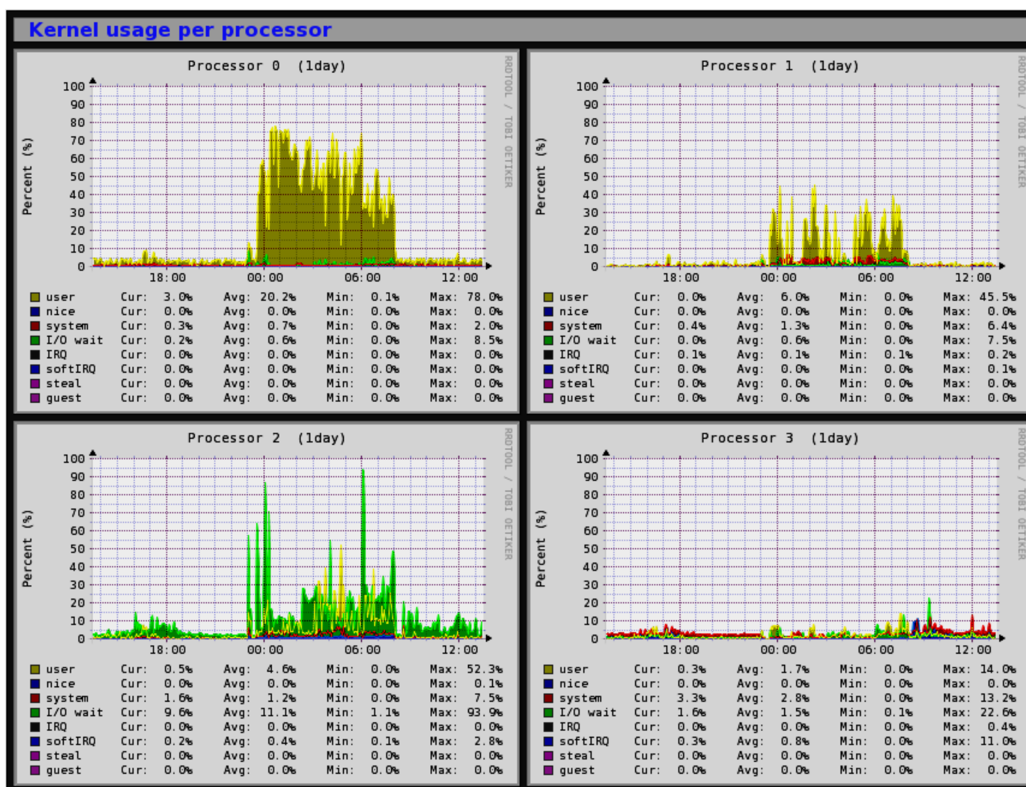
УНИВЕРСИТЕТ ИТМО

Д.А. Зубок, А.М. Кашевник, А.В. Маятин

ОПЕРАЦИОННЫЕ СИСТЕМЫ. ЛАБОРАТОРНЫЙ ПРАКТИКУМ



Санкт-Петербург
2021

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

УНИВЕРСИТЕТ ИТМО

Д.А. Зубок, А.М. Кашевник, А.В. Маятин

ОПЕРАЦИОННЫЕ СИСТЕМЫ. ЛАБОРАТОРНЫЙ ПРАКТИКУМ

УЧЕБНО-МЕТОДИЧЕСКОЕ ПОСОБИЕ

РЕКОМЕНДОВАНО К ИСПОЛЬЗОВАНИЮ В УНИВЕРСИТЕТЕ ИТМО

по направлению подготовки 09.03.02 Информационные системы и технологии в качестве учебно-методического пособия для реализации основных профессиональных образовательных программ высшего образования бакалавриата

Санкт-Петербург
2021

Зубок Д.А., Кашевник А.М., Маятин А.В., Операционные системы. Лабораторный практикум– СПб: Университет ИТМО, 2021. – 36 с.

Рецензент(ы):
Береснев Артем Дмитриевич, старший преподаватель (квалификационная категория "старший преподаватель") факультета инфокоммуникационных технологий, Университета ИТМО.

Учебно-методическое пособие содержит теоретический минимум, методические указания и задания для выполнения лабораторных работ в виртуальных машинах под управлением операционных систем семейств GNU/Linux и Microsoft Windows. Раскрываются вопросы управления процессами, организации межпроцессного взаимодействия, управления памятью, управления файлово-каталожной системой, автоматизации администрирования.

Университет ИТМО – ведущий вуз России в области информационных и фотонных технологий, один из немногих российских вузов, получивших в 2009 году статус национального исследовательского университета. С 2013 года Университет ИТМО – участник программы повышения конкурентоспособности российских университетов среди ведущих мировых научно-образовательных центров, известной как проект «5 в 100». Цель Университета ИТМО – становление исследовательского университета мирового уровня, предпринимательского по типу, ориентированного на интернационализацию всех направлений деятельности.

© Университет ИТМО, 2021
© Зубок Д.А., Кашевник А.М., Маятин А.В., 2021

Содержание

Введение..4

Лабораторная работа №1. Основы использования консольного интерфейса ОС Linux и интерпретатора bash 7

Лабораторная работа №2. Мониторинг процессов и ресурсов в ОС Linux........ 15

Лабораторная работа №3. Управление процессами в ОС Linux.......................... 19

Лабораторная работа №4. Работа с файлово-каталожной системой в ОС Linux.. 26

Лабораторная работа №5. Управление памятью в ОС Linux 30

Лабораторная работа №6. Консольный интерфейс ОС Microsoft Windows 34

Рекомендуемая литература 36

Введение

Операционная система является важным компонентом ИТ-инфраструктуры, компонентом, от которого критически зависит реализация нефункциональных требований – производительности, надежности и безопасности – любой информационной системы. В основе архитектуры распределенных систем, облачных решений находятся специализированные и универсальные операционные системы. Сервера и рабочие станции, мобильные устройства и активное телекоммуникационное оборудование, практически любой вычислительный узел работают под управлением операционной системы. Основная задача операционной системы – обеспечить эффективное распределение ресурсов вычислительного узла между прикладными программами, предоставить пользователю интерфейс для взаимодействия с приложениями, обеспечить управляемый доступ к данным, реализовать сетевое взаимодействие.

Универсальные операционные системы, используемые на бытовом уровне – в мобильных телефонах, на домашних компьютерах и планшетах – как правило, реализуют принципы автоматического управления и невмешательства пользователя в решение задачи распределения ресурсов, делая акцент на предоставлении удобного графического интерфейса для доступа к приложениям и пользовательским данным. Но в случае специализированных, серверных операционных систем, высокие требования к производительности, надежности и безопасности, сложность и изменчивость контекста, в котором необходимо принимать решения по управлению ресурсами, возможные внешние воздействия на вычислительный узел и его прикладное программное обеспечение приводят к необходимости регулярной деятельности по администрированию операционной системы, мониторингу работы прикладного программного обеспечения и в целом вычислительного узла. Для решения этих задач необходимо не только уметь использовать интерфейсы управления приложениями и встроенными утилитами операционной системы, но, прежде всего, требуется понимание принципов работы отдельных механизмов и в целом операционной системы.

Предлагаемое учебно-методическое пособие направлено на реализацию лабораторного практикума из шести лабораторных работ. Пособие является дополненной и переработанной редакцией пособия «Операционные системы. Методические указания по выполнению лабораторных работ», вышедшего в 2015 году [1]. В новой редакции изменена структура и содержание лабораторных работ, внесены изменения в задания и методические рекомендации в соответствии с развитием предметной области и с учетом опыта реализации дисциплины. Первые пять лабораторных работ выполняются в операционной системе GNU/Linux CentOS. Выбор этой операционной системы обусловлен тем, что она является unix-based операционной системой, а именно такие операционные системы наиболее распространены в серверных вычислительных узлах. Для unix-based

операционных систем характерен в целом схожий подход к управлению с помощью интерфейса командного интерпретатора и автоматизация задач управления с помощью скриптов, а также отображение всей информации о состоянии и параметрах объектов операционной системы в виде текстовых файлов. Поэтому в рамках первой лабораторной работы предлагается освоить синтаксис и основные конструкции скриптового языка bash – основного инструмента автоматизации администрирования в unix-based операционных системах, а также научиться анализировать и обрабатывать текстовые потоки и текстовые файлы с помощью регулярных выражений и специальных утилит: grep, sed, awk. Эти навыки необходимы для выполнения следующих заданий, связанных уже непосредственно с управлением объектами операционной системы. Вторая и третья лабораторные работы посвящены основным объектам управления операционной системы – процессам. Целью этих лабораторных работ является формирование навыков мониторинга параметров процессов и выделяемых им ресурсов, управления приоритетами исполнения процессов, планирования запуска процессов, а также использования механизмов обмена данными между процессами и межпроцессного взаимодействия, в частности, механизма сигналов. Четвертая лабораторная работа посвящена управлению файлами и каталогами в unix-based операционных системах. В рамках её выполнения необходимо разработать средства безопасного удаления и резервного копирования данных. Пятая лабораторная работа предполагает переход от управления объектами операционной системы в соответствии с заранее определенными рекомендациями к организации исследования для глубокого понимания принципов и механизмов управления. В качестве объекта для исследования выбран механизм контроля и выделения оперативной памяти процессам. Результатом выполнения всех перечисленных лабораторных работ является множество разнообразных скриптов, большинство из которых применимо в реальной практике серверного администрирования.

