

Министерство науки и высшего образования Российской  
Федерации  
Калужский филиал  
федерального государственного бюджетного образовательного  
учреждения высшего образования  
**«Московский государственный технический университет  
имени Н.Э. Баумана  
(национальный исследовательский университет)»  
(КФ МГТУ им. Н.Э. Баумана)**

Ю.С. Белов, Е.А. Черепков

## СОЗДАНИЕ СЦЕНАРИЯ ДЛЯ КОНФИГУРАЦИИ СИТЕМЫ

Методические указания к лабораторной работе  
по дисциплине «Операционные системы»

Калуга, 2018


УДК 004.62  
ББК 32.972.1  
Б435

Методические указания составлены в соответствии с учебным планом КФ МГТУ им. Н.Э. Баумана по направлению подготовки 09.03.04 «Программная инженерия» кафедры «Программного обеспечения ЭВМ, информационных технологий».

Методические указания рассмотрены и одобрены:


- Кафедрой «Программного обеспечения ЭВМ, информационных технологий» (ИУ4-КФ) протокол № 51.4/6 от «20» февраля 2019 г.

Зав. кафедрой ИУ4-КФ

 к.т.н., доцент Ю.Е. Гагарин


- Методической комиссией факультета ИУ-КФ протокол № 9 от «04» 03 2019 г.

Председатель методической  
комиссии факультета ИУ-КФ

 к.т.н., доцент М.Ю. Адкин

- Методической комиссией  
КФ МГТУ им.Н.Э. Баумана протокол № 5 от «5» 03 2019 г.

Председатель методической комиссии  
КФ МГТУ им.Н.Э. Баумана

 д.э.н., профессор О.Л. Перерва

Рецензент:

к.т.н., доцент кафедры ИУ3-КФ

 А.В. Фиошин

Авторы

к.ф.-м.н., доцент кафедры ИУ4-КФ  
ассистент кафедры ИУ4-КФ

 Ю.С. Белов  
 Е.А. Черепков

#### Аннотация

Методические указания к выполнению лабораторной работы по курсу «Операционные системы» содержат общие сведения о создании сценария для конфигурации системы.

Предназначены для студентов 3-го курса бакалавриата КФ МГТУ им. Н.Э. Баумана, обучающихся по направлению подготовки 09.03.04 «Программная инженерия».

© Калужский филиал МГТУ им. Н.Э. Баумана, 2019 г.  
© Ю.С. Белов, Е.А. Черепков, 2019 г.

## ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ .....	4
ЦЕЛЬ И ЗАДАЧИ РАБОТЫ, ТРЕБОВАНИЯ К РЕЗУЛЬТАТАМ ЕЕ ВЫПОЛНЕНИЯ.....	5
КРАТКАЯ ХАРАКТЕРИСТИКА ОБЪЕКТА ИЗУЧЕНИЯ, ИССЛЕДОВАНИЯ .....	6
НАПИСАНИЕ СКРИПТОВ.....	10
ЗАПУСК И ОСТАНОВКА ДЕМОНОВ.....	17
ПРИСОЕДИНЕНИЕ RS.D СКРИПТА К ИНФРАСТРУКТУРЕ .....	26
СОЗДАНИЕ БОЛЕЕ ГИБКИХ RS.D СКРИПТОВ.....	32
ЗАДАНИЕ НА ЛАБОРАТОРНУЮ РАБОТУ .....	36
КОНТРОЛЬНЫЕ ВОПРОСЫ И ЗАДАНИЯ .....	37
ФОРМА ОТЧЕТА ПО ЛАБОРАТОРНОЙ РАБОТЕ .....	38
ОСНОВНАЯ ЛИТЕРАТУРА .....	39
ДОПОЛНИТЕЛЬНАЯ ЛИТЕРАТУРА .....	39

## **ВВЕДЕНИЕ**

Настоящие методические указания составлены в соответствии с программой проведения лабораторных работ по курсу «Операционные системы» на кафедре «Программное обеспечение ЭВМ, информационные технологии» факультета «Информатика и управление» Калужского филиала МГТУ им. Н.Э. Баумана.

Методические указания, ориентированные на студентов 3-го курса направления подготовки 09.03.04 «Программная инженерия», содержат описание создания сценариев для конфигурации системы.

Методические указания составлены для ознакомления студентов с созданием сценариев для конфигурации системы. Для выполнения лабораторной работы студенту необходимы минимальные знания об операционной системе FreeBSD.

## **ЦЕЛЬ И ЗАДАЧИ РАБОТЫ, ТРЕБОВАНИЯ К РЕЗУЛЬТАТАМ ЕЕ ВЫПОЛНЕНИЯ**

Целью выполнения лабораторной работы является закрепление полученных навыков по настройке основных сервисов системы FreeBSD.

Основными задачами выполнения лабораторной работы являются:

1. Сконфигурировать систему исходя из заданной вам схемы сети (сетевые интерфейсы, маршрутизация, DNS).

Результатами работы являются:

1. Демонстрация навыков по написанию сценариев.
2. Подготовленный отчет.

## КРАТКАЯ ХАРАКТЕРИСТИКА ОБЪЕКТА ИЗУЧЕНИЯ, ИССЛЕДОВАНИЯ

Практическое применение знаний о подсистеме *rc.d*, полученных из официальной документации BSD далеко не всегда бывает простым делом для новичка. В этой статье мы рассмотрим несколько типичных вариантов скриптов запуска, покажем, как использовать возможности *rc.d* в каждом из этих случаев и обсудим как это работает. Этот обзор должен дать вам представление о подсистеме *rc.d* для последующей эффективной разработки.

Когда-то BSD имела монолитный скрипт запуска */etc/rc*. Его вызывал *init* во время загрузки системы, и этот скрипт выполнял все задачи в пространстве пользователя (*userspace*), необходимые для многопользовательских операций: проверку и монтирование файловых систем, настройку сети, запуск демонов и тому подобное. Но список таких задач не одинаков в разных системах, администраторам необходима более тонкая настройка. За исключением отдельных моментов, */etc/rc* легко поддавался модификации, и по-настоящему увлеченные люди часто и с удовольствием его модифицировали.

Основная проблема подобного монолитного решения заключалась в том, что оно не предоставляло механизма контроля над отдельными компонентами, запускаемыми из скрипта */etc/rc*. К примеру, */etc/rc* не мог перезапустить отдельный сервис. Системному администратору приходилось находить идентификатор процесса вручную, убивать этот процесс, дожидаться, фактического завершения процесса, находить в файле */etc/rc* параметры, с которыми этот процесс был запущен, вручную запускать процесс из командной строки. Эта последовательность действий становилась еще более запутанной и сложной, если перезапускаемый сервис порождал несколько процессов или требовал дополнительных действий при запуске. В двух словах, монолитный скрипт запуска не мог выполнить основную задачу всех скриптов - сделать жизнь системного администратора проще.

