере pg функцию записи значения по указанному индексу;
3. откомпилировать программу test_pg, пересобирать систему minix из исходных кодов, перезагрузить, проверить работоспособность;
4. Именовать системные вызовы в файле /usr/src/minix/include/minix/com.h по образцам, находящимся в том же файле. Сменить числовые константы на имена в файлах /usr/src/minix/servers/pg/main, /root/test_pg.c;
5. Вынести системные вызовы в файле /root/test_pg.c в отдельные функции по образцу;
6. откомпилировать программу test_pg, пересобирать систему minix из исходных кодов, перезагрузить, проверить работоспособность.

Разбиение по вариантам не предполагается.

2. Краткие теоретические сведения

МИКРОЯДРО

В отличие от традиционной архитектуры, согласно которой операционная система представляет собой монолитное ядро, реализующие основные функции по управлению аппаратными ресурсами и организующее среду для выполнения пользовательских процессов, микроядерная архитектура распределяет функции ОС между микроядром и входящими в состав ОС системными сервисами, реализованными в виде процессов, равноправных с пользовательскими приложениями.

Микроядро реализует базовые функции операционной системы, на которые опираются эти системные сервисы и приложения. В результате, такие важные

компоненты ОС как файловая система, сетевая поддержка и т. д. превращаются в по-настоящему независимые модули, которые функционируют как отдельные процессы и способны взаимодействовать с ядром и друг с другом.

Подробнее про minix см. [Таненбаум, Вудхалл, с. 138].

Структуру сообщений (тип message) см. в файле
/usr/src/minix/include/minix/ipc.h

ПРИМЕР КОМПИЛЯЦИИ И ЗАПУСКА ПРОГРАММЫ

```
root@minix:~# clang -otest_pg test_pg.c  
root@minix:~# ./test_pg
```

КОМПИЛЯЦИЯ И ПЕРЕЗАПУСК ОС

```
root@minix:~# cd /usr/src/releasetools/  
root@minix:/usr/src/releasetools# make hdboot  
root@minix:/usr/src/releasetools# reboot
```

3. Контрольные вопросы.

1. Перечислите основные функции микроядра.
2. Где хранятся сообщения?
3. Как реализована обратная посылка сообщения?
4. Что нужно сделать, чтобы добавить вызов?

Лабораторная работа №4 Бездисковая файловая система

1. Содержание работы

Работа предназначена для ознакомления с базовыми принципами работы файловых систем и структуры подсистемы управления файлами ОС.

В рамках представленного образа виртуальной машины реализован системный вызов записи значения в память:

1. Создан сервер файловой системы testfs (каталог /usr/src/minix/fs/testfs/)
2. Сервер testfs создан с использованием библиотеки VTreeFS (<http://wiki.minix3.org/doku.php?id=developersguide:vtreefs>)
3. В сервере testfs реализованы обработчики (callback hooks) подключения (mount) файловой системы и чтения содержимого файла с именем "test"

Действия по тестированию сервера testfs:

1. Компиляция и установка сервера
`make && make install`
2. Подключение файловой системы
`mount -t testfs none /mnt`
3. Просмотр содержимого файла
`cat /mnt/test`
4. Отключение файловой системы
`umount /mnt`

Необходимо:

1. В рамках сервера testfs реализовать обработчик по варианту;
2. Выявить назначение обработчика;
3. откомпилировать, проверить работоспособность.

Варианты:

1. `lookup_hook`
2. `write_hook`
3. `trunc_hook`
4. `mknod_hook`
5. `unlink_hook`
6. `slink_hook`
7. `rdlink_hook`
8. `chstat_hook`

Полезные команды:

1. Поиск файла по имени
`find /usr/src -name '*имя*'`
2. Поиск строки в файлах
- ⑩ `grep -ru 'строка' /usr/src`
- ⑩ `grep -u 'строка' /usr/src/minix/*/*.h`

Лабораторная работа №5 Свойства процессов

1. Содержание работы

Работа предназначена для ознакомления со структурами ядра операционной системы, содержащими информацию о процессах.

Необходимо выполнить следующие задания:

1. Модифицировать файловую систему procfs таким образом, чтобы в ее рамках появился новый файл, имеющий произвольное содержание.
2. Изучить структуры ядра, содержащие информацию, необходимую выполнения работы согласно варианту.
3. Получить необходимую информацию из ядра и отобразить в рамках файла, созданного в п.1.

Варианты:

1. Число спящих процессов
2. Число процессов «зомби»
3. Общее число процессов
4. Объем самого большого процесса (виртуальный)
5. Список названий процессов, начинающихся на букву «a»
6. Число процессов, принадлежащих пользователю root
7. Число процессов готовых к выполнению
8. Список процессов с номерами больше 10

Лабораторная работа №6

Создание процессов

Содержание работы

Работа предназначена для исследования способов создания процессов.

