МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

УНИВЕРСИТЕТ ИТМО

Д.А. Зубок, А.В. Маятин

ОПЕРАЦИОННЫЕ СИСТЕМЫ
методические указания по выполнению лабораторных работ

Санкт-Петербург

2015

Зубок Д.А., Маятин А.В. Операционные системы. Методические указания по выполнению лабораторных работ. – СПб: Университет ИТМО, 2015. – 48 с.

Пособие адресовано бакалаврам, обучающимся по направлениям подготовки 09.03.02 «Информационные системы и технологии» и 09.03.03 «Прикладная информатика» и содержит теоретический минимум, методические указания и задания для выполнения лабораторных работ по дисциплине «Операционные системы».

Рекомендовано к печати Ученым советом факультета информационных технологий и программирования, протокол №3 от 31.03.2015.

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, 2015

© Зубок Д.А., 2015

© Маятин А.В., 2015

Содержание

Введение 4

Виртуальная среда для выполнения лабораторных работ 6

Лабораторная работа №1. Основы использования консольного интерфейса ОС GNU/Linux. 9

Лабораторная работа №2. Обработка текстовых потоков в ОС GNU/Linux 14

Лабораторная работа №3. Мониторинг процессов в ОС GNU/Linux 20

Лабораторная работа №4. Управление процессами в ОС GNU/Linux 26

Лабораторная работа №5. Работа с файлово-каталожной системой в ОС GNU/Linux 34

Лабораторная работа №6. Консольный интерфейс ОС Microsoft Windows 39

Рекомендуемая литература 44

Введение

Современные операционные системы являются сложными программными комплексами, в которых реализуется множество технологических решений. Главная задача операционной системы – обеспечить распределение ресурсов вычислительного узла между прикладными программами и предоставить пользователю интерфейс для доступа к приложениям и данным, а также средства для контроля и управления распределением ресурсов и выполнением приложений. От эффективности работы механизмов операционной системы и корректности решений по управлению ресурсами напрямую зависит производительность, надежность и безопасность работы прикладного программного обеспечения.

Операционные системы, используемые на рабочих станциях и домашних компьютерах, как правило, реализуют принципы прозрачности управления и невмешательства пользователя в распределение ресурсов и решение других задач операционной системы, делая акцент на предоставлении удобного графического интерфейса для доступа к приложениям и пользовательским данным. Но в случае использования операционных систем для управления серверными приложениями, решения задач со специфичными требованиями к производительности, надежности и безопасности, необходимо регулярное администрирование операционной системы, мониторинг работы прикладного программного обеспечения и самой операционной системы, выбор оптимальных параметров управления, при необходимости ручное управление. Для этого необходимы не только умения использовать интерфейсы управления приложениями и утилитами операционной системы, но прежде всего понимание принципов работы механизмов операционной системы.

Предлагаемый лабораторный практикум состоит из шести лабораторных работ. Первые пять лабораторных работ выполняются в операционной системе GNU/Linux CentOS. Это обусловлено тем, что во-первых более двух третей серверов во всем мире работают под управлением unix-based операционных систем, а во-вторых *nix операционные системы обладают специфичным подходом к организации управления ресурсами и администрированию, принципиально отличающимися от других семейств операционных систем, в частности, распространенных на домашних компьютерах и ноутбуках. Для *nix операционных систем характерен подход к управлению с помощью интерфейса командной строки и управляющих скриптов, а также отображение всей информации о состоянии и настройке операционной системы в виде текстовых файлов. Поэтому в рамках первых двух лабораторных работ студенты освоят синтаксис и основные конструкции скриптового языка bash – основного средства автоматизации администрирования *nix семейства операционных систем, научатся анализировать текстовые потоки и текстовые файлы с помощью специальных утилит: grep, sed, awk. Эти навыки необходимы для

выполнения дальнейших заданий, связанных уже непосредственно с управлением объектами операционной системы. Третья и четвертая лабораторные работы посвящены основным объектам, с которыми работает операционная система – процессам. В процессе выполнения этих лабораторных работ студенты получат навыки мониторинга параметров процессов и выделяемых им ресурсов, управления приоритетами процессов, запуском процессов в заданное время, научатся использовать основные механизмы обмена данными и управлением между процессами, в частности, механизм сигналов. Пятая лабораторная работа раскрывает специфику управления файлами и каталогами в *nix операционных системах. Студентам будет предложено разработать средства автоматизации безопасного удаления и резервного копирования данных, хранящихся в файловой системе. Выполнение описанных лабораторных работ предполагается только с использованием текстовой консоли ввода-вывода.