Позже были предприняты попытки разделить `/etc/rc` на части, целью вынести некоторые наиболее важные элементы. Замечательный пример этого - скрипт `/etc/netstart`, инициализирующий работу сети. Этот скрипт позволял запустить сеть из однопользовательского режима, но его невозможно было полноценно интегрировать в систему автозапуска, поскольку части кода, имеющие отношение к сети нужно было чередовать с частями, к сети не относящимися. По этой причине скрипт `/etc/netstart` со временем мутировал в `/etc/rc.network`. Последний уже не является совершенно независимым скриптом, он состоит из больших, взаимозависимых функций, написанных на `sh`, которые, в свою очередь, вызываются из `/etc/rc` на разных этапах запуска системы. Тем временем, поскольку скрипты запуска стали больше, между ними появились сложные зависимости, "квази-модульная" система запуска стала еще запутаннее, чем монолитный скрипт `/etc/rc`.

Без простой и прозрачной подсистемы стартовые скрипты стали еще меньше удовлетворять потребности быстро развивающихся BSD-систем. Стало очевидно, что необходимы принципиальные шаги к достижению модульности и гибкости `rc` системы запуска. Исходя из этих соображений и была создана система запуска `BSD rc.d`. Ее признанными родителями стали Люк Мьюберн (Luke Mewburn) и сообщество NetBSD. Позже эта система была импортирована в FreeBSD. Ее название ссылается на расположение системных скриптов для отдельных сервисов, которые располагаются в `/etc/rc.d`. Немного позже мы рассмотрим отдельные компоненты системы [`rc.d`](#) и увидим, как запускаются отдельные скрипты.

Общая идея системы запуска `BSD rc.d` заключается в *высокой степени модульности и повторном использовании кода*. *Высокая степень модульности* значит, что каждый отдельный "сервис", такой как системный демон или простейшая задача, используют свой собственный скрипт, написанный на `sh`, который может запускать, останавливать, перезапускать этот сервис и проверять его статус. Стандартное действие выбирается из командной строки скрипта. `/etc/rc` так же как и раньше является основным загрузочным

скриптом, но теперь он не выполняет весь процесс загрузки, а вызывает отдельные скрипты с параметром *start*. Так же легко выполняется остановка системы - для этого запускается этот же набор скриптов, но с аргументом *stop*. Это действие выполняется управляющим скриптом */etc/rc.shutdown*. Обратите внимание, насколько точно эта система запуска соответствует традиционному в Unix принципу - использовать набор маленьких специализированных инструментов, каждый из которых призван решать конкретную задачу. *Повторное использование кода* - это подход, который значит, что часто используемые операции реализованы в виде функций на языке *sh* и объединены в файл */etc/rc.subr*. Теперь типичный скрипт может состоять всего из нескольких строк кода на языке *sh*. И наконец, очень важная часть подсистемы *rc.d* - программа *rcorder*, которая помогает скрипту */etc/rc* запускать отдельные скрипты в определенном порядке с учетом зависимостей между ними. В том числе, она так же помогает скрипту */etc/rc.shutdown*, поскольку правильный порядок завершения работы системы противоположен порядку при запуске.

Система запуска BSD *rc.d* более подробно описана в статье Люка Мьюберна, а устройство отдельных компонентов - в документации к компонентам на соответствующих страницах справочника. Тем не менее, эта документация не дает однозначного описания того, как новичок может используя небольшое количество простых элементов создать полноценный *rc.d* для специфической задачи. Поэтому, основная цель этой статьи - показать несколько примеров использования системы запуска *rc.d*. Мы разберем несколько наиболее часто встречающихся случаев и покажем как можно в этих случаях использовать *rc.d*. Обратите внимание, что этот документ не является пошаговым описанием ("HOWTO"), поскольку мы не ставим задачу дать готовые рецепты на все случаи жизни. Нашей задачей является дать вам точку входа для самостоятельного творчества в системе запуска *rc.d*. Так же, эта статья не в коей мере не является заменой страницам справочника. Не ленитесь заглядывать в них для более точной и полной информации во время чтения этой статьи.



Есть несколько требований для полного понимания изложенного материала. Во-первых, вы должны быть знакомы с программированием на скриптовом языке *sh*. Во-вторых, вы должны понимать, как система осуществляет стартовые и завершающие действия пространства пользователя, которые описаны в *rc*.

Эта статья делает основной упор на FreeBSD версию *rc.d*. Тем не менее, она так же может быть полезна разработчикам NetBSD. Несмотря на то, что системы запуска FreeBSD и NetBSD реализованы по разному, пользовательские интерфейсы предоставляемые ими практически идентичны.

## НАПИСАНИЕ СКРИПТОВ

### Обозначение задачи

Маленькое замечание перед началом. Не бойтесь переменной окружения \$EDITOR.

Для того, чтобы писать качественные *rc.d* скрипты для системных сервисов нам нужно ответить на следующие вопросы:

- Этот сервис обязательный или нет?
- Будет скрипт обслуживать только одну программу (т.е. демон) или будет выполнять более комплексные задачи?
- От каких сервисов будет зависеть этот сервис и какие будут зависеть от него?

Из следующих примеров мы можем увидеть, насколько важно знать ответы на эти вопросы перед написанием *rc.d* скриптов.

### Простейший скрипт

Этот скрипт просто выдает сообщение при каждом старте системы:

```
#!/bin/sh
. /etc/rc.subr
name="dummy"
start_cmd="${name}_start"
stop_cmd=":"
dummy_start()
{
    echo "Nothing started."
}
load_rc_config $name
run_rc_command "$1"
```

Здесь:

Скрипт должен начинаться с магической строки `#!/bin/sh` ("*shebang*"). Эта строка указывает командный интерпретатор для скрипта. Благодаря этой строке скрипт может быть вызван как любой другой исполняемый файл, если у него установлен бит исполнения (см. `chmod`). Например, системный администратор может запустить такой скрипт прямо из командной строки:

*#/etc/rc.d/dummy start*

### *Примечание*

Для того, чтобы подсистема [rc.d](#) могла правильно управлять этим скриптом, он должен быть написан на языке *sh*. Если ваш сервис или порт использует бинарное приложение для контроля запуска и остановки, установите это бинарное приложение в */usr/sbin* для системного сервиса или */usr/local/sbin* для портов, и вызывайте его из скрипта на *sh* в соответствующем каталоге.