Последняя, шестая лабораторная работа выполняется в операционной системе семейства Microsoft Windows. Несмотря на традиционное использование, в том числе для администрирования, в этом семействе операционных систем графического пользовательского интерфейса, целью этой лабораторной работы является формирование навыков автоматизации задач администрирования с помощью написания управляющих скриптов для встроенного интерпретатора cmd или PowerShell. В качестве объектов управления, кроме традиционных для операционных систем процессов и файлов, в заданиях лабораторной работы потребуется реализовать управление также службами и драйверами.

В процессе выполнения лабораторных работ необходимо иметь полные права доступа к управлению ресурсам и изменениям настроек операционной системы, но при этом иметь возможность оперативно восстанавливать работоспособность операционной системы после естественных на этапе обучения ошибок. Эти

требования предопределили выбор основного технического средства проведения лабораторных работ – виртуальной машины с развернутым в ней дистрибутивом операционной системы.

Первые пять лабораторных работ могут выполняться практически в любом дистрибутиве операционной системы GNU/Linux, но методические рекомендации и примеры приводятся для дистрибутива CentOS. CentOS (Community ENTerprise Operating System) – это дистрибутив Linux, версии которого основаны на исходном коде коммерческого дистрибутива Red Hat Enterprise Linux, являющегося корпоративным стандартом де-факто и имеющего хорошую документацию. Для выполнения лабораторных работ можно использовать готовую конфигурацию виртуальной машины, подготовленную преподавателем, или самостоятельно создать виртуальную машину и установить операционную систему Linux CentOS, скачав дистрибутив с сайта http://www.centos.org/.

В качестве средства виртуализации при выполнении лабораторных работ рекомендуется использовать решение Oracle VM VirtualBox. Это решение поддерживает большой набор операционных систем, включая операционные системы семейства GNU/Linux. В свою очередь, сама среда виртуализации может быть развернута на хостовых компьютерах под управлением большого набора операционных систем, в том числе ОС семейств Microsoft Windows, GNU/Linux и Mac OS. Благодаря тому, что Oracle VM VirtualBox распространяется бесплатно, имеется возможность во время самостоятельной работы использовать ту же конфигурацию, что и во время аудиторных занятий. Виртуальная машина распространяется в виде готового к развертыванию архива в формате ova. Для работы необходимо выделить виртуальной машине не менее 1Gb оперативной памяти (рекомендуется 2 Gb).

Дополнительные преимущества при выполнения лабораторных работ, требующих использования привилегированной учетной записи, дает возможность использования снимков виртуальных машин. Этот механизм позволяет зафиксировать состояние виртуальной машины в любой момент времени и, в случае сбоя, вернуться к нему. Oracle VM VirtualBox позволяет строить деревья снимков, фиксируя тем самым различные достигнутые состояния виртуальной машины. Снимок может быть сделан как для остановленной виртуальной машины, так и с виртуальной машины, находящейся в состоянии исполнения.

Лабораторная работа №1. Основы использования консольного интерфейса ОС Linux и интерпретатора bash

Рассматриваемые вопросы:

1. Использование встроенной документации на команды и утилиты интерпретатора bash
2. Запуск и работа в консольном текстовом редакторе
3. Создание и запуск скриптов для интерпретатора bash
4. Стандартный ввод и стандартный вывод процесса, перенаправление стандартного вывода
5. Построение конвейеров команд
6. Подстановка результатов вывода процесса в качестве параметра другого процесса
7. Фильтрация текстовых потоков с помощью регулярных выражений

Для доступа к подробной **справочной информации** по любой команде или утилите можно использовать утилиту `man` (от англ. manual), набрав в консоли «man название команды», для краткой справки – можно ввести название_команды --help.

Например, *man man* выведет справочное руководство по команде man, *man bash* – справочное руководство по интерпретатору bash.

Shell-скрипт – это обычный текстовый файл, содержащий последовательность команд, которую можно было бы ввести в командной строке. Команды, записанные в скрипте, выполняются командным интерпретатором – шеллом (shell). В Linux- и в целом в Unix-системах для того, чтобы бинарный файл или скрипт смогли быть исполнены, пользователю, который запускает такой файл, должны быть предоставлены соответствующие права на выполнение. Назначить право на исполнение файла можно с помощью утилиты `chmod u+x имя_скрипта`. В первой строке скрипта принято записывать специальную конструкцию – шебанг, обозначающую путь к интерпретатору: `#!/bin/bash`. Для того, чтобы набирать и редактировать текст скрипта, можно использовать текстовые редакторы `nano` или `vi`. Редактор можно запустить, набрав его название в командной строке. В качестве параметра можно передать имя редактируемого файла.

Рассмотрим основные синтаксические конструкции языка bash.

Комментарии. Последовательность символов, начинающаяся с символа # (за исключением шебанг (`#!`)), является комментарием и игнорируется

интерпретатором при исполнении скрипта. Комментарием может быть как целая строка, так и часть строки, начинающаяся с #.

Экранирование символов и подстрок. Для предотвращения интерпретации специальных символов могут быть использованы одиночные и двойные кавычки. Одиночные кавычки (' '), ограничивающие подстроку с двух сторон, предотвращают интерпретацию всех специальных символов. Двойные кавычки (" ") предотвращают интерпретацию специальных символов, за исключением $, ` (обратная кавычка) и \ (escape – обратный слэш). Рекомендуется использовать двойные кавычки при обращении к переменным. При необходимости вывести отдельный специальный символ можно также использовать экранирование: символ обратного слэша (\) предотвращает интерпретацию следующего за ним символа.

Переменные. Имена переменных в bash аналогичны традиционному представлению об идентификаторе: в качестве имени переменной может использоваться последовательность букв, цифр и нижнего подчеркивания, начинающаяся с буквы или нижнего подчеркивания. Встречая в тексте сценария имя переменной, интерпретатор подставляет вместо него значение этой переменной. Поэтому ссылки на переменные называются подстановкой переменных. Например, `variable1` – это имя переменной, а `$variable1` – это ссылка на ее значение. Имена переменных, без префикса $, могут использоваться только при объявлении переменной или для присваивания переменной некоторого значения. В отличие от множества других языков программирования, bash не поддерживает типы данных для переменных. Фактически все переменные bash являются строковыми переменными, но, в зависимости от контекста, определяемого используемыми констуркциями, bash позволяет реализовать целочисленную арифметику с переменными.

Оператор присваивания "=". При использовании оператора присваивания нельзя ставить пробелы слева и справа от знака равенства. Если необходимо присвоить переменной результат выполнения арифметические операции, перед ним добавляют оператор `let`, например:

```
let a=2\*2
```

(обратите внимание, что оператор умножения является специальным символом и, следовательно, должен быть экранирован).

Арифметические операторы:

"+" сложение
"−" вычитание

"*" умножение
"/" деление (целочисленное)
"**" возведение в степень
"%" остаток от целочисленного деления

Зарезервированные имена переменных. В bash существует ряд зарезервированных имен переменных, которые хранят специальные значения.

– Параметры вызова скрипта. Аргументы, передаваемые скрипту из командной строки, хранятся в зарезервированных переменных `$0, $1, $2, $3...`, где `$0` – это название файла сценария, `$1` – это первый аргумент, `$2` – второй, `$3` – третий и так далее. Аргументы, номер которых состоит более чем из одной цифры, должны заключаться в фигурные скобки, например: `${10}, ${11}, ${12}.` Передача параметров скрипту происходит в виде перечисления этих параметров после имени скрипта через пробел в момент его запуска.
– Другие зарезервированные переменные:

`$DIRSTACK` – содержимое вершины стека каталогов
`$UID` – идентификатор пользователя.
`$HOME` – домашний каталог пользователя
`$HOSTNAME` – hostname компьютера
`$HOSTTYPE` – архитектура машины.
`$PWD` – рабочий каталог
`$OSTYPE` – тип ОС
`$PATH` – путь поиска программ
`$PPID` – идентификатор родительского процесса
`$SECONDS` – время работы скрипта (в секундах)
`$#` – общее количество параметров, переданных скрипту
`$*` – все аргументы, переданные скрипту (выводятся в строку)
`$@` – то же самое, что и предыдущий, но параметры выводятся в столбик
`$!` – PID последнего запущенного в фоне процесса
`$$` – PID самого скрипта

Код завершения. Так же как в программах на языке С и многих других языках, для завершения работы сценария может использоваться команда `exit`. Эта команда может возвращать числовое значение, которое может быть проанализировано вызывающим процессом – код возврата. Команде `exit` можно указать код возврата через пробел, в виде `exit nnn`, где `nnn` – это число в диапазоне 0– 255.