Последняя шестая лабораторная работа выполняется в операционной системе Microsoft Windows Server 2003. Несмотря на наличие графического пользовательского интерфейса, цель этой лабораторной работы также получить навыки автоматизации администрирования операционной системы с помощью написания управляющих скриптов. При этом, студенты познакомятся с организацией управления службами и драйверами, основными командами для управления файлами и каталогами, мониторинга и управления процессами.

Виртуальная среда для выполнения лабораторных работ

Лабораторные работы по дисциплине «Операционные системы» должны позволить студенту понять организацию работы операционной системы и ее основные механизмы и приобрести практические навыки решения типовых задач управления ресурсами и приложениями. В процессе выполнения лабораторных работ студент должен иметь полные права доступа к ресурсам и настройкам операционной системы, но при этом иметь возможность допускать естественные на этапе обучения ошибки. Эти требования предопределили выбор технических средств проведения лабораторных работ – виртуальных машин с развернутыми в них дистрибутивами операционных систем.

Первые пять лабораторных работ выполняются в операционной системе GNU/Linux CentOS. CentOS (Community ENTerprise Operating System) – это дистрибутив Linux, который создан на основе исходного кода коммерческого Red Hat Enterprise Linux, являющегося корпоративным стандартом де-факто и имеющим хорошую документацию. Студенты могут использовать для выполнения лабораторных работ как конфигурацию виртуальной машины, подготовленную преподавателем, так и самостоятельно создать виртуальную машину и установить операционную систему GNU/Linux CentOS, скачав дистрибутив с сайта http://www.centos.org/.

В качестве средства виртуализации при выполнении лабораторных работ используется решение Oracle VM VirtualBox. Это решение позволяет создавать и управлять виртуальными машинами с большим набором операционных систем, включая операционные системы на базе ядра семейства GNU/Linux. В свою очередь сама среда виртуализации может быть развернута под управлением большого набора операционных систем на хостовых компьютерах, в том числе ОС семейств Microsoft Windows, GNU/Linux и Mac OS. Это значительно расширяет возможности по организации самостоятельной работы студентов.

Благодаря тому, что Oracle VM VirtualBox распространяется бесплатно, студенты имеют возможность самостоятельно работать с той же конфигурацией виртуальной машины, что и во время аудиторных занятий. Виртуальная машина распространяется в виде файла жесткого диска в формате vdi. Рекомендуется выделить виртуальной машине не менее 1Gb оперативной мамяти (желательно 2 Gb). Для создания своей виртуальной машины из на основе этого файла необходимо в меню Oracle Virtual Box выбрать «Машина – Создать». В диалоговых окнах мастера создания виртуальной машины рекомендуется внести следующие значения параметров (рис1, 2, 3):