Если вы хотите подробнее узнать почему *rc.d* скрипты должны быть написаны на языке *sh*, посмотрите как */etc/rc* вызывает их с помощью функции *run\_rc\_script*, а так же посмотрите, как устроена эта функция в файле */etc/rc.subr*.

В файле */etc/rc.subr* определено множество функций на языке *sh*, которые могут быть использованы в *rc.d* скриптах. Эти функции задокументированы в *rc.subr*. И хотя теоретически возможно написать *rc.d* скрипт вообще не пользуясь этими функциями, *rc.subr* функции настолько удобны и функциональны, что нет смысла усложнять себе жизнь не пользуясь ими. Поэтому, совершенно не удивительно, что все *rc.d* скрипты обращаются к *rc.subr*. И наш скрипт - не исключение.

Каждый *rc.d* должен "включать" в себя файл */etc/rc.subr* (здесь для этого используется команда *".")* *перед* тем как он вызовет функцию *rc.subr*. Наиболее предпочтительный вариант - сделать это перед любыми другими действиями.

### *Примечание*

Некоторые полезные функции, имеющие отношение к сети, находятся в еще одном важном файле */etc/network.subr*.

Обязательная переменная *name* определяет имя нашего скрипта, функции в *rc.subr* требуют наличия этой переменной. Поэтому каждый [rc.d](#) скрипт должен иметь установленную

переменную *name* до того, как он вызовет какую-либо из *rc.subr* функций.

Теперь самое подходящее время, чтобы выбрать уникальное имя раз и навсегда. Мы будем использовать это имя в нескольких местах в нашем скрипте. Для начала, дадим ему такое же имя, как и название нашего файла.

#### *Примечание*

В настоящее время при написании *rc.d* значения переменных заключаются в двойные кавычки. Помните, что это только стилистическая особенность и она не всегда применима. Вы можете опускать кавычки вокруг простых слов, не содержащих специальных символов *sh* или заключать значения в одиночные кавычки чтобы предотвратить преобразование значений интерпретатором *sh*. Программист должен отличать стилистические условности от потребностей языка и с пониманием использовать разные виды синтаксиса.

Главная идея [\*rc.subr\*](#) заключается в предоставлении *rc.d* методов вызова различных типичных действий. В частности, таким образом вызываются методы *start* и *stop* передаваемые скрипту как аргументы командной строки. Метод - это выражение на языке *sh* записанное как *argument\_cmd*, где *argument* является аргументом командной строки. Далее мы увидим как *rc.subr* предоставляет стандартные методы для типичных аргументов.

#### *Примечание*

Для стандартизации *rc.d* скриптов рекомендуется использовать переменную *\${name}* где это возможно. Благодаря этому, зачастую, основной работой в написании нового скрипта является копирование нескольких строк из уже существующего.

Нужно помнить, что *rc.subr* предоставляет стандартные методы для типичных аргументов. Поэтому, мы должны переопределить стандартный метод пустой операцией языка *sh* в случае, если мы хотим, чтобы наш скрипт ничего не делал.

Тело более сложного метода должно быть реализовано в виде функции. Хорошей практикой является осмысленное название таких функций.

### *Важно*

Настоятельно рекомендуется добавлять префикс `${name}` к именам всех функций определенных в нашем скрипте чтобы избежать переопределения функций из *rc.subr* или других включаемых файлов.

Эта команда *rc.subr* загружает переменные из файла *rc.conf*. Наш скрипт их пока не использует, но тем не менее, рекомендуется выполнять этот вызов в скрипте, так как в *rc.conf* могут содержаться переменные, управляющие непосредственно функциями *rc.subr*.

Обычно, это самая последняя команда в *rc.d* скрипте. Она непосредственно вызывает действия из *rc.subr* подготовленные методами и переменными, определенными в нашем скрипте.

## **Настраиваемый простейший скрипт**

Теперь давайте добавим несколько элементов управления в наш скрипт. Как вы знаете, *rc.d* скрипты управляются из файла *rc.conf*. К счастью, *rc.subr* скрывает от нас все сложные элементы разбора и анализа. Приведенный далее скрипт проверяет, было ли разрешено его выполнение в файле *rc.conf*, и если оно было разрешено, выводит сообщение при загрузке. Фактически, две эти задачи совершенно независимы. С одной стороны, *rc.d* скрипт поддерживает разрешение и запрещение его исполнения через *rc.conf*, с другой стороны *rc.d* может иметь конфигурационные переменные. Мы реализуем обе эти возможности в нашем скрипте:

```
#!/bin/sh
. /etc/rc.subr
name="dummy"
rcvar=`set_rcvar`
start_cmd="${name}_start"
stop_cmd=":"
load_rc_config $name
eval "${rcvar}=\${${rcvar}:-'NO'}"
dummy_msg=${dummy_msg:-"Nothing started."}
```

```
dummy_start()
{
    echo "$dummy_msg"
}
run_rc_command "$1"
```

Что изменилось в этом примере?

Переменная *rcvar* определяет значение переменной, которая должна разрешать или запрещать запуск скрипта. Необходимость получать имя переменной из *rc.subr* вызовом функции *set\_rcvar* обусловлена тем, что разные операционные системы используют разные соглашения в именовании переменных. Например, FreeBSD использует переменные вида *\${name}\_enable*, а [NetBSD](#) использует *\${name}* в файле *rc.conf*. Таким образом, наш скрипт будет руководствоваться значением переменной *dummy\_enable* под FreeBSD и значением переменной *dummy* под NetBSD.

Теперь *load\_rc\_config* вызывается до загрузки переменных из *rc.conf*

#### *Примечание*

При изучении *rc.d* скриптов, помните, что *sh* откладывает интерпретацию переменных до момента, когда они будут фактически вызваны, то есть, не является ошибкой вызвать *load\_rc\_config* позже, но до вызова *run\_rc\_command*.

Вызов функции *run\_rc\_command* выдаст предупреждение, если переменная *rcvar* определена, но для нее не выставлено значение. Если ваш *rc.d* скрипт предназначен для базовой системы, вы должны добавить значения по-умолчанию для него в файл */etc/defaults/rc.conf* и задокументировать их в *rc.conf*. В противном случае, вы должны определить значения по-умолчанию в самом скрипте. Переносимый вариант последнего случая показан в нашем примере.