Необходимо выполнить следующие задания:

1. Создать процесс способом по варианту.
2. Получить информацию о коде завершения порожденного процесса.

Варианты

| Вариант | API | Среда | Тип процесса |
|---------|--------|---------------|-----------------|
| 1 | winapi | Visual studio | нить |
| 2 | winapi | mingw | нить |
| 3 | posix | cygwin | нить |
| 4 | posix | linux | нить |
| 5 | posix | minix | нить |
| 6 | winapi | Visual studio | Тяжелый процесс |
| 7 | winapi | mingw | Тяжелый процесс |
| 8 | posix | cygwin | Тяжелый процесс |
| 9 | posix | linux | Тяжелый процесс |
| 10 | posix | minix | Тяжелый процесс |

Лабораторная работа №7

Взаимодействие процессов winapi

Содержание работы

Цель работы: научиться организовывать взаимодействие двух процессов в среде Windows.

Необходимо выполнить следующие задачи:

1. Создать два процесса, тип процесса определить по номеру варианта.
2. Процесс ПЕРВЫЙ должен передать ВТОРОМУ сигнал событие.
3. ВТОРОЙ процесс должен послать первому произвольную строку, используя способ взаимодействия процессов согласно варианту.
4. ПЕРВЫЙ процесс должен напечатать на экране полученные данные.

Варианты

| Вариант | Тип процесса | Способ взаимодействия |
|---------|--------------|---------------------------------|
| 1 | нить | Неименованный (Анонимный) канал |
| 2 | процесс | Неименованный (Анонимный) канал |
| 3 | нить | Именованный канал |
| 4 | процесс | Именованный канал |
| 5 | нить | Мэйлслот |
| 6 | процесс | Мэйлслот |
| 7 | нить | Маппирование файла |
| 8 | процесс | Маппирование файла |

Лабораторная работа №8 Взаимодействие процессов posix

1. Содержание работы

Цель работы: научиться организовывать взаимодействие двух процессов.
Необходимо выполнить следующие задачи:

5. Создать два процесса, тип процесса определить по номеру варианта.
6. Процесс ОДИН должен передать ВТОРОМУ сигнал SIGUSR1.
7. ВТОРОЙ процесс должен послать первому произвольную строку, используя способ взаимодействия процессов согласно варианту.
8. ПЕРВЫЙ процесс должен напечатать на экране полученные данные.

Варианты

| Вариант | Тип процесса | Способ взаимодействия |
|---------|--------------|-----------------------|
| 1 | нить | Неименованный канал |
| 2 | процесс | Неименованный канал |
| 3 | нить | Именованный канал |
| 4 | процесс | Именованный канал |
| 5 | нить | IPC: сообщение |
| 6 | процесс | IPC: сообщение |
| 7 | нить | IPC: общая память |
| 8 | процесс | IPC: общая память |

1. Краткие теоретические сведения

Способы взаимодействия процессов

1. использование `rtgase` (взаимодействует отладчик и отлаживаемая программа)
2. передача сигналов (передаются только сигналы, данные передавать невозможно);
(`kill` – передаются; `signal` – установка обработчика)
3. неименованные каналы (`pipe`. Может взаимодействовать процесс и его потомки);
4. именованные каналы (специальный файл, `mknod` - создание);
5. используя текст IPC (межпроцессорное взаимодействие);
6. через систему сокетов (взаимодействие по сети).

Сигналы

За многими сигналами закреплены специальные функции.

signal.h - соответствует сигналу мнемокоду

#define SIGHUP 15

SIGHUP (Hang Up) опускание трубки телефона , заверш. управл. процесс

SIGINT (Interact) Ctrl+C прерывание с клавиатуры

QUIT- выход

ILL – неверная инструкция

FPE – деление на 0

KILL – нельзя обработать процессом

SEGV – нарушение сегментации

PIPE – возникновение проблем в конвейере

ALRM – сигнал будильника

TERM – один из основных сигналов для завершения процесса

USR1 – не закреплены никакие функции

USR2 – можно определить самому

CHLD – порожденный процесс завершился

STOP – не обрабатывается процессами

CONT - продолжение

PWR – нет напряжения в сети

dd – снимает образы CD/DVD

dd if = /dev/cdrom of = /dev/hda2

ps – номер процесса

dd – обраб. сигн. USR1 USR2

kill посылает сигналы. Соответствует вызову:

kill (<номер процесса> <номер сигнала>)

Если задавать сигнал в форме SIG..., необходимо подключить signal.h

Сигналы обрабатываются асинхронно. Обработчик сигнала устанавливается с помощью вызова:

`signal (<номер сигнала>, <обработчик>)`

 SIG_INT SIG_DFL указатель на функцию
 (игнорирование) (по умолчанию)

Функция – обработчик:

`void <имя> (int <номер сигнала>)`

Пример:

```
#include <signal.h>
```

```
#include <stdio.h>
```

```
void obr (int n)
```

```
{ printf ("%d", n);
```

```
    fflush (stdout); } // чистит буфер вывода, чтобы выводил сразу
```

```
main ()
```

```
{ signal(SIGUSR2, obr);
```

```
    while (1); }
```

```
// kill – SIGUSR2 <номер процесса>
```

```
// while (1); - грузит ЦП, поэтому
```

```
while (1); pause;
```

```
// БУДИЛЬНИК
```

```
wakeup ()
```

```
{ printf (“Я проснулся”);
```

```
    fflush (stdout); }
```

```
main ()
```

```
{ signal (SIGALRM, wakeup);
```

```
    while (1)
```

```
        { alarm (5); // сигнал будильника с задержкой на 5 секунд
```

```

        pause();

    }}

    // вместо alarm(5) имитация его kill

    kill (getpid(),SIGALRM); //возвращает номер процесса

    wakeup ()

    {   printf ('умираю');

        exit(0);

    }

    wakeup1 ()

    {   printf ('успел');

        wait(0);

    }

    main ()

    {   char buf [16];

        int nch, inp; // номер пород. процесса, дескриптор файла

        signal (SIGALRM, wakeup);

        signal (SIGCHLD, wakeup1);

        if (nch = fork()) // родительский

        {   sleep(1);

            kill (nch, SIGALRM);

        }

        else // потомок

        {   inp = open ("/dev/tty", )_RDOHLY);

            read (inp, buf, sizeof (buf));

        }}

    sigation – лучший вариант, чем signal

```

sigprocmask – можно задавать номер сигнала, который будут игнорировать

Неименованные каналы

Применяются только при взаимодействии между процессом и его потомком.

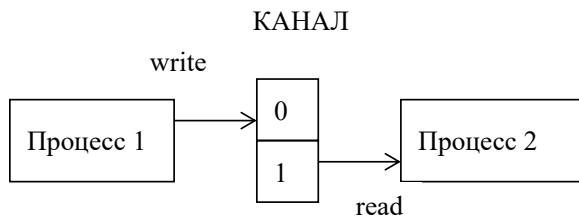
Создается дескриптор, состоящий из двух элементов:

Командой pipe (fdp) он определяется.

pipe - создание неименованного канала

```
int pipe (fildes)
int fildes [2];
```

Системный вызов pipe создает механизм ввода/вывода, называемый каналом, и возвращает два дескриптора файла fildes[0] и fildes[1]. Дескриптор fildes[0] открыт на чтение, дескриптор fildes[1] - на запись.



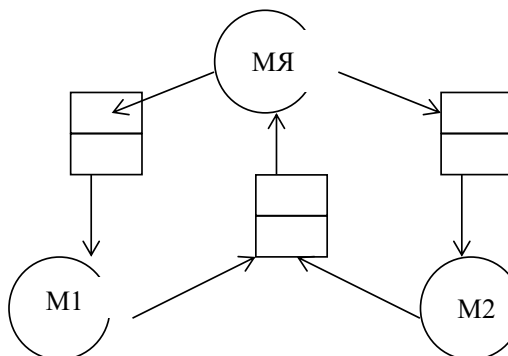
Канал буферизует до 5120 байт данных; запись в него большего количества информации без считывания приведет к блокированию пишущего процесса. Посредством дескриптора fildes[0] информация читается в том же порядке, в каком она записывалась с помощью дескриптора fildes[1].

Системный вызов pipe завершается неудачей, если выполнено хотя бы одно из следующих условий:

- 1) Превышается максимально допустимое количество файлов, открытых одновременно в одном процессе.
- 2) Переполнена системная таблица файлов.

При успешном завершении результат равен 0; в случае ошибки возвращается -1, а переменной errno присваивается код ошибки.

Использовать один канал для двух сторон обмена неудобно. Взаимодействуют только родственные каналы.



Если используется один канал, сделать запись в него, а потом прочитать то, что только что было в него записано. Для двустороннего взаимодействия можно создать второй канал.

Пример:

Записывать будем с помощью `write (fdp [1]);`

Читать с помощью `read (fdp [0]).`

`main ()`

```
{ char buf [2];
```

```
    int to [2], from [2];
```

```
    buf [1] = '\0'; //конец строки
```

```
    pipe (to); pipe (from);
```

```
    if (fork()) //родитель
```

```
    {      signal (SIGCHLD, kid);
```

```
        while(1)
```

```
        {      scanf ("%c", &buf [0]);
```

```
                write (to [1], buf, 2);
```

```
                read (from [0], buf, 2);
```

```
                printf ("%s", buf);
```

```
    }}
```

```
else //порожденный
```

```
{ while(1)
```

```
    {      read (to [0], buf, 2);
```

```
        if (buf [0] == 'e') exit(0);
```

```
        buf [0]++;
```

```
        write (from [1], buf, 2);
```

```
    }}
```



```
void kid()
{
    wait (0);

    exit (0); //завершение в родит. процесс
}
```

Именованный канал

При работе с именованным каналом создаем специальный файл:

mknod filename p, где p - тип файла.