Оператор вывода. `Echo переменные_или_константы`

Оператор ввода. `Read имя_переменной`. Одной командой `read` можно прочитать (присвоить) значения сразу для нескольких переменных. При этом, если переменных в `read` будет указано больше, чем их введено (через пробелы), оставшимся переменным присваивается в качестве значений пустая строка. Если же передаваемых значений будет больше, чем переменных, указанных в команде `read`, лишние значения игнорируются.

Условный оператор.
`If команда; then команда; [else команда]; fi.`
Если команда вернет в результате исполнения значение "истина", то выполняется команда после `then`. Условный оператор не может сам осуществлять операции сравнения переменных, поэтому, если в качестве условия необходимо сравнивать значения переменных и/или констант, после `if` используется специальная конструкция `[[ выражение ]]`. Фактически это оператор, которому передаются параметры, поэтому обязательно ставить пробелы между скобками и выражением, например:

```
if [[ "$a" -eq "$b" ]]
then echo "a = b"
fi
```

Операторы сравнения:

Для строк

`-z` # является ли строка пустой
`-n` # является ли строка непустой
`=, (==)` # равенство строк
`!=` # неравенство строк
`<` # лексикографически меньше
`>` # лексикографически больше

Для числовых значений
`-eq` # равно
`-ne` # не равно
`-lt` # меньше
`-le` # меньше или равно
`-gt` # больше
`-ge` # больше или равно

`!` # отрицание логического выражения
`-a, (&&)` # логическое «И»
`-o, (||)` # логическое «ИЛИ»

Множественный выбор. В качестве оператора множественного выбора может использоваться оператор `case`.

```
case переменная in
значение1 )
набор команд 1
 ;;
значение2 )
набор команд 2
 ;;
esac
```

Выбираемые значения выделяются правой скобкой в конце значения. Разделитель вариантов – `;;`

Цикл for. Для записи цикла `for` можно использовать две синтаксические конструкции:

1. Стандартная – `for переменная in список_значений; do; команды; done`. Например:

```
for i in 0 1 2 3
do
echo $i
done
```

2. С-подобная

```
for ((i=0; c <=3; i++))
do
echo $i
done
```

Цикл while. Синтаксис цикла с проверкой условия: `while условие; do; команда; done`. Требования к записи условия такие же, как и в условном операторе.

Управление циклами. Аналогично другим языкам программирования, для управления ходом выполнения цикла можно использовать команды `break` и `continue`. Команда `break` прерывает исполнение цикла, а `continue` возвращает управление в начало цикла, минуя все последующие команды в теле цикла.

Управление вводом-выводом команд (процессов)

Для любого процесса по умолчанию всегда открыты три дескриптора файла: 0, 1 и 2 – **`stdin`** (стандартный ввод, клавиатура), **`stdout`** (стандартный вывод, экран) и **`stderr`** (стандартный вывод сообщений об ошибках, экран), соответственно. Эти и любые другие открытые процессом файлы могут быть перенаправлены.

Перенаправление означает возможность получить вывод из одного команды (программы, сценария) и передать его на вход в другую команду (программу, сценарий).

`команда > файл` – перенаправление в файл стандартного вывода, содержимое существующего файла замещается.
`команда >> файл` – перенаправление в файл стандартного вывода, при котором выводящийся поток дописывается в конец файла.
`команда1 | команда2` – перенаправление стандартного вывода левой команды на стандартный ввод правой команды. Такая конструкция называется конвейером команд.
`команда1 $(команда2)` – подстановка вывода команды 2 в качестве параметров запуска команды 1. Конструкция **`$(команда2)`** может использоваться, например, для передачи результатов работы команды 2 в параметры цикла **`for … in`.**

Работа со строками (встроенные команды bash)
`${#string}` – выводит длину строки (**`string`** – имя переменной);
`${string:position:length}` – извлекает **`$length`** символов из **`$string`**, начиная с позиции **`$position`**. Частный случай: **`${string:position}`** извлекает подстроку из **`$string`**, начиная с позиции **`$position`**.
`${string#substring}` – удаляет самую короткую из найденных подстрок **`$substring`** в строке **`$string`**. Поиск ведется с начала строки. **`$substring`** – может быть регулярным выражением (см. ниже).
`${string##substring}` – удаляет самую длинную из найденных подстрок **`$substring`** в строке **`$string`.** Поиск ведется с начала строки. **`$substring`** – также может быть регулярным выражением.
`${string/substring/replacement}` – замещает первое вхождение **`$substring`** строкой **`$replacement`**. **`$substring`** – может быть регулярным выражением.
`${string//substring/replacement}` – замещает все вхождения **`$substring`** строкой **`$replacement`**. **`$substring`** – может быть регулярным выражением.

Работа со строками (внешние команды - утилиты)
Для каждой утилиты доступно собственное управление с помощью передаваемых команде параметров, в том числе ключей. Рекомендуем ознакомиться с документацией по этим командам с помощью команды man.
`sort` – сортирует строки файла в порядке убывания или возрастания, в зависимости от заданных опций.
`uniq` – удаляет повторяющиеся строки из отсортированного файла, в том числе с возможностью подсчета повторений.

cut – выводит отдельные поля из текстовых файлов (поле – последовательность символов в строке до разделителя).
head – выводит начальные строки из файла.
tail – выводит последние строки из файла.
wc – подсчитывает количество слов/строк/символов в файле или в потоке
tr – заменяет одни символы на другие.
Полнофункциональные многоцелевые утилиты:
grep – многоцелевая поисковая утилита, использующая регулярные выражения.
sed – неинтерактивный "потоковый редактор". Принимает текст из входного потока, выполняет некоторые операции над строками и затем выводит результат в выходной поток. **Sed** определяет, над какими строками следует выполнить операции по заданному адресному пространству строк. Адресное пространство строк задается либо их порядковыми номерами, либо с помощью шаблона. Например, команда **3d** заставит **sed** удалить третью строку, а команда **/text/d** означает, что должны быть удалены все строки, содержащие "**text**". Наиболее часто используются команды **p** – вывод (в **stdout**), **d** – удаление и **s** – замена.
awk – многофункциональная утилита контекстного поиска и преобразования текста. Может быть использована как инструмент для извлечения и/или обработки полей в структурированных текстовых файлах. **Awk** разбивает каждую строку на отдельные поля. По умолчанию поля – это последовательности символов, отделенные друг от друга пробелами или табуляцией, однако имеется возможность назначения других символов в качестве разделителя полей. **Awk** использует свой встроенный язык программирования для определения алгоритма работы с переменными, образованными выделенными из строки полями.

Регулярные выражения – это набор символов и/или метасимволов, которые соответствуют некоторому множеству подобных строк.

Регулярные выражения предназначены для поиска фрагментов текста по шаблону и работы со строками. При построении регулярных выражений используются следующие конструкции (в порядке убывания приоритета), некоторые из которых могут быть применены только в расширенных версиях соответствующих команд (например, при запуске **grep** с ключом -**E**).

c Любой неспециальный символ **c** соответствует самому себе
\c Указание убрать любое специальное значение символа **c** (экранирование)
^ Начало строки
$** Конец строки; выражение **"^$" соответствует пустой строке.
. Любой одиночный символ, за исключением символа перевода строки

[...] Любой символ из **...;** допустимы диапазоны типа **a-z;** возможно объединение диапазонов, например **[a-z0-9]**
[^...] Любой символ не из **...;** допустимы диапазоны
r* Ноль или более вхождений символа **r** (может применяться и для диапазонов)
r+ Одно или более вхождений символа **r** (может применяться и для диапазонов)
r? Ноль или одно вхождение символа **r** (может применяться и для диапазонов)
\<...\> Границы слова
\{ \} Число вхождений предыдущего выражения. Например, выражение **"[0-9]\{5\}"** соответствует подстроке из пяти десятичных цифр
r1r2 За **r1** следует **r2**
r1|r2 **r1** или **r2**
(r) Регулярное выражение **r;** может быть вложенным

Классы символов POSIX
Конструкция **[:class:]** может быть использована как альтернативный способ указания диапазона символов.
[:alnum:] соответствует символам латинского алфавита и цифрам. Эквивалентно выражению [**A-Za-z0-9**].
[:alpha:] соответствует символам латинского алфавита. Эквивалентно выражению [**A-Za-z**].
[:blank:] соответствует символу пробела или символу табуляции.
[:digit:] соответствует набору десятичных цифр. Эквивалентно выражению [**0-9**].
[:lower:] соответствует набору символов латинского алфавита в нижнем регистре. Эквивалентно выражению [**a-z**].
[:space:] соответствует пробельным символам (пробел и горизонтальная табуляция).
[:upper:] соответствует набору символов латинского алфавита в верхнем регистре. Эквивалентно выражению [**A-Z**].
[:xdigit:] соответствует набору шестнадцатиричных цифр. Эквивалентно выражению [**0-9A-Fa-f**].