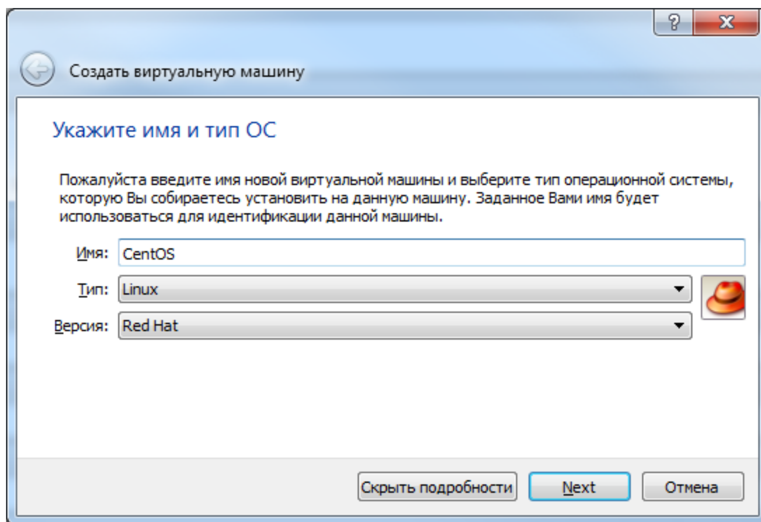

Рисунок 1. Выбор типа операционной системы

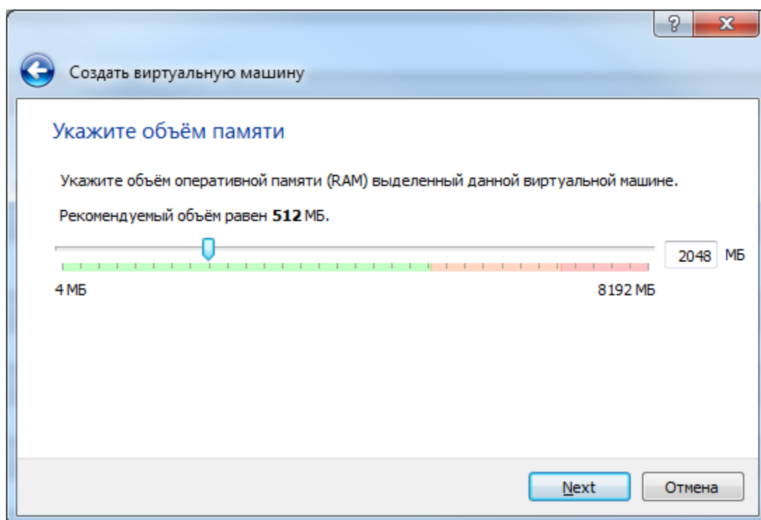

Рисунок 2. Определение объема оперативной памяти

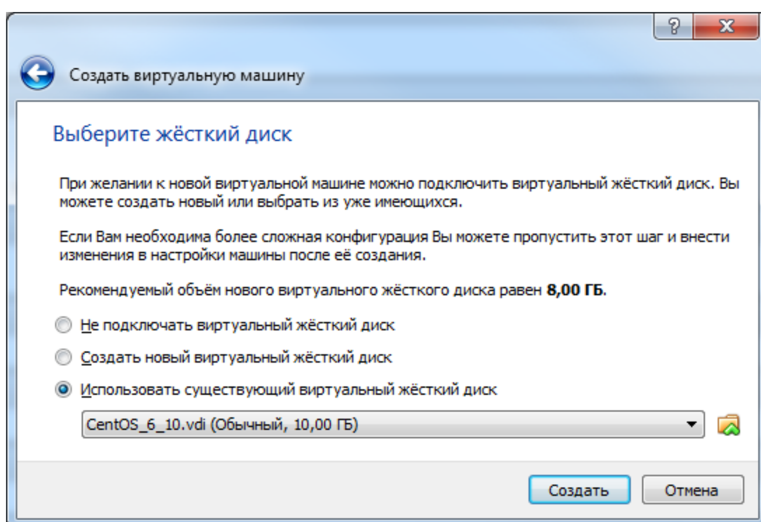

Рисунок 3. Выбор файла жесткого диска

Значительные преимущества для выполнения лабораторных работ дает возможность использования снимков виртуальных машин. Этот механизм позволяет зафиксировать состояние виртуальной машины в любой момент времени для того, чтобы потом была возможность вернуться к нему. Технически это устроено следующим образом. Данные виртуальной машины хранятся в одном файле, например в формате VDI. Если создается снимок, с этого момента все изменения перестают вносится в исходный файл виртуальной машины, а начинают записываться в новый разностный файл. Oracle VM VirtualBox позволяет строить деревья снимков, фиксируя тем самым различные достигнутые состояния виртуальной машины. Этот механизм очень удобен для комфортного изучения настроек и способов администрирования операционной системы. В случае ошибки, в том числе приводящей к невозможности дальнейшей корректной работы операционной системы нет необходимости в ее переустановке, требующей значительных временных затрат и потере материалов выполнения лабораторных работ. Всегда существует возможность отменить последний снимок и вернуться к предыдущему зафиксированному состоянию виртуальной машины и продолжить выполнение лабораторной работы. Снимок может быть сделан как с остановленной виртуальной машины, так и с виртуальной машины, находящейся в состоянии исполнения. Управление снимками осуществляется в отдельной вкладке приложения Oracle VM VirtualBox (рис.4).