Работа с именованным каналом производится также как с обычным файлом (теми же системными вызовами), за исключением того, что реально на диске информация не сохраняется, а передается от процесса, вызвавшего запись, процессу, вызвавшему чтение.

Пример:

```
main ()
{
    int rd = open ('имя ф. канала', O_WRONLY);

    write (fd, "Hello", 6);

    close (fd);
}

main ()
{
    char buf [16];

    int fd = open ('имя ф. канала', O_RDONLY);

    read (fd, buf, 16);

    printf ("%s", buf);

    close (fd);
}
```

пакет IPC

В IPC содержится три механизма взаимодействия:

1. механизм сообщений;
2. механизм распределения памяти;

3. семафоры.

| | Сообщения | Память | Семафоры |
|-----------|-----------|--------|----------|
| Создание | msgget | shmget | semget |
| Настройка | msgctl | shmctl | semctl |
| Работа | msgrcv | shmat | semop |
| | msgsnd | shmdt | |

1. Механизм сообщений позволяет принимать и посылать потоки сформированных данных.

За передачу сообщений отвечают четыре системных вызова:

msg get \approx возвращает дескриптор сообщения;

msg clt \approx устанавливает параметры сообщений;

msg cnt \approx передает сообщение;

msg rcv \approx принимает сообщение.

2. Механизм распределения памяти позволяет совместно использовать отдельные части виртуального адресного пространства.

shm get \approx создает новую область;

shm at \approx логически присоединяет;

shm dt \approx логически отсоединяет;

shm ctl \approx работает с параметрами области.

3. Семафоры синхронизацию выполнения параллельных процессов. В любой момент времени над

семафором возможна только одна реализация.

sem get \approx создание набора семафоров;

sem ctl \approx управление этим набором;

sem op \approx работа со значениями.

Семафоры

Используются для синхронизации выполнения приложений и защиты критических секций.

```
#define SEMKEY 77
```

```

union semun //одно из полей буде использоваться
{
    int val;

    struct semid_ds *bat;

    unsigned short *array;

    struct seminfo *buf;
}

main()
{
    union semun inisem; //для инициализацииq

    unshort ainisem[1]={1};

    int semid;

    int i,j,pid;

    struct sembuf p,v;

    semid=semget(SEMKEY,1,0777|[IPC_CREAT]); //создание, IPC_CREAT -
    макрос создания, 1 – число символов, 0777 - моды доступа

    inisem=ainisem; //команда

    semctl(semid,0,SETALL,inisem); //инициализация

    p.sem_num=0;

    p.sem_op=-1;

    p.sem_flg=SEM_UNDO;

    v.sem_num=0;

    v.sem_op=1;

    v.sem_flg=SEM_UNDO;

    fork();

    pid=getpid(); //определяем номер процесса

    for(i=0;;i<10;i++)

```

```

{
semop(semid,&p,1); // сколько операций выполнено

//критическая секция
for(j=0;g<5;g++)
{
printf(“%d%d”, pid, j);

semop(semid,&v,1);
}}

```

Механизм передачи сообщений

Системный вызов msgget()

Системный вызов msgget предназначен для выполнения операции доступа к очереди сообщений и, в случае ее успешного завершения, возвращает дескриптор для этой очереди (целое неотрицательное число).

```
int msgget(key_t key, int msgflg);
```

Описание системного вызова

Параметр key является ключом для очереди сообщений. Параметр msgflg – флаги – играет роль только при создании новой очереди сообщений и определяет права различных пользователей при доступе к очереди.

Возвращаемое значение

Системный вызов возвращает значение дескриптора System V IPC для очереди сообщений при нормальном завершении и значение -1 при возникновении ошибки.

Системный вызов msgsnd()

Вызов msgsnd() копирует пользовательское сообщение в очередь сообщений.

```
int msgsnd(int msqid, struct msgbuf *ptr, int length, int flag);
```

Описание системного вызова

Системный вызов копирует сообщение, расположенное по адресу, на который указывает параметр ptr, в очередь сообщений, заданную дескриптором msqid.

Параметр flag может принимать два значения: 0 и IPC_NOWAIT.

Возвращаемое значение

Системный вызов возвращает значение 0 при нормальном завершении и значение -1 при возникновении ошибки.

Системный вызов `msgrcv()`

```
int msgrcv(int msqid, struct msgbuf *ptr, int length, long type,
int flag);
```

Описание системного вызова

Системный вызов `msgrcv` предназначен для получения сообщения из очереди сообщений, т. е. является реализацией примитива `receive`.

Параметр `msqid` является дескриптором для очереди, из которой должно быть получено сообщение.