#### *Примечание*

Вы можете заставить *rc.subr* действовать так, как будто значение управляющей переменной выставлено в ОН независимо от реального

значения добавляя к аргументу скрипта префиксы *one* или *force*, таким образом запуская скрипт с опциями *onestart* или *forcestop*. Помните, что *force* имеет ряд опасных побочных эффектов, которые мы обсудим позже, в то время как *one* просто переопределяет значение ON/OFF переменной. Другими словами, если предположить, что значение переменной *dummy\_enable* установлено в OFF, следующая команда выполнит метод *start* вопреки этому значению:

```
# /etc/rc.d/dummy onestart
```

Теперь сообщение, выводимое скриптом при старте, не является жестко прописанным в скрипте и неизменным. Оно может быть определено как переменная *dummy\_msg* в *rc.conf*. Это простейший пример того, как переменные *rc.conf* могут управлять поведением *rc.d* скрипта.

#### *Важно*

Все *rc.conf* переменные, используемые только нашим скриптом, должны иметь один и тот же префикс: *\${name}*. Например, *dummy\_mode*, *dummy\_state\_file*, и так далее.

#### *Примечание*

Хотя вполне возможно использовать внутри скрипта короткие имена переменных, то есть просто *msg*, тем не менее добавление уникального префикса *\${name}* ко всем глобальным переменным, используемым нашим скриптом предохранит нас от потенциальных переопределений переменных из пространства имен *rc.subr*.

До тех пор, пока переменные из *rc.conf* и их используемые в скрипте эквиваленты одинаковы, мы можем использовать более компактную запись для установки значения по-умолчанию:

```
: ${dummy_msg:="Nothing started."}
```

Хотя, в текущем соглашении о стиле все же рекомендуется использовать более развернутую форму записи.

Как правило, для *rc.d* скриптов базовой системы не устанавливают значения по-умолчанию в файле *rc.conf*, поскольку эти значения должны находиться в файле */etc/defaults/rc.conf*. С другой стороны, скрипты из системы портов должны устанавливать значения по-умолчанию сами, как это было сделано в нашем скрипте.

Здесь мы используем переменную *dummy\_msg* непосредственно для действия, то есть чтобы отобразить сообщение.



## ЗАПУСК И ОСТАНОВКА ДЕМОНОВ

### Запуск и остановка простого демона

Как уже упоминалось ранее, *rc.subr* может предоставить некоторые методы по-умолчанию. Очевидно, что такие стандартные методы не могут быть универсальными. Они предназначены в общем случае для запуска и остановки простого приложения-демона. Давайте предположим, что нам нужно написать *rc.d* скрипт для такого демона. Пусть демон называется *mumbled*. Вот этот скрипт:

```
#!/bin/sh
. /etc/rc.subr
name="mumbled"
rcvar=`set_rcvar`
command="/usr/sbin/${name}"
load_rc_config $name
run_rc_command "$1"
```

Необычайно просто, не так ли? Давайте посмотрим на него немного внимательнее. Отметить нужно следующее:

Переменная *command* очень важна для *rc.subr*. Если она установлена, то *rc.subr* будет действовать согласно сценарию обслуживания стандартного демона. В частности, стандартные методы предоставляются для аргументов: *start*, *stop*, *restart*, *poll* и *status*.

Демон будет запущен командой *\$command* с ключами определенными в переменной *\$mumbled\_flags*. Таким образом, все данные, необходимые для стандартного метода *start* определены в переменных на момент запуска скрипта. В отличие от метода *start*, другим методам может потребоваться дополнительная информация о запущенном процессе. Например, *stop* должен знать об идентификаторе запущенного процесса (PID) чтобы завершить его. В нашем случае, *rc.subr* найдет среди запущенных процессов процесс с именем *\$procname*. Эта переменная выставляется *rc.subr*

автоматически и по-умолчанию ее значение соответствует переменной *command*. Другими словами, когда мы указываем значение *command*, мы тем самым указываем и *procname*. Это позволяет нашему скрипту проверить, запущен ли демон или завершить его работу.

### *Примечание*

Некоторые программы на самом деле - запускаемые скрипты. Система запускает такие скрипты в командном интерпретаторе, передавая ему имя скрипта в качестве аргумента. Это отражается и на списке процессов, в том числе сбивает с толку *rc.subr*. Если вы запускаете такой скрипт, то вы должны указать интерпретатор как значение переменной *command\_interpreter*, чтобы *rc.subr* мог правильно найти его в списке процессов.

Для каждого *rc.d* скрипта есть необязательная переменная *rc.conf*, имеющая приоритет над переменной *command*. Она называется *\${name}\_program*, где *name* - обязательная переменная, которая обсуждалась ранее. Другими словами, в нашем случае переменная будет выглядеть как *tumbled\_program*. Эта переменная фактически переопределяет переменную *rc.subr command*.

Особенности языка *sh* позволяют указать значение переменной *\${name}\_program* в файле *rc.conf* или даже в самом скрипте и даже в том случае, если переменная *command* не установлена. В этом случае специальные свойства переменной *\${name}\_program* теряются и она становится обыкновенной переменной внутри скрипта и вы можете использовать ее в своих целях. При таком использовании теряется смысл переменной *\${name}\_program*, так как в *rc.d* скрипте переменные *\${name}\_program* и *command* при фактическом выполнении будут иметь одно значение.

Для более подробной информации смотрите страницу справочника *rc.subr*.

## Запуск более сложного демона

Давайте теперь нарастим немного мяса на костяк нашего предыдущего скрипта, усложним его и добавим возможностей. Методы по-умолчанию могут проделать отличную работу для нас, но иногда нужно их немного настроить. Сейчас мы научимся подстраивать стандартные методы под наши нужды.

```
#!/bin/sh

. /etc/rc.subr

name="mumbled"
rcvar=`set_rcvar`

command="/usr/sbin/${name}"
command_args="mock arguments > /dev/null 2>&1"

pidfile="/var/run/${name}.pid"

required_files="/etc/${name}.conf
/usr/share/misc/${name}.rules"

sig_reload="USR1"

start_precmd="${name}_prestart"
stop_postcmd="echo Bye-bye"

extra_commands="reload plugh xyzzy"

plugh_cmd="mumbled_plugh"
xyzzy_cmd="echo 'Nothing happens.'"

mumbled_prestart()
{
    if checkyesno mumbled_smart; then
        rc_flags="-o smart ${rc_flags}"
    fi
    case "$mumbled_mode" in
    foo)
        rc_flags="-frotz ${rc_flags}"
        ;;
    bar)

```

```

        rc_flags="-baz ${rc_flags}"
        ;;
*)
    warn "Invalid value for mumbled_mode"
    return 1
    ;;
esac
run_rc_command xyzzy
return 0
}

mumbled_plugh()
{
    echo 'A hollow voice says "plugh".'
}

load_rc_config $name
run_rc_command "$1"

```

Дополнительные аргументы команде *\$command* можно передать в переменную *command\_args*. Эти аргументы будут добавлены в командную строку после значения переменной *\$mumbled\_flags*. Поскольку командная строка, получившаяся в результате этих преобразований передается команде *eval*, то перенаправления ввода-вывода могут быть указаны в переменной *command\_args*.