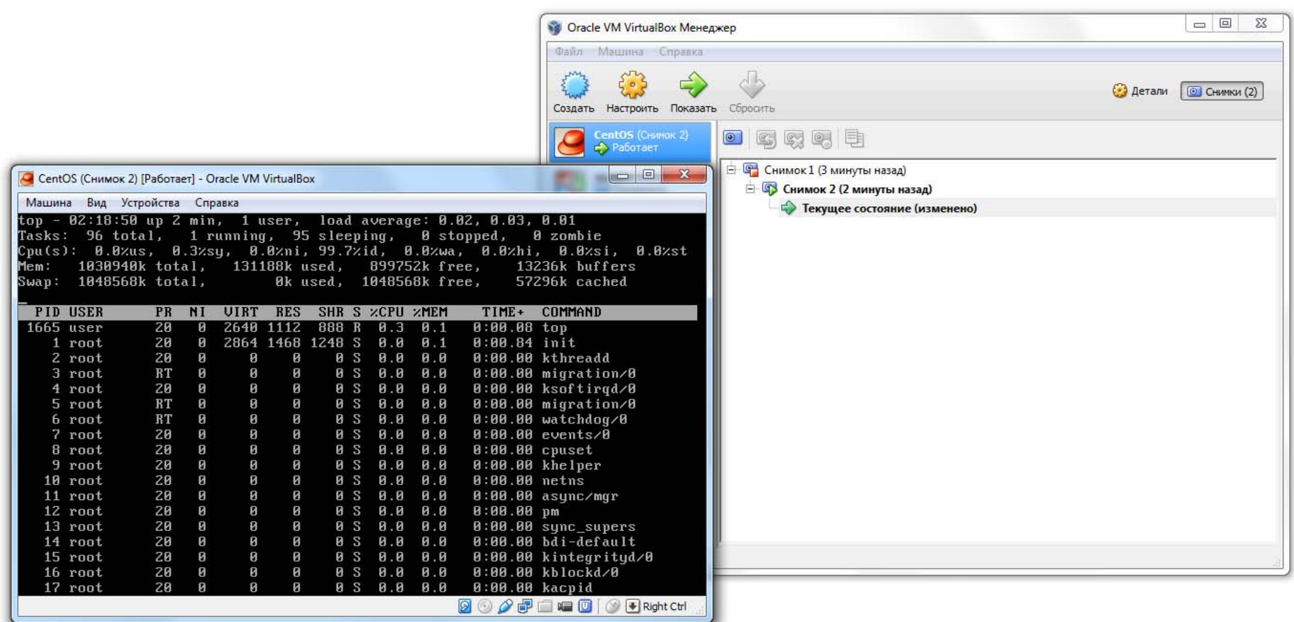

Рисунок 4. Управление снимками виртуальной машины

Лабораторная работа №1. Основы использования консольного интерфейса ОС GNU/Linux.

Рассматриваемые вопросы:

1. Работа с документацией по командам интерпретатора
2. Использование консольного текстового редактора
3. Создание скриптов для интерпретатора bash

Методические рекомендации:

Для получения подробного **справочного руководства** по любой команде можно набрать в консоли «man название команды», для кратной справки – название_команды **-h** или название_команды **--help**. Примеры: **man man** – справочное руководство по команде man; **man bash** – справочное руководство по интерпретатору **bash**.

Shell-скрипт – это обычный текстовый файл, в который последовательно записаны команды, которые пользователь может обычно вводить в командной строке. Файл выполняется командным интерпретатором – шеллом (shell). В Linux- и Unix-системах для того, чтобы бинарный файл или скрипт смогли быть запущены на выполнение, для пользователя, который запускает файл, должны быть установлены соответствующие права на выполнение. Это можно сделать с помощью команды **chmod u+x имя_скрипта**. В первой строке скрипта указывается путь к интерпретатору **#!/bin/bash**.

Для создания скрипта можно воспользоваться текстовым редактором **nano** или **vi**, набрав имя редактора в командной строке.