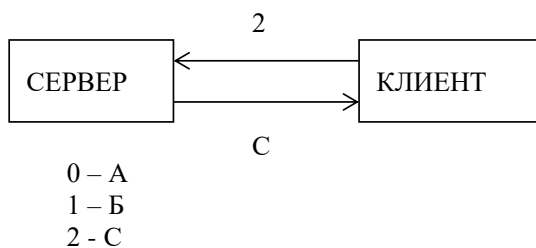
Параметр `length` должен содержать максимальную длину полезной части информации.

Параметр `flag` может принимать значение 0 или быть какой-либо комбинацией флагов `IPC_NOWAIT` и `MSG_NOERROR`.

Возвращаемое значение

Системный вызов возвращает при нормальном завершении действительную длину полезной части информации (т. е. информации, расположенной в структуре после типа сообщения), скопированной из очереди сообщений, и значение -1 при возникновении ошибки.

Все прочитанные сообщения по умолчанию удаляются из очереди.



Сервер:

```
#define MSGKEY 81

struct msgform {

long mtype; //идентификатор сообщения

char text [256]; //данные

}
```

```

main ()

{char mas [] = {'a','b','c'};

struct msgform msg;

int msgid, *pint;

char *pchar;

msgid = msgget (MSGKEY, 0777 IPC_CREATE);

pint = (int*) msg.txt;

pchar = (char*) msg.text;

msgrcv (msgid, &msg, sizeof (int), 8, 0) //8 – идентификатор

*pchar = mas [*pint];

msg.mtype = 9;

msgsnd (msgid, &msg, sizeof(char),0); // 0 - флаг

}

```

Клиент:

```

#define MSGKEY 81

struct msgform {

long mtype; //идентификатор сообщения

char text [256]; //данные

}

main ()

{char mas [] = {'a','b','c'};

struct msgform msg;

int msgid, *pint, g;

char *pchar;

msgid = msgget (MSGKEY, 0777);

pint = (int*) msg.txt;

```

```
pchar = (char*) msg.text;

scanf ("%d", &g);

*pint = g;

msg.mtype = 8;

msgsnd (msgid, &msg, sizeof(int),0); // 0 – флаг

msgrcv (msgid, &msg, sizeof(char), g , 0);

printf ('номер %d у буквы %c', g , *pchar);

}
```

Лабораторная работа №9

Взаимодействие процессов по сети

Содержание работы

Цель работы: научиться организовывать взаимодействие двух процессов.

Необходимо разработать программу на Си из двух частей (клиент и сервер). Обмен между клиентом и сервером должен происходить с использованием одного из способов взаимодействия процессов (по варианту). При этом на каждое обращение клиента сервер должен обрабатывать в отдельном потоке (способ организации потока по вариантам).

Оrientировочная последовательность действий

Сервер

- принимает соединение от клиента (по варианту)
- запускает процесс обработки запроса (по варианту)
- процесс обработки запроса обрабатывает запрос
- процесс обработки запроса передает клиенту ответ

Клиент

- запрашивает у пользователя данные для запроса (по варианту)
- посылает запрос серверу
- принимает ответ от сервера
- ответ выдает ответ пользователю

Варианты заданий

Таблица 1 - Варианты заданий

| Вариант | ОС | Способ взаимодействия | Способ организации многозадачности (потоков) | Запрос (расшифровка приведена в п. "Запросы") |
|---------|-------|-----------------------|--|---|
| 1 | minix | неименованные каналы | процессы | 1 |
| 2 | " | " | нити | 1 |
| 3 | " | именованные каналы | процессы | 1 |
| 4 | " | " | нити | 1 |
| 5 | " | сигналы | процессы | 3 |
| 6 | " | " | нити | 3 |
| 7 | " | сокеты TCP | процессы | 1 |

| | | | | | |
|----|-------|---------------------------|----------|---|--|
| 8 | " | " | нити | 1 | |
| 9 | " | сокеты UDP | процессы | 1 | |
| 10 | " | " | нити | 1 | |
| 11 | " | неименованные каналы | процессы | 2 | |
| 12 | " | " | нити | 2 | |
| 13 | " | именованные каналы | процессы | 2 | |
| 14 | " | " | нити | 2 | |
| 15 | " | неименованные каналы | процессы | 4 | |
| 16 | " | " | нити | 4 | |
| 17 | " | сокеты TCP | процессы | 1 | |
| 18 | " | " | нити | 1 | |
| 19 | linux | IPC механизм общей памяти | процессы | 1 | |
| 20 | " | " | нити | 1 | |

Запросы

1. клиент должен передать серверу число, сервер должен передать клиенту сумму чисел, поступивших серверу, начиная от старта сервера
2. сервер должен передать клиенту номер запроса клиента, начиная от старта сервера
3. сервер должен передать клиенту четность номера запроса клиента, начиная от старта сервера
4. клиент должен передать серверу строку, сервер должен передать клиенту конкатенацию строк, поступивших серверу, начиная от старта сервера

Краткие теоретические сведения

Сокеты - универсальные методы взаимодействия процессов на основе использования многоуровневых сетевых протоколов. Сокеты предназначены для работы по сети: блокируют драйверы, для удобства выполнения локальных подпрограмм. Сокеты используют для работы по IP – сетям.