#### *Примечание*

Никогда не указывайте опций, начинающихся со знака -, таких как -X или --foo в переменной *command\_args*. Значение переменной *command\_args* добавляется к концу формируемой командной строки, поэтому они будут добавлены после значений из *\${name}\_flags*. Но многие приложения не могут интерпретировать флаги после обычных аргументов. Лучшим вариантом будет добавить эти аргументы к переменной *\$command*, таким образом в конечной командной строке они будут добавлены перед значением переменной *\${name}\_flags*. Другой вариант - переопределить переменную *rc\_flags*, как это будет показано ниже.

Хорошо воспитанный демон должен создавать файл с идентификатором процесса (*pidfile*), чтобы в дальнейшем можно было

с легкостью найти запущенный процесс. Если переменная *pidfile* установлена, она указывает *rc.subr* где искать файл с идентификатором процесса для использования внутри своих методов.

#### *Примечание*

Фактически, *rc.subr* будет использовать эту переменную, и для того, чтобы проверить, запущен ли демон, перед тем как запустить его. Эта проверка может быть пропущена при использовании аргумента *faststart*.

Если демону для запуска требуются определенные файлы, просто перечислите их в переменной *required\_files*, и *rc.subr* будет проверять наличие этих файлов при старте. В том числе, существуют специальные переменные *required\_dirs* и *required\_vars* для каталогов и переменных окружения соответственно. Все эти переменные перечислены на странице справочника *rc.subr*.

#### *Примечание*

Стандартные методы *rc.subr* можно заставить не проводить такие проверки, для этого нужно использовать аргумент *forcestart* для скрипта.

В случае, если демон использует нестандартные сигналы, они так же могут быть определены в скрипте. В нашем случае переменная *sig\_reload* определяет сигнал, который заставит демона перечитать свою конфигурацию. По умолчанию, таким сигналом является *SIGHUP*. Другим сигналом, который можно переопределить, является *SIGTERM*, который используется для остановки демона, и этот сигнал определяется переменной *sig\_stop*.

#### *Примечание*

Названия сигналов должны быть переданы *rc.subr* без префикса *SIG*, как показано в примере. FreeBSD версия программы *kill* умеет распознавать префикс *SIG*, но версии других операционных систем могут этого не уметь.

Дополнительные действия, которые нужно выполнить до или после стандартных процедур можно определить с помощью переменных *argument\_precmd* и *argument\_postcmd*. Команды на

языке *sh*, определенные в этих переменных будут выполнены до или после соответствующих методов, в зависимости от названия переменной.

#### *Примечание*

Переопределение стандартных методов пользовательскими параметрами *argument cmd* не мешает использованию *argument\_precmd* или *argument\_postcmd* там, где они необходимы. В частности, в случаях, когда необходимы дополнительные проверки перед выполнением команды, использование *argument\_precmd* вместе с *argument\_cmd* позволяет логически разделить проверку и команду.

Не забывайте, что эти переменные должны содержать правильные выражения на языке *sh*. Зачастую, наиболее удобным способом является указание в этих переменных пользовательской функции, выполняющей всю работу. Это соответствует общей стилистике написания скриптов запуска, но не позволяйте ограничениям стиля ограничивать ваше понимание процесса в целом.

В случаях, когда необходимо определить дополнительные методы и аргументы кроме стандартных, их можно перечислить в переменной *extra commands*. Кроме того, нужно описать сами методы.

#### *Примечание*

Аргумент *reload* и его метод особые. С одной стороны, этот метод описан в *rc.subr*, с другой стороны, сам метод *reload* по-умолчанию не добавляется к списку методов скрипта. Причина этого кроется в том, что разные демоны используют различные механизмы перезагрузки, а некоторые вообще не имеют таких механизмов. Поэтому, если мы хотим выполнить определенные действия для перезагрузки демона, мы должны сделать это через переменную *extra\_commands*.

Так что же делает стандартный метод *reload*? Очень часто демоны используют сигнал SIGHUP для перечитывания своей конфигурации, поэтому *rc.subr* пытается перезагрузить демон посылая ему этот сигнал. Этот сигнал может быть переопределен в переменной *sig\_reload* если это необходимо.

Наш скрипт имеет два нестандартных аргумента *plugh* и *xuzzy*. Мы можем видеть их в [extra commands](#), и здесь мы предоставляем методы для этих аргументов. Метод *xuzzy* определен прямо как значение переменной, а метод *plugh* реализован в виде функции *tumbled\_plugh*.

Это нестандартные аргументы, и они не будут вызываться при запуске и остановке системы. Обычно, эти команды нужны для выполнения их администратором. В том числе, они могут использоваться сторонними подсистемами, например, *devd*, если они указаны в файле *devd.conf*.

Полный список доступных команд можно посмотреть, если вызвать наш скрипт без аргументов. Этот список будет автоматически сформирован *rc.subr*. Например, такой вывод мы увидим, вызвав наш скрипт без аргументов:

```
# /etc/rc.d/mumbled
```

```
Usage:
```

```
/etc/rc.d/mumbled
```

```
[fast|force|one](start|stop|restart|rcvar|reload|plugh|xuzzy|status|poll)
```

Скрипт может вызывать свои собственные стандартные или нестандартные команды, если это необходимо. Это может выглядеть как вызов функций в теле скрипта, но нужно помнить, что вызов функции командной оболочки и вызов скрипта - это не одно и то же. Например, команда *xuzzy* в нашем случае не реализована в виде функции. Кроме того, могут быть команда с префиксами *pre-* и *post-* необходимые для выполнения функции. Поэтому, правильный путь - запускать команды так, как они представляются *rc.subr*, как это и показано в примере.

*rc.subr* предоставляет еще одну очень полезную функцию *checkyesno*. Она принимает в качестве аргумента имя переменной и возвращает нулевой код возврата, если значение переменной установлено в YES, TRUE, ON или 1 не зависимо от регистра. В противном случае, функция возвращает ненулевой код возврата. В

последнем случае, функция так же проверяет, имеет ли переменная значение NO, FALSE, OFF или 0 не зависимо от регистра, и если переменная содержит что-то кроме этих значений (мусор), выводит предупреждение.