Задание на лабораторную работу
1. Создайте каталог для выполнения лабораторной работы в директории /home/user/ Все скрипты создавайте внутри этого каталога или его подкаталогов. (`mkdir lab1`)
2. Напишите скрипты, решающие следующие задачи:

i) В параметрах при запуске скрипта передаются три целых числа. Вывести максимальное из них.
ii) Считывать строки с клавиатуры, пока не будет введена строка "q". После этого вывести последовательность считанных строк в виде одной строки.
iii) Создать текстовое меню с четырьмя пунктами. При вводе пользователем номера пункта меню происходит запуск редактора nano, редактора vi, браузера links или выход из меню.
iv) Если скрипт запущен из домашнего каталога, вывести на экран путь к домашнему каталогу и завершить работу с кодом 0. В противном случае вывести сообщение об ошибке и завершить работу с кодом 1.
v) Создать файл **`info.log`**, в который поместить все строки из файла /**`var/log/anaconda/syslog`**, второе поле в которых равно **`INFO`**.
vi) Создать **`full.log`**, в который вывести строки файла /**`var/log/anaconda/X.log`**, содержащие предупреждения и информационные сообщения, заменив маркеры предупреждений и информационных сообщений на слова **`Warning`**: и **`Information:`** так, чтобы в получившемся файле сначала шли все предупреждения, а потом все информационные сообщения. Вывести этот файл на экран.
vii) Создать файл **`emails.lst`**, в который вывести через запятую все адреса электронной почты, встречающиеся во всех файлах каталога /**`etc`**.
viii) Вывести список пользователей системы с указанием их UID, отсортировав по UID. Сведения о пользователей хранятся в файле /**`etc/passwd`**. В каждой строке этого файла первое поле – имя пользователя, третье поле – UID. Разделитель полей – двоеточие.
ix) Подсчитать общее количество строк в файлах, находящихся в директории /**`var/log`**/ и имеющих расширение **`log`**.
x) Вывести три наиболее часто встречающихся слова из **`man`** по команде **`bash`** длиной не менее четырех символов.

3. Предъявите скрипты преподавателю и получите вопрос или задание для защиты лабораторной работы.

Лабораторная работа №2. Мониторинг процессов и ресурсов в ОС Linux

Рассматриваемые вопросы

1. Получение информации о запущенных процессах
2. Получение информации об используемых процессами ресурсах
3. Представление информации в удобном для анализа и обработки виде

Идентификация процессов

Система идентифицирует процессы по уникальному номеру, называемому идентификатором процесса и обозначаемому **PID** (process ID).

Все процессы, работающие в системе GNU/Linux, организованы в виде дерева. Корнем этого дерева является процесс **init** или процесс **systemd** – процессы системного уровня, запускаемые во время загрузки. Для каждого процесса хранится идентификатор его родительского процесса (**PPID,** Parent Process ID). Для корневого процесса в качестве идентификатора родительского процесса указывается 0.

Получение общих сведений о запущенных процессах

Команда **ps** (сокращение от process status)

Запуск **ps** без аргументов покажет только те процессы, которые были запущены вызывающим команду пользователем и привязаны к текущему терминалу.

Часто используемые параметры (в отличие от ключей, указываются без "-"):
a – вывод процессов, запущенных всеми пользователями;
x – вывод процессов, не привязанных к управляющему терминалу или привязанных к другим управляющим терминалам;
u – вывод для каждого из процессов имя запустившего его пользователя и времени запуска.
Обозначения состояний процессов (в колонке **STAT**)
R – процесс выполняется в данный момент или находится в состоянии готовности;
S – процесс ожидает события (прерываемое ожидание);
D – процесс ожидает ввода/вывода (непрырываемое ожидание);
I – простаивающий поток ядра;
Z – zombie-процесс;
T – процесс остановлен.

Команда **pstree**

Команда **pstree** выводит процессы в виде дерева, наглядно демонстрируя наследование процессов при их создании.

Часто используемые параметры:
-p – вывести **PID** всех процессов;
-c – развернуть все ветви;
-u – вывести имена пользователей, запустивших процессы.

Команда **top**

top – утилита, используемая для наблюдения за процессами в режиме реального времени. Является интерактивной и управляется с клавиатуры. Для получения справки можно нажать **h**. Часто используемые команды для мониторинга процессов:

M – используется для сортировки процессов по объему занятой ими памяти (поле **%MEM**);
P –используется для сортировки процессов по занятому ими процессорному времени (поле **%CPU**). Это метод сортировки по умолчанию;
U – эта команда используется для вывода процессов заданного пользователя. Необходимо ввести имя пользователя, а не его UID. Если не ввести никакого имени, будут показаны все процессы;
i – по умолчанию **top** выводит все процессы, даже спящие. Команда **i** обеспечивает вывод информации только о работающих в данный момент процессах (процессы, у которых поле **STAT** имеет значение **R**, Running). Повторное использование этой команды возвращает к списку всех процессов.

Получение детальных сведений о запущенных процессах

/proc – псевдо-файловая система, которая реализует интерфейс к структурам данных в ядре операционной системы. Большинство отображаемых в ней файлов доступны только для чтения, но в некоторые файлы возможна запись, что позволяет изменять переменные ядра.

Каждому запущенному процессу в **/proc** соответствует подкаталог с именем, совпадающим с идентификатором этого процесса (его **PID**). Каждый из этих подкаталогов содержит следующие файлы-интерфейсы, в том числе в подкаталогах (указаны наиболее часто использующиеся файлы). Часть из этих файлов доступна только в каталогах процессов, запущенных от имени текущего пользователя, или при обращении от имени суперпользователя **root**.

cmdline – полная командная строка запуска процесса.
cwd – ссылка на текущий рабочий каталог процесса.
environ – окружение процесса. Записи в файле разделяются нулевыми символами, и в конце файла также может быть нулевой символ.
exe – символьная ссылка, содержащая фактическое полное имя исполняемого файла процесса.
fd – подкаталог с файловыми дескрипторами, содержащий по одной символической ссылке на каждый файл, который в данный момент открыт процессом. Имя каждой ссылки соответствует номеру файлового дескриптора.

Так, **0** – это стандартный ввод, **1** – стандартный вывод, **2** – стандартный вывод ошибок и т. д.
io – файл, содержащий сведения об объемах данных, прочитанных и записанных процессом в хранилище.
maps – файл, содержащий адреса областей памяти, которые используются программой в данный момент, и права доступа к ним.
sched – файл, содержащий значения переменных для каждого процесса, использующихся планировщиком **CFS** для принятия решения о выделении процессу процессорного времени. Например, **sum_exec_runtime** – это переменная, содержащая оценку суммарного времени выполнения процесса, а **nr_switches** – переменная, хранящая количество переключений контекста.
stat – машиночитаемая информация о процессе в виде набора полей;
status – предоставляет значительную часть информации из **stat** в более удобном для прочтения формате.
statm – предоставляет информацию о состоянии памяти, используя в качестве единиц измерения страницы. Список полей в файле:

size	общий размер программы
resident	размер резидентной части
shared	количество разделяемых страниц
text	текст (код)
lib	библиотеки (не используется, начиная с ядра 2.6)
data	данные + стек
dt	"грязные" (dirty) страницы (не используется, начиная с ядра 2.6)

Обработка данных о процессах

Обработка данных о процессах осуществляется, как правило, с помощью конвейера команд обработки текстовых потоков и (или) через циклическую обработку строк файлов. Рекомендуется применять команды, изученные в рамках первой лабораторной работы – **grep, sed, awk, tr, sort, uniq, wc, paste**, а также функции для работы со строками.

Задание на лабораторную работу

1. Создайте каталог для выполнения лабораторной работы в директории /**home/user/** Все скрипты и файлы для вывода результатов создавайте внутри этого каталога или его подкаталогов. (**mkdir lab2**)
2. Напишите скрипты, решающие следующие задачи:
   i) Подсчитать количество процессов, запущенных пользователем **user**, и вывести в файл получившееся число, а затем пары **PID:команда** для таких процессов.

ii) Вывести в файл список **PID** всех процессов, которые были запущены командами, расположенными в каталоге **/sbin/**
iii) Вывести на экран **PID** процесса, запущенного последним (с последним временем запуска).
iv) Для всех зарегистрированных в данный момент в системе процессов определить среднее время непрерывного использования процессора (CPU_burst) и вывести в один файл строки
ProcessID=PID : Parent_ProcessID=PPID : Average_Running_Time=ART.
Значения **PPid** взять из файлов **status**, которые находятся в директориях с названиями, соответствующими **PID** процессов в **/proc**. Значения **ART** получить, разделив значение **sum_exec_runtime** на **nr_switches**, взятые из файлов **sched** в этих же директориях. Отсортировать эти строки по идентификаторам родительских процессов.
v) В полученном на предыдущем шаге файле после каждой группы записей с одинаковым идентификатором родительского процесса вставить строку вида
Average_Running_Children_of_ParentID=N is M,
где **N** = **PPID**, а **M** – среднее, посчитанное из **ART** для всех процессов этого родителя.
vi) Используя псевдофайловую систему **/proc**, найти процесс, которому выделено больше всего оперативной памяти. Сравнить результат с выводом команды **top**.
vii) Написать скрипт, определяющий три процесса, которые за 1 минуту, прошедшую с момента запуска скрипта, считали максимальное количество байт из устройства хранения данных. Скрипт должен выводить **PID**, строки запуска и объем считанных данных, разделенные двоеточием.