Используют клиент-серверный механизм.



Сокеты находятся в областях связи (доменах). Домен сокета - это абстракция, которая определяет структуру адресации и набор протоколов. Сокеты могут соединяться только с сокетами в том же домене. Всего выделено 23 класса сокеты (см. файл `<sys/socket.h>`), из которых обычно используются только UNIX-сокеты и Интернет-сокеты.

Поддерживаются домены:

- "UNIX system" - для взаимодействия процессов внутри одной машины
- "Internet" (межсетевой) - для взаимодействия через сеть с помощью протокола

У сокета 3 атрибута:

- 1) домен
- 2) тип
- 3) протокол

Для создания сокета используется системный вызов `socket`.

`s = socket(domain, type, protocol);`

Например, для использования особенностей Internet, значения параметров должны быть следующими:

`s = socket(AF_INET, SOCK_STREAM, 0);`

Основные типы сокеты:

- 1) Поточный

- обеспечивает двухсторонний, последовательный, надежный, и недублированный поток данных без определенных границ. Тип сокета - `SOCK_STREAM`, в домене Интернета он использует протокол TCP.

2) Датаграммный

- поддерживает двухсторонний поток сообщений. Приложение, использующее такие сокеты, может получать сообщения в порядке, отличном от последовательности, в которой эти сообщения посылались. Тип сокета - `SOCK_DGRAM`, в домене Интернета он использует протокол UDP.

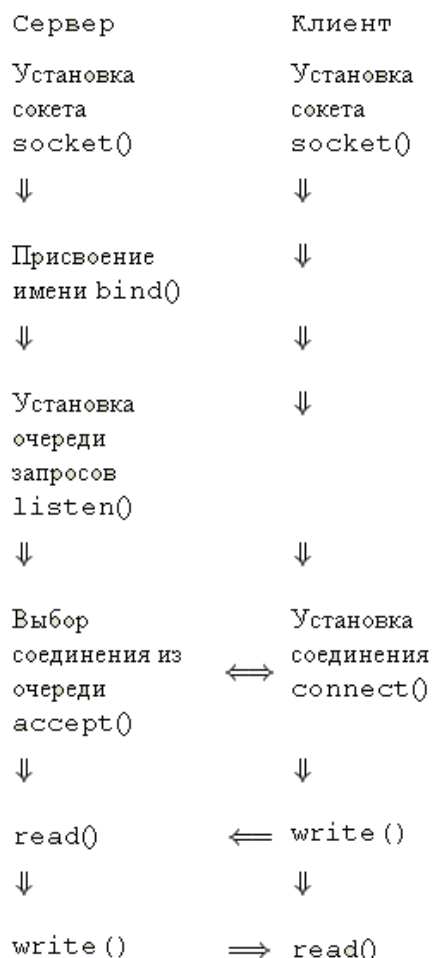
3) Сокет

последовательных пакетов - обеспечивает двухсторонний, последовательный, надежный обмен датаграммами фиксированной максимальной длины. Тип сокета - `SOCK_SEQPACKET`. Для этого типа сокета не существует специального протокола.

4) Простой

сокет - обеспечивает доступ к основным протоколам связи.

Обмен между сокетами происходит по следующей схеме:



Типы передаваемых пакетов:

SOCK_STREAM соответствует потоковым сокетам, реализующим соединения «точка-точка» с надежной передачей данных.

SOCK_DGRAM указывает датаграммный сокет. Датаграммные сокеты осуществляют ненадежные соединения при передаче данных по сети и допускают широковещательную передачу данных.

SOCK_RAVE для низкоуровневого управления пакетами данных

AF_INET сокет для работы по сети

AF_UNIX соответствует сокетам в файловом пространстве имен

Причины успеха сокетов заключаются в их простоте и универсальности. Программы, обменивающиеся данными с помощью сокетов, могут работать в одной системе и в разных, используя для обмена данными как специальные объекты системы, так и сетевой стек. Как и каналы, сокеты используют простой интерфейс, основанный на «файловых» функциях read(2) и write(2) (открывая сокет, программа Unix получает дескриптор файла, благодаря которому можно работать с сокетами, используя файловые функции), но, в отличие от каналов, сокеты позволяют передавать данные в обоих направлениях, как в синхронном, так и в асинхронном режиме.

Действия с сокетами

Установка связи:

Со стороны клиента связь устанавливается с помощью стандартной функции connect, которая иницирует установление связи на сокете, используя дескриптор сокета s и информацию из структуры serveraddr, имеющей тип sockaddr_in, которая содержит адрес сервера и номер порта на который надо установить связь:

```
error = connect(s, serveraddr, serveraddrlen);
```

Со стороны сервера процесс установления связи сложнее. Для этих целей используется системный вызов listen:

```
error = listen(s, qlength);
```

где s это дескриптор сокета, а qlength это максимальное количество запросов на установление связи, которые могут стоять в очереди.