Ниже приводятся основные правила программирования на языке bash.

Комментарии. Строки, начинающиеся с символа # (за исключением комбинации #!), являются комментариями. Комментарии могут также располагаться и в конце строки с исполняемым кодом.

Особенности работы со строками. Одиночные кавычки (' '), ограничивающие строки с обеих сторон, служат для предотвращения интерпретации специальных символов, которые могут находиться в строке. Двойные кавычки (" ") предотвращают интерпретацию специальных символов, за исключением $, ` (обратная кавычка) и \ (escape – обратный слэш). Желательно использовать двойные кавычки при обращении к переменным. При необходимости вывести специальный символ можно также использовать экранирование: символ \ предотвращает интерпретацию следующего за ним символа.

Пробелы и переводы строк. Интерпретаторы sh и bash чувствительны к пробелам и переводам строк. Отдельные команды должны начинаться с новой строки. Если есть необходимость написать еще одну команду в той же строке, что и предыдущая – можно поставить точку с запятой в конце предыдущей команды. Пробел, как правило, разделяет название команды и параметры, которые ей передаются, а также параметры между собой.

Переменные. Имя переменной аналогично традиционному представлению об идентификаторе, т.е. именем может быть последовательность букв, цифр и подчеркиваний, начинающаяся с буквы или подчеркивания. Когда интерпретатор встречает в тексте сценария имя переменной, то он вместо него подставляет значение этой переменной. Поэтому ссылки на переменные называются подстановкой переменных. Если `variable1` – это имя переменной, то `$variable1` – это ссылка на ее значение. "Чистые" имена переменных, без префикса `$`, могут использоваться только при объявлении переменной или при присваивании переменной некоторого значения. В отличие от большинства других языков программирования, Bash не производит разделения переменных по типам. По сути, переменные Bash являются строковыми переменными, но, в зависимости от контекста, Bash допускает целочисленную арифметику с переменными. Определяющим фактором здесь служит содержимое переменных.

Оператор присваивания "=". При использовании оператора присваивания нельзя ставить пробелы слева и справа от знака равенства. Если в процессе присваивания требуется выполнить арифметические операции, то перед записью арифметического выражения используют оператор `let`, например:

```
Let a=2\*2
```

(оператор умножения является специальным символом и должен быть экранирован).

Арифметические операторы:

"+" сложение
"-" вычитание
"*" умножение
"/" деление (целочисленное)
"**" возведение в степень
"%" остаток от деления

Специальные переменные. Для Bash существует ряд зарезервированных имен переменных, которые хранят определенные значения.

– Позиционные параметры. 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, следующие за $9, 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
$EUID – эффективный UID.
$UID – ... содержит реальный идентификатор, который устанавливается только при логине.
$GROUPS – массив групп к которым принадлежит текущий пользователь
$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. Команда exit может использоваться для завершения работы сценария, точно так же как и в программах на языке C. Кроме того, она может возвращать некоторое значение, которое может быть проанализировано вызывающим процессом. Команде exit можно явно указать код возврата, в виде exit nnn, где nnn – это код возврата (число в диапазоне 0– 255).

Оператор вывода. Echo переменные_или_строки

Оператор ввода. Read имя_переменной. Одна команда read может прочитать (присвоить) значения сразу для нескольких переменных. Если переменных в read больше, чем их введено (через пробелы), оставшимся присваивается пустая строка. Если передаваемых значений больше, чем переменных в команде read, то лишние игнорируются.

Условный оператор.

```
If команда1
then команда2
[else
команда3]
fi.
```

Если `команда1` вернула после выполнения значение "истина", то выполняется `команда2` после `then`. Если есть необходимость сравнивать значения переменных и/или констант, после `if` используется специальная команда `[[ выражение ]].` Обязательно ставить пробелы между выражением и скобками, например:

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

Операции сравнения:

Операции сравнения целых чисел:
`-eq` # равно
`-ne` # не равно
`-lt` # меньше
`-le` # меньше или равно
`-gt` # больше
`-ge` # больше или равно

Операции сравнения строк:
`-z` # строка пуста
`-n` # строка не пуста
= или == # строки равны
`!=` # строки не равны
< # меньше (сравниваются коды символов)
<= # меньше или равно (сравниваются коды символов)
> # больше (сравниваются коды символов)
>= # больше или равно (сравниваются коды символов)

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

Множественный выбор. Для множественного выбора может применяться оператор `case`.

```
case переменная in
значение1 )
команда 1
 ;;
значение2 )
команда 2
 ;;
esac
```

Выбираемые значения обозначаются правой скобкой в конце значения. Разделитель ситуаций – `;;`

Цикл for. Существует два способа задания цикла for.