3. Предъявите скрипты преподавателю и получите вопрос или задание для защиты лабораторной работы.

Лабораторная работа №3. Управление процессами в ОС Linux

Рассматриваемые вопросы

1. Директивы объединения команд
2. Планирование времени запуска процессов
3. Управление приоритетами процессов
4. Организация межпроцессного взаимодействия

Директивы (команды) объединения команд

Командный интерпретатор **`bash`** поддерживает следующие директивы объединения команд:

`команда1 | команда2` – конвейер: перенаправление стандартного вывода,
`команда1 ; команда2` – последовательное выполнение команд,
`команда1 && команда2` – выполнение команды 2 при успешном завершении команды 1,
`команда1 || команда2` – выполнение команды 2 при неудачном завершении команды 1,
`команда1 $(команда2)` – подстановка вывода команды 2 в качестве аргументов (параметров) запуска команды 1,
`команда 1 > файл` – направление стандартного вывода в файл (содержимое существующего файла замещается),
`команда 1 >> файл` – направление стандартного вывода в файл (поток дописывается в конец файла).

```
{
команда1
команда2
}
```

– объединение команд до или после директив **`||`**, **`&&`** или в теле циклов и функций.

`команда1 &` – запуск команды в фоновом режиме (стандартный вход отрывается от консоли, из которой запускается процесс; управление процессом возможно в общем случае только с помощью сигналов).

Утилиты для управления процессами

(с подробным описанием функционала и синтаксисом утилит можно ознакомиться в документации, доступной по команде **`man название_утилиты`**)

`kill` – отправляет сигнал процессу. Передаваемый сигнал может указываться в параметре утилиты в виде его номера или символьного обозначения. По умолчанию (без указания сигнала) утилита передает сигнал требования о завершении процесса (**`TERM`**). Идентификация процесса, которому передает сигнал утилита **`kill`**, производится по PID. Перечень системных сигналов, доступных в GNU/Linux с указанием их номеров и символьных обозначений, можно получить с помощью параметра утилиты **`kill -l`**;
`killall` – аналогично утилите kill посылает сигнал, но для идентификации процесса использует его символьное имя, а не PID;
`pidof` – определяет PID процесса по его символьному имени;
`pgrep` – определяет PID процессов с заданными характеристиками, например, запущенные конкретным пользователем;
`pkill` – отправляет сигнал группе процессов с заданными характеристиками;

nice – запускает процесс с заданным значением приоритета. Уменьшение значения (повышение приоритета выполнения) относительно базового приоритета может быть инициировано только пользователем root;
renice – изменяет значения приоритета для запущенного процесса. Уменьшение значения (повышение приоритета выполнения) может быть инициировано только пользователем root;
at – осуществляет однократный запуск команды в указанное время.
cron – демон, который занимается запуском команд по определенным датам и в определенное время. Команды, выполняемые периодически, указываются с использованием команды **crontab**. Для однократного запуска команд используют **at.** Синтаксис строки в **crontab** подробно описан здесь: http://www.opennet.ru/man.shtml?topic=crontab&category=5&russian=2.
tail – не только выводит последние n строк из файла, но и может быть использована для "слежения" за файлом. В этом режиме утилита отслеживает добавление новых строк в файл и выводит строки, появляющиеся в конце файла.
sleep – задает паузу между командами при выполнении скрипта.

Организация взаимодействия двух процессов

Существует несколько различных способов организации взаимодействия процессов. Поскольку цель взаимодействия состоит в передаче данных или управления от одного процесса к другому, рассмотрим два распространенных варианта организации взаимодействия процессов: передачу данных через именованный канал и передачу управления через сигнал.

Взаимодействие процессов через именованный канал

Именованный канал – специальный тип файла в Linux. Создается командой **mkfifo имя_файла**. Взаимодействие с именованным каналом происходит обычными средствами для взаимодействия с файлами, но при этом такой файл не будет сохраняться на носителе, а представляет собой буфер в памяти для организации межпроцессного обмена данными.

Для демонстрации передачи информации через именованный канал рассмотрим два скрипта – «Генератор» и «Обработчик». Пусть требуется считывать информацию с одной консоли с помощью процесса «Генератор» и затем выводить ее на экран другой консоли с помощью процесса «Обработчик», причем таким образом, чтобы считывание генератором строки «QUIT» приводило к завершению работы обработчика. Каждый процесс запускается отдельным скриптом независимо в своей виртуальной консоли. Переключаясь между консолями, можно управлять процессами и наблюдать результаты их работы.

Перед запуском скриптов создадим именованный канал с помощью команды **mkfifo pipe**

<table>
<tr><th>Генератор</th><th>Обработчик</th></tr>
<tr><td><pre>#!/bin/bash
while true; do
    read LINE
    echo $LINE > pipe
done</pre></td><td><pre>#!/bin/bash
(tail -f pipe) |
while true; do
    read LINE;
    case $LINE in
        QUIT)
            echo "exit"
            killall tail
            exit
            ;;
        *)
            echo $LINE
            ;;
    esac
done</pre></td></tr>
</table>

Скрипт «Генератор» в бесконечном цикле считывает строки с консоли и записывает их в именованный канал **pipe**.

Скрипт «Обработчик» рассмотрим подробнее.

Обычно команда **tail** используется для считывания последних **n** строк из файла. Но одним из распространенных вариантов ее использования является организация «слежения» за файлом. При использовании конструкции **tail -f** будет происходить считывание только новых строк, добавляемых в файл. Для того, чтобы передавть выход команды **tail** на вход скрипта «Обработчик», используем конструкцию **(команда)|** Оператор «круглые скобки» позволяет запустить подпроцесс (дочерний процесс) внутри родительского процесса «Обработчик», а оператор конвейера в конце позволяет направить выход этого подпроцесса на вход родительского процесса. Таким образом, команда **read** в этом скрипте читает строки на выходе команды **tail.** Остальная часть скрипта использует конструкции, изученные в предыдущих лабораторных работах, и не требует дополнительного комментирования. Исключение составляет только

команда **killall tail.** С ее помощью завершается созданный в виде подпроцесса процесс **tail** перед завершением родительского процесса. Использование **killall** в этом случае используется для упрощения кода, но, строго говоря, не является корректным, поскольку сигнал будет послан всем процессам с таким именем в системе. Правильнее определять PID конкретного процесса **tail,** вызванного в скрипте, и завершать его с помощью команды **kill.**

Взаимодействие процессов с помощью сигналов

Сигналы являются основной формой передачи управления от одного процесса к другому. В Linux используются 64 различных сигнала. Любой процесс при создании наследует от родительского процесса таблицу указателей на обработчики сигналов и, тем самым, готов принять любой из 64-х сигналов. Обработчики по умолчанию, как правило, приводят к завершению процесса или, реже, игнорированию сигнала, но, за исключением двух сигналов (**KILL** и **STOP**), обработчики могут быть заменены на другие. У большинства сигналов есть предопределенное назначение и ситуации, в которых они будут передаваться, поэтому изменение их обработчиков возможно, но рекомендуется только в рамках расширения функционала соответствующего обработчика. Такие сигналы считаются системными. Но есть два сигнала (**USR1** и **USR2**), для которых предполагается, что обработчики будут создаваться разработчиком приложения и использоваться для передачи и обработки пользовательских, неспецифицированных сигналов (по умолчанию их обработка предполагает завершение процесса). Для замены обработчиков сигналов в **sh** (**bash**) используется встроенная команда **trap** с форматом **trap action signal**

При вызове команды указываются два параметра: обработчик, который должен быть запущен при получении сигнала, и собственно символьное имя сигнала, для которого будет выполняться указанный обработчик. Обычно в качестве обработчика указывают вызов функции, описанной выше в коде скрипта.

С помощью команды **trap** можно задать обработчик не только для пользовательского сигнала, но и подменить обработчик для практически всех из системных сигналов (кроме тех, переопределение обработчика которых запрещено). В этом случае при получении процессом системного сигнала обработка сигнала перейдет к указанному в **trap** обработчику.

Для демонстрации передачи управления от одного процесса к другому с помощью сигнала рассмотрим еще одну пару скриптов.