Передача данных:

Когда связь установлена, с помощью различных функций может начаться процесс передачи данных. При наличии связи, пользователь может посылать и получать сообщения с помощью функций read и write:

```
write(s, buf, sizeof(buf)); read(s, buf, sizeof(buf));
```

Вызовы send и recv практически идентичны read и write, за исключением того, что добавляется аргумент флагов.

```
send(s, buf, sizeof(buf), flags); recv(s, buf, sizeof(buf), flags);
```

Закрывание сокетов:

Если сокет больше не используется, процесс может закрыть его с помощью функции close, вызвав ее с соответствующим дескриптором сокета:

```
close(s);
```

Приложение 1. Командный интерпретатор shell

shell является удобным и часто используемым интерпретируемым языком программирования. Он содержит стандартные конструкции для циклов, ветвления, объявления функций и т. п. Данный язык часто используется в UNIX-подобных системах при создании различных сценариев (скриптов) работы, в частности, сценариев автоматического конфигурирования исходных кодов программ перед их компиляцией. Отличительная особенность языка sh — многие операции, которые в традиционных языках программирования являются встроенными, выполняются с помощью вызова внешних программ.

Shell обрабатывает команды трех типов:

- Во-первых, в качестве имени команды может быть указано имя исполняемого файла в объектном коде, полученного в результате компиляции исходного текста программы (например, программы на языке Си).

- Во-вторых, именем команды может быть имя командного файла, содержащего набор командных строк, обрабатываемых shell'ом.

- В-третьих, команда может быть внутренней командой языка shell (в отличие от исполняемого файла). Наличие внутренних команд делает shell языком программирования в дополнение к функциям командного процессора; командный язык shell включает команды организации циклов (for-in-do-done и while-do-done), команды выполнения по условиям (if-then-else-fi), оператор выбора, команду изменения текущего для процесса каталога (cd) и некоторые другие. Синтаксис shell'a допускает сравнение с образцом и обработку параметров. Пользователям, запускающим команды, нет необходимости знать, какого типа эти команды.

Командный процессор shell ищет имена команд в указанном наборе каталогов, который можно изменить по желанию пользователя, вызвав shell. Shell обычно исполняет команду синхронно, с ожиданием завершения выполнения команды прежде, чем считать следующую командную строку. Тем не менее, допускается и асинхронное исполнение, когда очередная командная строка считывается и исполняется, не дожидаясь завершения выполнения предыдущей команды. О командах, выполняемых асинхронно, говорят, что они выполняются на фоне других команд.

ПРОСТЕЙШИЕ СРЕДСТВА SHELL

- ; - последовательное выполнение команд
\$ sleep 20 ; reboot
- & - асинхронное (фоновое) выполнение предшествующей команды
\$ cc pr.c &
- && - выполнение последующей команды при условии нормального завершения предыдущей
\$ mkdir sun && rmdir sun
- || - выполнение последующей команды при условии ненормального завершения предыдущей
\$ cd sun || mkdir sun
- {} — группировка команд в блок
- () — кроме выполнения функции группировки, выполняют и функцию вызова нового экземпляра интерпретатора shell
- . - для выполнения программы из файла в текущем Shell-e

СТАНДАРТНЫЙ ВВОД ВЫВОД

'stdin' – стандартный ввод осуществляется с клавиатуры терминала
 'stdout' – стандартный вывод направлен на экран терминала
 'stderr' – стандартный файл диагностических и отладочных обобщений

Работа с ними осуществляется как с файлами и им соответствуют первые три дескриптора.

stdin - 0
 stdout - 1
 stderr - 2

Эти файлы можно переопределять регулярными файлами.

Пользователь имеет удобные средства перенаправления ввода и вывода на другие файлы (устройства). Символы '>' и '>>' обозначают перенаправление вывода.

\$ команда > f2

В этом случае f2 создается. Если он существовал до этого, то он обнулится.

Для того, чтобы файл не обнулялся нужно использовать:

\$ команда >> f2

Символы '<' и '<<' обозначают перенаправление ввода.

команда < f1

Перенаправление вывода команды в строчку осуществляется с помощью ''

\$ cd `pwd`

Символ '|' используется для организации конвейера между командами. Это обозначает, что стандартный выход одной направляется на стандартный вход другой.

\$ ls -l | more

ls выдаёт файлы, а с помощью more их построчно выводит.

Поскольку shell является пользовательской программой и не входит в состав ядра операционной системы, его легко модифицировать и помещать в конкретные условия эксплуатации.

КОМАНДЫ SHELL

sh - вызывает командный процессор. Shell может работать в двух режимах: интерактивном режиме (режиме командной строки) и режиме выполнения программы. Символ "\$" в начале строки обозначает, что shell находится в интерактивном режиме и готов к вводу команд.