Помните, что в языке *sh* нулевой код возврата соответствует значению *истина*, а ненулевой код соответствует значению *ложь*.

*Важно*

Функция *checkyesno* принимает в качестве значения *имя переменной*. Не передавайте ей развернутое значение переменной; это приведет к совершенно неожиданным результатам.

Пример правильного использования функции *checkyesno*:

```
if checkyesno mumbled_enable; then
    foo
fi
```

И наоборот, вызов функции *checkyesno* как показано ниже не будет работать, или, по крайней мере, будет работать не так, как вы этого ожидаете:

```
if checkyesno "${mumbled_enable}"; then
    foo
fi
```

Мы можем поменять флаги, передаваемые *\$command* изменяя переменную *rc\_flags* в *\$start\_precmd*.

В некоторых случаях бывает необходимо отобразить важные сообщения и в то же время передать их в *syslog*. Это с легкостью можно проделать с помощью функций *debug*, *info*, *warn*, и *err*, определенных в *rc.subr*. Функция *err* после своего вызова завершает скрипт с указанным кодом возврата.

Коды возврата методов и их pre- команд по умолчанию не игнорируются. Если *argument\_precmd* возвращает ненулевой код,



главный метод не будет выполнен. Так же и *argument\_postcmd* не будет выполнен, если главный метод вернет ненулевой код.

#### *Примечание*

Тем не менее, в командной строке можно дать указание *rc.subr* игнорировать коды ошибок всех функций и выполнять все команды. Для этого нужно добавить к аргументу скрипта в качестве префикса *force*, например *forcestart*.

## ПРИСОЕДИНЕНИЕ RC.D СКРИПТА К ИНФРАСТРУКТУРЕ

После того, как скрипт был написан, его необходимо присоединить к *rc.d* инфраструктуре. Решающим шагом является установка скрипта в */etc/rc.d* (для базовой системы) или в */usr/local/etc/rc.d* (для портов). В файлах *<bsd.prog.mk>* и *<bsd.port.mk>* существуют специальные процедуры для этого, и вам не придется заботиться о правильных правах доступа и владельце скриптов. Системные скрипты должны устанавливаться из директории *src/etc/rc.d* и должны быть указаны для этого в *Makefile*, находящемся в этой директории. Скрипты из портов должны устанавливаться с помощью переменной *USE\_RC\_SUBR*, как это описано в руководстве по созданию портов.

Но перед установкой мы должны указать место нашего скрипта в загрузочной последовательности. Сервисы, управляемые нашими скриптами почти наверняка зависят от других сервисов. Например, любой сетевой демон не сможет работать без активных сетевых интерфейсов, роутинга и тому подобного. Даже если кажется, что сервис ни от чего не зависит, ему, по крайней мере, требуется смонтированная файловая система.

Мы уже упоминали об *rcorder*. Сейчас настал момент, взглянуть на эту программу поближе. В двух словах, *rcorder* принимает в качестве аргументов список файлов, проверяет их содержимое и выводит упорядоченный список в *stdout*. Идея держать информацию о зависимостях *внутри* заключается в том, чтобы каждый файл имел структурную зависимость только от себя. Внутри файла можно указать следующую информацию: названия "условий" (в нашем случае - сервисов), которые наш скрипт *предоставляет*;

- названия "условий", которые *необходимы* скрипту для запуска;
- названия "условий", которые должны быть выполнены *до* запуска нашего сервиса;
- дополнительные *ключевые слова* (keywords), чтобы выделить группу файлов из всего набора. (можно дать

команду *rcorder* игнорировать или использовать только те файлы, в которых есть определенное ключевое слово.)

В нормальном случае *rcorder* может оперировать только с текстовыми файлами, синтаксис которых похож на синтаксис *sh*. Поэтому, специальные строки, обрабатываемые *rcorder* выглядят как комментарии в языке *sh*. Синтаксис этих строк очень жесткий, чтобы упростить их обработку. Смотрите *rcorder* для более детальной информации.

Кроме того, использование специальных строк позволяет скрипту настаивать на зависимости от другого сервиса с принудительным стартом последнего. Это может быть необходимым в случае, когда сервис-зависимость является необязательным или не запускается, потому что администратор по ошибке отключил его в *rc.conf*.

Обладая этими знаниями, давайте улучшим наш скрипт для запуска простого демона, добавив в него зависимости:

```
#!/bin/sh

# PROVIDE: mumbled oldmumble
# REQUIRE: DAEMON cleanvar frotz
# BEFORE: LOGIN
# KEYWORD: nojail shutdown

. /etc/rc.subr

name="mumbled"
rcvar=`set_rcvar`
command="/usr/sbin/${name}"
start_precmd="${name}_prestart"

mumbled_prestart()
{
    if ! checkyesno frotz_enable && \
        ! /etc/rc.d/frotz forcestatus 1>/dev/null
2>&1; then
        force_depend frotz || return 1
    fi
    return 0
}
```

```
load_rc_config $name  
run_rc_command "$1"
```

А теперь, как и прежде, проведем детальный анализ изменений:

Эта строка описывает "условия", предоставляемые нашим скриптом. Теперь, другие скрипты могут перечислять в списке "условий" "условия", предоставляемые нашим скриптом, тем самым они станут зависимы от нашего скрипта.

#### *Примечание*

Обычно, скрипт может предоставлять только одно условие, тем не менее, ничто не мешает нам указать несколько условий. Например, для совместимости с предыдущими версиями.

В любом случае, название первого из этих условий (или единственного) должно быть таким же, как значение переменной `${name}`.

Кроме всего прочего, наш скрипт определяет "условия", от которых он зависит. Следуя этим строкам *rcorder* запустит наш скрипт после скриптов, предоставляющих условия *DAEMON* и *cleanvar*, но до скрипта предоставляющего условие *LOGIN*.

#### *Примечание*

Строка *BEFORE*: не формирует полный список зависимостей. Правильный способ ее использования - если другой скрипт может корректно запуститься без нашего скрипта, но наш скрипт может помочь ему сделать это лучше. Типичный пример из жизни - сетевые интерфейсы и фаерволы. Интерфейс на самом деле не зависит от фаервола, но из соображений безопасности системы иногда необходимо, чтобы фаервол был инициализирован до появления какого бы то ни было сетевого трафика зависимостей. Правильный способ ее использования - если другой скрипт может корректно запуститься без нашего скрипта, но наш скрипт может помочь ему сделать это лучше. Типичный пример из жизни - сетевые интерфейсы и фаерволы. Интерфейс на самом деле не зависит от фаервола, но из соображений безопасности системы иногда необходимо, чтобы фаервол был инициализирован до появления какого бы то ни было сетевого трафика.