Генератор	Обработчик

`#!/bin/bash`
`while true; do`
`    read LINE`
`    case $LINE in`
`        STOP)`
`            kill -USR1 $(cat .pid)`
`                ;;`
`        *)`
`                :`
`                ;;`
`    esac`
`done`
	`#!/bin/bash`
	`echo $$ > .pid`
	`A=1`
	`MODE="rabota"`
	`usr1()`
	`{`
	`     MODE="ostanov"`
	`}`
	`trap 'usr1' USR1`
	`while true; do`
	`     case $MODE in`
	`          "rabota")`
	`               let A=$A+1`
	`               echo $A`
	`               ;;`
	`          "ostanov")`
	`               echo "Stopped by`
	`SIGUSR1"`
	`               exit`
	`               ;;`
	`     esac`
	`     sleep 1`
	`done`

В этом примере скрипт «Генератор» в бесконечном цикле считывает строки с консоли и бездействует (используется оператор `:`) для любой входной строки, кроме строки STOP. Получив эту строку, скрипт отправит пользовательский сигнал **`USR1`** процессу «Обработчик». Поскольку процесс «Генератор» должен указать PID процесса «Обработчик», передача этого идентификационного номера осуществляется с помощью скрытого файла. В скрипте «Обработчик» определение PID процесса производится с помощью зарезервированной системной переменной **`$$.`**

Процесс «Обработчик» выводит на экран строки с натуральными числами до момента получения сигнала **`USR1`**. При получении сигнала запускается обработчик сигнала **`usr1()`**, который меняет значение переменной MODE. В результате на следующем шаге цикла выводится сообщение о прекращении работы в связи с получением сигнала, и работа скрипта будет завершаться.

Задание на лабораторную работу

Создайте скрипты или запишите последовательности выполнения команд для перечисленных заданий и предъявите их преподавателю.

1. Создайте и однократно выполните скрипт, который не содержит условный оператор и операторы проверки свойств и значений, но позволяет выполнить следующие задачи: пытается создать каталог **`test`** в домашнем каталоге. Если создание каталога пройдет успешно, скрипт выведет в файл **`~/report`** сообщение вида "**`catalog test was created successfully`**" и создаст в каталоге **`test`** файл с именем **`Дата_Время_Запуска_Скрипта`**. Затем независимо от результатов предыдущего шага скрипт опросит с помощью команды **`ping`** хост www.net_nikogo.ru и, если этот хост недоступен, допишет сообщение об ошибке в файл **`~/report.`** Сообщение об ошибке должно начинаться с текущей **`Дата_Время`**, а затем содержать через пробел произвольный текст сообщения об ошибке.
2. Задайте еще один однократный запуск скрипта из пункта 1 через 2 минуты. Консоль после этого должна оставаться свободной. Выполнив отдельную команду, организуйте слежение за файлом **`~/report`** и выведите на консоль новые строки из этого файла, как только они появятся.
3. Задайте запуск скрипта из пункта 1 в каждую пятую минуту каждого часа в день недели, в который вы будете выполнять работу.
4. Создайте три фоновых процесса, выполняющих одинаковый бесконечный цикл вычисления (например, перемножение двух чисел). После запуска процессов должна сохраниться возможность использовать виртуальную консоль, с которой их запустили. Используя команду **`top`**, проанализируйте процент использования ресурсов процессора этими процессами. Создайте скрипт, который будет в автоматическом режиме обеспечивать, чтобы тот процесс, который был запущен первым, использовал ресурс процессора не более чем на 10%. Послав сигнал, завершите работу процесса, запущенного третьим. Проверьте, что созданный скрипт по-прежнему удерживает потребление ресурсов процессора первым процессом в заданном диапазоне.
5. Создайте пару скриптов: генератор и обработчик. Процесс «Генератор» должен передавать информацию процессу «Обработчик» с помощью именованного канала. Процесс «Обработчик» должен осуществлять следующую обработку считываемых из именованного канала строк: если строка состоит из единственного символа «**+**», то процесс обработчика переключает режим на «сложение» и ждет ввода численных данных. Если строка состоит из единственного символа «*****», то обработчик переключает режим на «умножение» и ждет ввода численных данных. Если строка

содержит целое число, то обработчик выполняет текущую активную операцию (в соответствии с выбранным режимом) над текущим значением обрабатываемой переменной и считанным значением (например, складывает или перемножает результат предыдущего вычисления со считанным числом). При запуске скрипта режим устанавливается в «сложение», а вычисляемая переменная инициализируется значением 1. В случае получения строки **`QUIT`** скрипт «Обработчик» выводит в консоль сообщение о плановой остановке, и оба скрипта завершают работу. В случае получения любых других значений строки оба скрипта завершают работу с сообщением об ошибке входных данных.

6. Создайте пару скриптов: генератор и обработчик. Процесс «Генератор» в бесконечном цикле считывает с консоли строки. Если считанная строка состоит из единственного символа «**+**», он посылает процессу «Обработчик» сигнал **`USR1`**. Если строка состоит из единственного символа «*****», генератор посылает обработчику сигнал **`USR2`**. Если строка содержит слово **`TERM`**, генератор посылает обработчику сигнал **`SIGTERM`** и завершает свою работу. Другие значения входных строк игнорируются. Обработчик добавляет 2 или умножает на 2 текущее значение обрабатываемого числа (начальное значение инициализировать единицей) в зависимости от полученного пользовательского сигнала и сразу же выводит результат на экран. Вычисление и вывод производятся один раз в секунду. Получив сигнал **`SIGTERM`**, обработчик завершает свою работу, выведя сообщения о штатном завершении работы по сигналу от другого процесса.

Лабораторная работа №4. Работа с файлово-каталожной системой в ОС Linux

Рассматриваемые вопросы

1. Основные команды и утилиты для работы с файлами и каталогами
2. Использование механизма жестких и символьных ссылок

Основные команды для работы с файлами и каталогами

`cd` – изменение текущего каталога

`cp` - копирование файла

`ls` - вывод списка файлов (в том числе каталогов) в указанном каталоге

`file` - вывод типа указанного файла

`find` – утилита поиска файлов

`ln` - создание жестких и символьных ссылок

`mkdir` - создание каталога

`mv` - перемещение файла или каталога

pwd – вывод имени текущего каталога
rm - удаление файла
rmdir - удаление каталога
cat - слияние и вывод содержимого файлов

Ссылки на файлы
В Linux существует два вида ссылок на файл, обычно называемых жесткая ссылка и символьная, или "мягкая" ссылка.
Жесткая ссылка является собственно символьным именем какого-либо файла – записью в соответствующем каталоге с указанием индексного дескриптора этого файла. Таким образом, файл может иметь одновременно несколько символьных имен, в том числе в различных каталогах. Файл будет удален с диска только тогда, когда будет удалено последнее из его символьных имен. С этой точки зрения все жесткие ссылки равноправны: все символьные имена одного файла имеют одинаковый статус, ссылаясь на один и тот же индексный дескриптор.
Мягкая ссылка (или символьная ссылка, или symlink) принципиально отличается от жесткой ссылки, поскольку она является отдельным специальным файлом (со своим индексным дескриптором), который содержит полный путь к другому файлу. Преимуществом символьной ссылки является то, что она может указывать на файлы, которые находятся на других файловых системах, и в том числе могут быть недоступными в данный момент времени. Когда производится попытка доступа к файлу, ядро операционной системы заменяет ссылку на тот путь, который она содержит. Команда **rm** удаляет саму ссылку, а не файл, на который она указывает. Для того, чтобы опросить доступность файла, на который указывает символьная ссылка, а при доступности вывести его полное имя, используется команда **readlink**.

Имя файла может задаваться как с помощью абсолютного пути от корня файловой системы, например, **/home/user/file**, так и с помощью относительного пути – пути, заданного относительно текущего каталога. Относительные пути часто используют в скриптах, для того чтобы иметь возможность, запуская скрипт в различных каталогах, создавать прямо в них временные файлы, подкаталоги т т.д.
Для этого в каждом каталоге есть два служебных подкаталога:
.. – указывает на родительский каталог
. – указывает на текущий каталог
Например, команда **cd ..** позволит перейти на уровень выше – в родительский каталог, а команда **cd .** не изменит текущий каталог.
Другой часто встречающийся пример: команда **./script.bash.** Она запускает скрипт именно из текущего каталога, даже если в каталогах, перечисленных в системной переменной $PATH будут встречаться файлы с таким же именем.

Наконец, если мы, находясь в домашнем каталоге пользователя user, используем путь к файлу **./../../home/user/file** он будет соответствовать пути к файлу **/home/user/file**.
Все пути формируются от корневого каталога, перейти в который можно используя команду **cd /**
Для обозначения домашнего каталога текущего пользователя используется символ **~**. Соответственно, запись **cd ~** будет эквивалентна записи **cd $HOME**.

Задание на лабораторную работу
Создайте скрипты для перечисленных заданий и предъявите их преподавателю.
ВНИМАНИЕ! Все скрипты должны обрабатывать любые сценарии, соответствующие заданию, в том числе некорректный ввод параметров пользователем, использование имен файлов, содержащих необычные, но не запрещенные для использования в именах файлов символы, различные последовательности запусков разработанных скриптов и других действий пользователя в файловой системе. Все возникающие ошибки при выполнении скриптов, в том числе возникающие при выполнении отдельных утилит операционной системы, должны обрабатываться, и содержательные сообщения о них должны выводиться пользователю командами разрабатываемого скрипта.