Синтаксис:

sh [опции] [программа_на_shell]

Пример:

\$ sh prog.sh

man – выдает справку

cp - копирует файлы. Можно скопировать один файл в другой или список файлов в каталог.

Синтаксис:

cp исходный_файл файл_назначения или

cp исходный_список каталог_назначения

где: исходный_файл - это файл, который нужно скопировать;
 файл_назначения - имя файла, в который будет скопирован исходный_файл;
 исходный_список - это список файлов для копирования, разделенных пробелами;

каталог_назначения - это каталог, куда копируются файлы.

Пример:

```
$ cp a.txt /tmp
```

mv - пересылает или переименовывает файл.

Синтаксис:

```
mv [-fi] oldfile newfile
```

```
mv [-fi] file... destination_directory
```

Команда mv в первой форме пересылает(или изменяет) файл oldfile с превращением его в newfile. Команда mv во второй форме пересылает один или несколько файлов в каталог destination_directory.

Пример:

```
$ mv f1.txt f2.txt
```

who - показывает пользователей, вошедших в систему.

Синтаксис:

```
who [am i]
```

Команда who показывает Unix-имена пользователей, которые к данному времени вошли в систему.

Команда who am i показывает ваше входное имя.

echo - берет аргументы, которые ей передали, и записывает их в стандартный вывод.

Синтаксис:

```
echo [-n] строка
```

где: -n - это ключ, который подавляет особенность выводить все выходные данные с новой строки.

Пример:

```
$ echo "Hello All!"
```

date - устанавливает/выводит системную дату и время.

Синтаксис:

```
date MDhmY
```

где: M - месяц(01-12); D - день(01-31); h - час(00-23); m - минуты(00-59); Y - год(00-99).

cat - связывает или соединяет файлы вместе. Эту команду также можно использовать для вывода файла на экран. Если никакие файлы не указаны, команда cat читает со стандартного ввода. Синтаксис:

```
cat [ключи] [список_файлов]
```

Пример:

```
$ cat f1 f2 > f3
```

mkdir - создает каталог.

Синтаксис:

```
mkdir название_каталога
```

Пример:

```
$ mkdir v10/a
```

file - определяет тип файла.

Синтаксис:

`file [ключи] [список-файлов]`

ключи: `-f` - проверяет файлы из списка

type - указывает физическое нахождение вызванной программы.

Пример:

`$ type myprog`

Результатом действия является: `/bin/myprog`

find - просматривает список каталогов, отыскивая файлы удовлетворяющие некоторому критерию.

Синтаксис:

`find [список_каталогов] [спецификация_сопоставления]`

Спецификация_сопоставления определяет условия соответствия для искомых файлов и может принимать следующие значения:

`-name` файл - заставляет команду `find` искать указанный файл;

`-print` - выводит имена найденных файлов.

Пример:

`$ find . -name myfile -print`

(поиск файла с именем `myfile` ведется в текущем каталоге и его подкаталогах, когда файл найден, на экран выводится полное имя маршрута к нему).

grep - отыскивает текст в файлах.

Синтаксис:

`grep [-il][-e]expression[file...]`

ключи: `-e` `expression` -здесь `expression` есть отыскиваемый текст. Для задания одного выражения указывать `-e` перед ним не нужно. Если выражение `expression` содержит пробелы, они должны быть заключены в кавычки.

`-i` - игнорирует различие прописных и строчных букв.

`-l` - вместо показа соответствующих строк выводит только файлы, имеющие соответствия.

Пример: для отыскания всех файлов в текущем каталоге, содержащих слово "john", применяется команда:

`$ grep -i john`

wc - выводит число строк, слов и символов в файле.

Синтаксис:

`wc [-clw] file...`

Команда `wc` подсчитывает число символов, строк и слов в файле или файлах.

Ключи:

`-c` -подсчитывает только число символов.

`-l` -подсчитывает только число строк.

`-w` -подсчитывает только число слов.

cal - отображает календарь в стандартный вывод. По умолчанию берется текущий месяц и год.

Синтаксис:

`cal [месяц] [год]`

Пример:

`$ cal 2000`

(выводит календарь на 2000 год).

diff - сравнивает два текстовых файла и сообщает, что должно быть сделано, чтобы один из них стал подобен другому.

Синтаксис:

diff [ключи] старый_файл новый_файл

Список использованных источников

1. Таненбаум Э., Вудхал А. Операционные системы. Разработка и реализация. 3-е изд. – СПб.: Питер, 2007. – 704 с.
2. Иртегов Д.В. Введение в операционные системы. - СПб: БХВ-Петербург, 2002.- 624с.
3. Рейчард К., Фостер-Джонсон Э. UNIX.- СПб: Питер, 1999.- 374с.
4. Келли-Бутл С. Введение в UNIX: Пер. с англ.- М.: Лори, 1997.- 341с.
5. Морис Бах. Архитектура Unix. <http://lib.ru/LINUXGUIDE/>