Кроме условий, соответствующих отдельному сервису, существуют мета-условия и их "контейнеры". Они описывают группы операций. Их можно отличить по написанию их названий в верхнем регистре. Список контейнеров и их назначение можно найти на странице справочника *rc*.

Помните, что размещение сервиса в строке REQUIRE: не гарантирует, что этот сервис будет действительно запущен до запуска нашего сервиса. Требуемый сервис может быть отключен в *rc.conf* или просто не сумеет запуститься из-за ошибки. Очевидно, что *rcorder* не способен отследить такие тонкости, так же этого не может проконтролировать и *rcorder*. Тем не менее, наш скрипт должен уметь проконтролировать подобную ситуацию. Как это можно сделать в некоторых случаях описано ниже.

Как мы помним, ключевые слова *rcorder* могут использоваться для запуска групп скриптов или исключения таких групп из запуска. А именно, используя опции *rcorder -k* и *-s* мы можем указывать список ключевых слов для групп, которые нужно выполнить (keep list) или которые нужно пропустить (skip list) соответственно. Из всех файлов, которые необходимо отсортировать в порядке исполнения *rcorder* выберет только те, в которых указано ключевое слово из списка выполнения и не указано ключевое слово из списка пропуска.

В FreeBSD, [\*rcorder\*](#) используется в скрипте */etc/rc* и скрипте */etc/rc.shutdown*. Эти два скрипта определяют стандартный для FreeBSD список ключевых слов и их значений.

Например:

*nojail*

Сервис не для запуска в *jail*. Автоматические процедуры запуска и остановки системы будут игнорировать такие скрипты, если они запускаются изнутри *jail*.

*nostart*

Сервис не должен запускаться вручную или не должен запускаться вообще. Автоматическая процедура запуска будет игнорировать такие скрипты. В сочетании с ключевым словом *shutdown* это может использоваться для скриптов, которые должны делать что-либо только при завершении системы.

### *Shutdown*

Это ключевое слово должно использоваться *только* для сервисов, которые нужно остановить перед остановкой всей системы.

### *Примечание*

Когда система завершает работу, запускается скрипт `/etc/rc.shutdown`. Этот скрипт подразумевает, что большая часть `rc.d` скриптов ничего не делает в этот момент. Поэтому, `/etc/rc.shutdown` не пытается запустить `rc.d` скрипты, а запускает только те из них, в которых содержится ключевое слово `shutdown` и полностью игнорирует все остальные. Для еще более быстрого завершения работы системы `/etc/rc.shutdown` завершает все скрипты с аргументом `faststop`, то есть, скрипты пропускают некоторые стандартные проверки, такие как проверку на наличие файла с идентификатором процесса (`pidfile`). Так как зависимые сервисы должны быть завершены раньше, чем те от которых они зависят, то `/etc/rc.shutdown` запускает скрипты в обратном порядке.

Если вы пишете [`rc.d`](#) скрипт, вы должны знать как он будет вести себя в момент завершения работы системы. Другими словами, если ваш скрипт выполняет свои действия по команде `start`, то вам не нужно указывать в нем ключевое слово `shutdown`. Если же ваш скрипт управляет работой сервиса, хорошей идеей будет остановить его перед фактическим завершением работы системы, описанным в `halt`. В частности такой подход необходим к сервисам, которым необходимо время для корректного завершения. Типичный пример такого сервиса - база данных.

Для начала, помните, что функция `force_depend` должна использоваться с большой осторожностью. В общем случае, лучше будет изменить иерархию конфигурационных переменных в ваших `rc.d` скриптах, если они независимы.

Если же вы никак не можете обойтись без функции `force_depend`, в нашем примере представлен типичный пример, как это сделать корректным образом. Нашему демону `tumbled` дополнительно нужен для работы сервис `frotz`. Тем не менее сервис `frotz` - вспомогательный. К счастью, наш скрипт имеет доступ ко всем переменным `rc.conf`. Если переменная `frotz_enable` установлена и ее значение - истина, мы напомним на лучшее и полностью полагаемся на `rc.d` в запуске сервиса `frotz`. В противном случае, мы форсированно проверяем статус сервиса `frotz`, и если сервис не запущен, мы запускаем сервис `frotz` функцией `force_depend`. Предупреждающее сообщение будет показано при вызове этой функции так как этот вызов произойдет только в случае ошибки конфигурации.

## СОЗДАНИЕ БОЛЕЕ ГИБКИХ *rc.d* СКРИПТОВ

Во время запуска и остановки системы *rc.d* скрипт предположительно оперирует над той или иной подсистемой целиком. К примеру, скрипт */etc/rc.d/netif* должен запускать и останавливать все сетевые интерфейсы, описанные в *rc.conf*. Так же каждая задача должна быть целиком описана одной командой - *start* или *stop*. Но между запуском системы и ее остановкой *rc.d* должны помогать системному администратору управлять запущенными процессами, и зачастую для этого необходимо больше гибкости. Например, администратор может сконфигурировать новый интерфейс в *rc.conf* и ему понадобится запустить только этот интерфейс, не затрагивая все остальные. В следующий раз, администратору может понадобиться остановить этот единственный интерфейс. Для этого можно передавать дополнительные опции для скрипта, такие как название интерфейса, в виде дополнительных аргументов командной строки.

К счастью, *rc.subr* позволяет передать скрипту любое количество аргументов (в пределах системных ограничений). Поэтому, можно обойтись минимальными изменениями скрипта.

Как *rc.subr* может получить доступ к аргументам командной строки? Передать их непосредственно внутри скрипта функции *run\_rc\_command* невозможно, так как функции *sh* не имеют доступа к позиционным параметрам вызывающего их объекта. Кроме того, в *rc.d* скриптах считается хорошим тоном, когда скрипт сам определяет какие аргументы должны быть переданы его методам.

В *rc.subr* используется следующее решение: функции *run\_rc\_command* передаются все аргументы командной строки, при этом подразумевается, что первым из этих аргументов является имя запускаемого метода - *start*, *stop* и т.д. Эти аргументы сдвигаются внутри функции *run\_rc\_command* с помощью оператора *sh shift*. Таким образом, переменная *\$2* в изначальной командной строке передается методу как *\$1*, и так далее.