1. Скрипт **rmtrash**
   a. Скрипту передается один параметр – имя удаляемого файла в текущем каталоге запуска скрипта.
   b. Скрипт проверяет, существует ли скрытый каталог **trash** в домашнем каталоге пользователя. Если он не существует – создает его.
   c. После этого скрипт создает в этом скрытом каталоге жесткую ссылку на удаляемый файл с уникальным именем (например, присваивает каждой новой ссылке имя, соответствующее очередному натуральному числу) и удаляет файл из текущего каталога.
   d. Затем в конец скрытого файла **trash.log** в домашнем каталоге пользователя помещается запись, содержащая полный путь к удаленному файлу и имя файла с созданной жесткой ссылкой.
2. Скрипт **untrash**
   a. Скрипту передается один параметр – короткое имя файла, который нужно восстановить (без полного пути – только имя).
   b. Скрипт по файлу **trash.log** определяет все записи, содержащие в качестве имени файла переданный параметр, и выводит на экран по одному полные имена таких файлов с запросом подтверждения для каждого.
   c. Если пользователь отвечает на подтверждение утвердительно, то предпринимается попытка восстановить файл по указанному полному пути (в соответствующем каталоге создается жесткая ссылка на файл из **trash** и затем удаляется соответствующий файл

из **trash**). Если каталог, указанный в полном пути к файлу, уже не существует, то файл восстанавливается в домашний каталог пользователя с выводом на экран соответствующего сообщения. При невозможности создать жесткую ссылку, например, из-за конфликта имен, пользователю предлагается изменить имя восстанавливаемого файла.

3. Скрипт **backup**
   a. Скрипт создает в каталоге **/home/user/** подкаталог с именем **Backup-YYYY-MM-DD**, где YYYY-MM-DD – дата запуска скрипта в том случае, если в **/home/user/** нет каталога с именем, соответствующим дате, отстоящей от текущей менее чем на 7 дней. Если в **/home/user/** уже есть каталог резервного копирования, созданный не ранее 7 дней от даты запуска скрипта, то новый каталог не создается. Для определения текущей даты и времени можно воспользоваться командой **date**.
   b. Если был создан новый каталог, то скрипт копирует в этот каталог все файлы из каталога **/home/user/source/** (для тестирования скрипта создайте такой каталог и набор файлов в нем). После этого скрипт дописывает в конец файла **/home/user/backup-report** следующую информацию: строка со сведениями о создании нового каталога с резервными копиями с указанием его имени и даты создания, а затем список файлов из **/home/user/source/**, которые были скопированы в этот каталог.
   c. Если есть подходящий готовый каталог для резервного копирования, скрипт копирует в него все файлы из **/home/user/source/** по следующим правилам: если файла с таким именем в каталоге резервного копирования нет, то он просто копирует файл из **/home/user/source**. Если файл с таким именем есть, то сравнивает его размер с размером одноименного файла в действующем каталоге резервного копирования. Если размеры совпадают, то файл не копируется. Если размеры отличаются, то скрипт копирует файл с автоматическим созданием версионной копии. Таким образом, в актуальном каталоге резервного копирования сохраняются обе версии файла: ранее записанный файл переименовывается путем добавления дополнительного постфикса «**.YYYY-MM-DD**» (дата запуска скрипта), а новый копируемый файл сохраняет имя. После окончания копирования в файл **/home/user/backup-report** дописывается строка о внесении изменений в актуальный каталог резервного копирования с указанием его имени и даты внесения изменений, затем дописываются строки, содержащие имена добавленных файлов с новыми именами, а затем дописываются строки с именами

добавленных файлов с существовавшими ранее именами с указанием через разделитель нового имени, присвоенного предыдущей версии этого файла.

4. Скрипт **upback**
   a. Скрипт копирует в каталог **/home/user/restore/** все файлы из актуального на данный момент каталога резервного копирования (имеющего в имени наиболее новую дату), за исключением файлов, являющихся предыдущими версиями.

Лабораторная работа №5. Управление памятью в ОС Linux

Рассматриваемые вопросы

1. Использование утилиты top для мониторинга параметров памяти
2. Использование имитационных экспериментов для анализа работы механизмов управления памятью.

Основные источники данных о состоянии памяти вычислительного узла

команда **free**
файл **/proc/meminfo** (документация в соответствующем разделе **man proc**)
файл **/proc/[PID]/statm** (документация в соответствующем разделе **man proc**)
утилита **top**

Организация управления памятью в ОС Linux

В Linux используется страничная организация виртуальной памяти. Память разбита на страницы. Размер страницы можно посмотреть в параметрах конфигурации с помощью команды **getconf PAGE_SIZE**. При обращении к адресу в памяти происходит динамическое преобразование адреса путем замены старших бит виртуального адреса на номер физической страницы с сохранением значения младших бит как смещения на странице.

Обычно, кроме физической памяти, используется также раздел подкачки. В этом случае адресное пространство процесса состоит из страниц, находящихся в оперативной памяти, и страниц, находящихся в разделе подкачки. Параметры раздела подкачки можно узнать из файла **/proc/swaps**. При динамическом выделении памяти процессу операционная система сначала пытается выделить страницы в физической памяти, но если это невозможно, инициирует страничный обмен, в рамках которого ряд страниц из физической памяти вытесняется на раздел подкачки, а адреса, соответствующие вытесненным страницам, выделяются процессу под новые страницы.

Операционная система контролирует выделение памяти процессам. Если процесс попробует запросить расширение адресного пространства, которое невозможно в пределах имеющейся свободной оперативной памяти, его работа будет аварийно остановлена с записью в системном журнале.

Задание на лабораторную работу

Проведите два виртуальных эксперимента в соответствии с требованиями и проанализируйте их результаты. В указаниях ниже описано, какие данные необходимо фиксировать в процессе проведения экспериментов. Рекомендуется написать «следящие» скрипты и собирать данные, например, из вывода утилиты **`top`** автоматически с заданной периодичностью, например, 1 раз в секунду. Можно проводить эксперименты и фиксировать требуемые параметры и в ручном режиме, но в этом случае рекомендуется замедлить эксперимент, например, уменьшив размер добавляемой к массиву последовательности с 10 до 5 элементов.

Требования к проведению экспериментов и содержанию отчета

Зафиксируйте в отчете данные о текущей конфигурации операционной системы в аспекте управления памятью:

- Общий объем оперативной памяти.
- Объем раздела подкачки.
- Размер страницы виртуальной памяти.
- Объем свободной физической памяти в ненагруженной системе.
- Объем свободного пространства в разделе подкачки в ненагруженной системе.

Эксперимент №1

Подготовительный этап:

Создайте скрипт **`mem.bash`**, реализующий следующий сценарий. Скрипт выполняет бесконечный цикл. Перед началом выполнения цикла создается пустой массив и счетчик шагов, инициализированный нулем. На каждом шаге цикла в конец массива добавляется последовательность из 10 элементов, например, (1 2 3 4 5 6 7 8 9 10). На каждом 100000-ом шаге в файл **`report.log`** добавляется строка с текущим значением размера массива (перед запуском скрипта файл обнуляется).

Первый этап:

Задача – оценить изменения параметров, выводимых утилитой **`top`** в процессе работы созданного скрипта.

Ход эксперимента:

Запустите созданный скрипт **`mem.bash`**. Дождитесь аварийной остановки процесса и вывода в консоль последних сообщений системного журнала. Зафиксируйте в отчете последнюю запись журнала - значения параметров, с

которыми произошла аварийная остановка процесса. Также зафиксируйте значение в последней строке файла **report.log**.
Подготовьте две консоли. В первой запустите утилиту **top**. Во второй запустите скрипт и переключитесь на первую консоль. Убедитесь, что в **top** появился запущенный скрипт. Наблюдайте за следующими значениями (и фиксируйте их изменения во времени в отчете):

- значения параметров памяти системы (верхние две строки над основной таблицей);
- значения параметров в строке таблицы, соответствующей работающему скрипту;
- изменения в верхних пяти процессах (как меняется состав и позиции этих процессов).

Проводите наблюдения и фиксируйте их в отчете до аварийной остановки процесса скрипта и его исчезновения из перечня процессов в **top**.
Посмотрите с помощью команды **dmesg | grep "mem.bash"** последние две записи о скрипте в системном журнале и зафиксируйте их в отчете. Также зафиксируйте значение в последней строке файла **report.log**.

Второй этап:
Задача – оценить изменения параметров, выводимых утилитой **top** в процессе работы нескольких экземпляров созданного скрипта.
Ход эксперимента:
Создайте копию скрипта, созданного на предыдущем этапе, в файл **mem2.bash**. Настройте её на запись в файл **report2.log**. Создайте скрипт, который запустит немедленно друг за другом оба скрипта в фоновом режиме.
Подготовьте две консоли. В первой запустите утилиту **top**. Во второй запустите созданный перед этим скрипт и переключитесь на первую консоль. Убедитесь, что в **top** появились **mem.bash** и **mem2.bash**. Наблюдайте за следующими значениями (и фиксируйте их изменения во времени в отчете):

- значения параметров памяти системы (верхние две строки над основной таблицей);
- значения параметров в строке таблицы, соответствующей работающему скрипту;
- изменения в верхних пяти процессах (как меняется состав и позиции этих процессов).