Хорошо проиллюстрировать это нам поможет следующий пример. Давайте изменим наш первый простейший скрипт так, чтобы его сообщения зависели от дополнительных аргументов. Итак:

```
#!/bin/sh

. /etc/rc.subr

name="dummy"
start_cmd="${name}_start"
stop_cmd=":"
kiss_cmd="${name}_kiss"
extra_commands="kiss"

dummy_start()
{
    if [ $# -gt 0 ]; then
        echo "Greeting message: $"
    else
        echo "Nothing started."
    fi
}

dummy_kiss()
{
    echo -n "A ghost gives you a kiss"
    if [ $# -gt 0 ]; then
        echo -n " and whispers: $"
    fi
    case "$*" in
        *.[!?] )
            echo
            ;;
        *)
            echo .
            ;;
    esac
}

load_rc_config $name
run_rc_command "$@"
```

Какие значимые изменения были сделаны в этом файле?

Все аргументы, переданные скрипту после аргумента *start* будут переданы как аргументы соответствующему методу. Мы можем использовать их любым способом в соответствии с поставленной задачей и в меру наших навыков и фантазии. В настоящем примере мы просто передаем их все команде *echo* как одну строку в переменной *\$\** внутри двойных кавычек. Вот пример как может быть вызван такой скрипт:

```
# /etc/rc.d/dummy start
Nothing started.
```

```
# /etc/rc.d/dummy start Hello world!
Greeting message: Hello world!
```

Такой же трюк можно проделать с любыми методом нашего скрипта, а не только со стандартными. Добавим нестандартный метод *kiss*, и он будет иметь те же возможности, что и метод *start*:

```
# /etc/rc.d/dummy kiss
A ghost gives you a kiss.
```

```
# /etc/rc.d/dummy kiss Once I was Etaoin Shrdlu...
A ghost gives you a kiss and whispers: Once I was Etaoin Shrdlu...
```

Если мы хотим просто передать все дополнительные аргументы любому методу, мы можем просто заменить "\$1" на "\$@" в последней строке нашего скрипта, в которой мы вызываем *run\_rc\_command*.

### *Важно*

Программисты на языке *sh* обязаны понимать тонкую разницу между специальными переменными *\$\** и *\$@* и тем, как они передают

позиционные параметры. Для более тщательного изучения этого вопроса ознакомьтесь с очень подробной страницей справочника *sh*. *Не используйте* эти переменные, если вы не понимаете их действие, поскольку их неправильное использование может сделать ваш скрипт неправильно работающим и небезопасным.

#### *Примечание*

В настоящее время функция *run\_rc\_command* имеет ошибку, которая может привести к потере разделителей между аргументами. Поэтому, аргументы, в которых присутствует пробел могут обрабатываться некорректно. Появление этой ошибки может быть спровоцировано неправильным использованием переменной *\$\**.

## ЗАДАНИЕ НА ЛАБОРАТОРНУЮ РАБОТУ

Сконфигурировать систему исходя из заданной вам схемы сети (сетевые интерфейсы, маршрутизация, DNS). Продемонстрировать работу команд. Выполнить следующие шаги:

1. Ознакомиться с предложенным материалом для получения информации об управлении учетными записями в ОС FreeBSD
2. Создать простейший сценарий.
3. Настроить простейший сценарий.
4. Создать простейший демон
5. Запустить простейший демон.
6. Создать более сложный демон.
7. Присоединить сценарий к инфраструктуре.
8. Написать сценарий исходя из заданной схемы сети.
9. Проверить работоспособность написанного скрипта.
10. Сделать скрипт более гибким.

Ответить на контрольные вопросы и подготовить отчет.

## КОНТРОЛЬНЫЕ ВОПРОСЫ И ЗАДАНИЯ

1. Объясните, что такое `rs.d`.
2. Объясните, что такое сценарий.
3. Объясните, что такое демон.
4. Раскройте область применения демонов.
5. Перечислите команды для работы со сценариями.
6. Раскройте суть аргументов сценария.
7. Опишите назначение `sh`.
8. Раскройте смысл `PID`.
9. Объясните, зачем нужен `Makefile`.
10. Перечислите преимущества гибких скриптов.

## **ФОРМА ОТЧЕТА ПО ЛАБОРАТОРНОЙ РАБОТЕ**

На выполнение лабораторной работы отводится 2 занятия (4 академических часа: 3 часа на выполнение и сдачу лабораторной работы и 1 час на подготовку отчета).

Отчет на защиту предоставляется в печатном виде.

Структура отчета (на отдельном листе(-ах)): титульный лист, формулировка задания, ответы на контрольные вопросы, описание процесса выполнения лабораторной работы, выводы.

## ОСНОВНАЯ ЛИТЕРАТУРА

1. Вирт, Н. Разработка операционной системы и компилятора. Проект Оберон [Электронный ресурс] / Н. Вирт, Ю. Гуткнехт. — Москва: ДМК Пресс, 2012. 560 с. Режим доступа: <https://e.lanbook.com/book/39992>
2. Войтов, Н.М. Основы работы с Linux. Учебный курс [Электронный ресурс]: учебное пособие / Н.М. Войтов. — Москва : ДМК Пресс, 2010. — 216 с. — Режим доступа: URL: <https://e.lanbook.com/book/1198>
3. Стащук, П.В. Краткое введение в операционные системы [Электронный ресурс] : учебное пособие / П.В. Стащук. — 3-е изд., стер. — Москва : ФЛИНТА, 2019. — 124 с.— URL: <https://e.lanbook.com/book/125385>

## ДОПОЛНИТЕЛЬНАЯ ЛИТЕРАТУРА

4. Войтов, Н.М. Администрирование ОС Red Hat Enterprise Linux. Учебный курс [Электронный ресурс] : учеб. пособие — Москва: ДМК Пресс, 2011. 192 с. Режим доступа: <https://e.lanbook.com/book/1081>
5. Стащук П.В. Администрирование и безопасность рабочих станций под управлением Mandriva Linux: лабораторный практикум. [Электронный ресурс]: учебно-методическое пособие / П.В. Стащук. — 2-е изд., стер. - М: Флинта, 2015. <https://e.lanbook.com/book/70397>

### Электронные ресурсы:

1. Научная электронная библиотека <http://eLIBRARY.RU>.
2. Электронно-библиотечная система <http://e.lanbook.com>.
3. Электронно-библиотечная система «Университетская библиотека онлайн» <http://biblioclub.ru>.
4. Электронно-библиотечная система IPRBook <http://www.iprbookshop.ru/>
5. Losst - Linux Open Source Software Technologies <https://losst.ru>