Проводите наблюдения и фиксируйте их в отчете до аварийной остановки последнего из двух скриптов и их исчезновения из перечня процессов в **top**.
Посмотрите с помощью команды **dmesg | grep "mem[2]*.bash"** последние записи о скриптах в системном журнале и зафиксируйте их в отчете. Также зафиксируйте значения в последних строках файлов **report.log** и **report2.log**.

Обработка результатов:

Постройте графики изменения каждой из величин, за которыми производилось наблюдение на каждом из этапов. Объясните динамику изменения этих величин исходя из теоретических основ управления памятью в рамках страничной организации памяти с разделом подкачки. Объясните значения пороговых величин: размер массива, при котором произошла аварийная остановка процесса, параметры, зафиксированные в момент аварийной остановки системным журналом. Сформулируйте письменные выводы.

Эксперимент №2

Подготовительный этап:

Создайте копию скрипта **mem.bash** в файл **newmem.bash**. Измените копию таким образом, чтобы она завершала работу, как только размер создаваемого массива превысит значение **N**, передаваемое в качестве параметра скрипту. Уберите запись данных в файл.

Основной этап:

Задача – определить граничные значения потребления памяти, обеспечивающие безаварийную работу для регулярных процессов, запускающихся с заданной интенсивностью.

Ход эксперимента:

Создайте скрипт, который будет запускать **newmem.bash** каждую секунду, используя один и тот же параметр **N** так, что всего будет осуществлено **K** запусков. Возьмите в качестве значения **N**, величину, в 10 раз меньшую, чем размер массива, при котором происходила аварийная остановка процесса в первом этапе предыдущего эксперимента. Возьмите в качестве **K** значение 10. Убедитесь, что все **K** запусков успешно завершились и в системном журнале нет записей об аварийной остановке **newmem.bash**.

Измените значение **K** на 30 и снова запустите скрипт. Объясните, почему ряд процессов завершился аварийно. Подберите такое максимальное значение **N**, чтобы при **K=30** не происходило аварийных завершений процессов.

Укажите в отчете сформулированные выводы по этому эксперименту и найденное значение **N**.

Лабораторная работа №6. Консольный интерфейс ОС Microsoft Windows

Рассматриваемые вопросы

1. Познакомиться с интерфейсом командной строки MS Windows.
2. Научится управлять основными объектами ОС MS Windows, используя командные файлы (скрипты).

Методические рекомендации

1. Работа может быть выполнена в любой версии MS Windows (от Windows XP и выше), но требует прав администратора для выполнения. Рекомендуется работать в виртуальной машины во избежание возможных повреждений конфигурации основной операционной системы.
2. Запуск командной строки осуществляется через меню пуск–выполнить–cmd. Работа может быть выполнена с использованием среды PowerShell.
3. Рекомендуемые команды и утилиты (в случае использования обычного интерпретатора командной строки): **`cmd, mem, discpart, cd, md, rd, dir, copy, xcopy, at, sc, find, sort, if, for call`**. Приведенный список команд неполон, при выполнении заданий можно использовать другие команды, в том числе устанавливая дополнительные компоненты операционной системы.
4. Перенаправление вывода команды или утилиты в файл осуществляется с помощью символа **`>.`**
5. **`Имя_команды /?`** позволяет получить справку по команде или утилите.

Задание на лабораторную работу

1. *Работа с файлами и каталогами*
   1. Создать в корне диска **`C:`** каталог с именем **`LAB6.`** В нем создать отдельные файлы с информацией о версии операционной системы, о свободной и занятой памяти и о подключенных к системе жестких дисках. Имена файлов должны соответствовать применяемым для получения соответствующих данных утилитам.
   2. Создать подкаталог **`TEST`** и скопировать в него содержимое каталога **`LAB6.`** Сделать этот каталог текущим.
   3. Одной командой создать файл, объединив содержимое всех файлов каталога **`TEST`**.
   4. Удалить все файлы в текущем каталоге, кроме файла, созданного последним, не перечисляя имена файлов явным образом.
   5. Создать текстовый файл со списком использованных команд и параметрами их запуска, использованными для выполнения п.п. 1.1–1.4.

2. *Запуск и остановка процессов*
    1. Узнать имя компьютера (хоста). Создать общий сетевой ресурс **`\\имя_хостового_компьютера\temp`**
    2. Создать исполняемый командный файл, производящий копирование любого файла из каталога **`C:\Windows`** объемом более 20 Мбайт на ресурс **`\\имя_хостового_компьютера\temp`** с поддержкой продолжения копирования при обрыве. При необходимости создать большой файл, объединив в один архив несколько файлов меньшего размера.
    3. Запланировать однократный запуск исполняемого файла из предыдущего пункта по расписанию через 1 минуту.
    4. Проверить запуск процедуры копирования, и, как только создался соответствующий процесс, принудительно завершить его до его штатного завершения.
    5. Сравнить исходный и конечный файл. Вывести сведения о проверки их целостности.
    6. Продолжить копирование файла с места разрыва.
    7. Создать текстовый файл со списком использованных команд с параметрами их запуска, использованными для выполнения п.п. 2.1–2.6.
3. *Управление системными службами*
    1. Создать файл, содержащий список служб, работающих в данный момент в системе.
    2. Создать командный файл, выполняющий следующие действия:
        1. остановку службы **`DNS-client;`**
        2. через 1 минуту создающий файл, содержащий обновленный список служб, работающих в данный момент в системе;
        3. запускающий отдельный командный файл, сравнивающий файлы, полученные в пп. 3.1 и 3.2, и создающий разностный файл;
        4. еще через 1 минуту восстанавливающий работу остановленной службы.
    3. Создать текстовый файл со списком использованных команд и параметрами их запуска, использованными для выполнения пп. 3.1–3.2.
4. *Поиск и сортировка информации в файлах*
    1. Записать список всех имен драйверов, загруженных в системе, в файл **`DRIVERS,`** в виде таблицы.
    2. Отсортировать полученные в п.п. 4.1 данные в обратном лексикографическом порядке.
    3. Создать текстовый файл со списком использованных команд и параметрами их запуска, использованными для выполнения п.п. 4.1–4.2.

Рекомендуемая литература

1. Зубок Д.А., Маятин А.В. Операционные системы. Методические указания по выполнению лабораторных работ. – СПб: Университет ИТМО, 2015. – 48 с.
2. Дейтел, Харви М. Операционные системы = Operating Systems : [в т.] / Х. М. Дейтел, П. Дж. Дейтел, Д. Р. Чофнес ; пер. с англ. под ред. С. М. Молявко .— 3-е изд .— М. : Бином, 2011, 1023 с.: ил.
3. Таненбаум Э. Современные операционные системы. 3-е изд. / Э. Таненбаум. – Электрон. Дан. – Санкт-Петербург: Питер, 2013. – 2120 с. – Режим доступа: https://ibooks.ru/reading.php?productid=344100
4. Олифер В.Г., Олифер Н.А. Сетевые операционные системы Изд. 2-е. — СПб.: Питер, 2009 .— 668 с.
5. Карпов В.Е., Коньков К.А. Основы операционных систем: учебное пособие. Изд. 2-е, доп. и испр .— М.: Интернет-Университет информационных технологий (ИНТУИТ.РУ), 2005 .— 531 с.
6. Курячий Г.В. Операционная система Linux. Курс лекций : учебное пособие : рек. для студентов высших учебных заведений, обучающихся по специальностям в области информационных технологий / Г. В. Курячий, К. А. Маслинский ; ИНТУИТ .— М. : Интернет-Университет Информационных Технологий, 2011 .— 387, [1] с.
7. Колисниченко Д.Н. Linux. От новичка к профессионалу : [наиболее полное руководство] / Д. Н. Колисниченко .— СПб. : БХВ-Петербург, 2008 .— 852 с. : ил.
8. Далхаймер М.К. Уэлш М. Запускаем Linux. Пер. с англ. СПб.: Символ-плюс, 2008. – 992 с.
9. Тейнсли Д. Linux и Unix: программирование в shell. Руководство разработчика: Пер. с англ. – К.: Издательская группа BHV, 2001. – 464 с.
10. Купер. М. Искусство программирования на языке сценариев командной оболочки. Электронный ресурс. URL: http://www.opennet.ru:8101/docs/RUS/bash_scripting_guide/

Маятин Александр Владимирович
Кашевник Алексей Михайлович
Зубок Дмитрий Александрович

Операционные системы. Лабораторный практикум

Учебно-методическое пособие

В авторской редакции
Редакционно-издательский отдел Университета ИТМО
Зав. РИО Н.Ф. Гусарова
Подписано к печати
Заказ №
Тираж
Отпечатано на ризографе

Редакционно-издательский отдел
Университета ИТМО
197101, Санкт-Петербург, Кронверкский пр., 49, литер А