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

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

2. С-подобный

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

Цикл while: `while условие; do; команда; done.` Синтаксис записи условия такой же, как и в условном операторе, например:

```
i=0
while [ i –le 3 ]
do
echo $i
let i+=1
done
```

Управление циклами. Для управления ходом выполнения цикла служат команды `break` и `continue`. Они точно соответствуют своим аналогам в других языках программирования. Команда `break` прерывает исполнение цикла, в то время как `continue` передает управление в начало цикла, минуя все последующие команды в теле цикла.

Задание на лабораторную работу

1. Создайте свой каталог в директории /home/user/ Все скрипты создавайте внутри этого каталога или его подкаталогов. (`mkdir lab1`)
2. Напишите скрипты, решающие следующие задачи:

   i) В параметрах скрипта передаются две строки. Вывести сообщение о равенстве или неравенстве переданных строк.
   ii) В параметрах при запуске скрипта передаются три целых числа. Вывести максимальное из них.
   iii) Считывать строки с клавиатуры, пока не будет введена строка "q". После этого вывести последовательность считанных строк в виде одной строки.
   iv) Считывать с клавиатуры целые числа, пока не будет введено четное число. После этого вывести количество считанных чисел.
   v) Создать текстовое меню с четырьмя пунктами. При вводе пользователем номера пункта меню происходит запуск редактора nano, редактора vi, браузера links или выход из меню.
   vi) Если скрипт запущен из домашнего директория, вывести на экран путь к домашнему директорию и выйти с кодом 0. В противном случае вывести сообщение об ошибке и выйти с кодом 1.

3. Предъявите скрипты преподавателю и получите вопрос или задание для защиты лабораторной работы.
4. После защиты лабораторной работы удалите созданный директорий со всем его содержимым:

(`rm -R lab1`)

Лабораторная работа №2. Обработка текстовых потоков в ОС GNU/Linux

Рассматриваемые вопросы

1. Понятие стандартного ввода и стандартного вывода процесса
2. Перенаправление стандартного вывода в файл
3. Связь процессов по вводу/выводу
4. Использование вывода процесса как параметра другого процесса
5. Регулярные выражения и фильтрация текстовых потоков

Методические рекомендации:

Основным интерфейсом в операционных системах GNU/Linux является консольный интерфейс с текстовым вводом и выводом данных. Это определяет подход к управлению объектами операционной системы в их текстовом отображении. Например, состояние процессов отображается в виде набора текстовых файлов в псевдофайловой системе /proc, сведения о событиях в системе хранятся в текстовых файлах журналов, настройки отдельных пакетов в текстовых конфигурационных файлах. Это делает необходимым для решения дальнейших задач управления операционной системы освоение инструментария работы с текстовыми потоками.

Управление вводом-выводом команд (процессов)

У любого процесса по умолчанию всегда открыты три файла – **`stdin`** (стандартный ввод, клавиатура), **`stdout`** (стандартный вывод, экран) и **`stderr`** (стандартный вывод сообщений об ошибках на экран). Эти и любые другие открытые файлы могут быть перенаправлены. В данном случае термин "перенаправление" означает: получить вывод из файла (команды, программы, сценария) и передать его на вход в другой файл (команду, программу, сценарий). Дескрипторы файлов открытых по умолчанию:

```
0 = stdin
1 = stdout
2 = stderr
```

`команда > файл` – перенаправление стандартного вывода в файл, содержимое существующего файла удаляется.

`команда >> файл` – перенаправление стандартного вывода в файл, поток дописывается в конец файла.

`команда1 | команда2` – перенаправление стандартного вывода первой команды на стандартный ввод второй команды = образование конвейера команд.

`команда1 $(команда2)` – передача вывода команды 2 в качестве параметров при запуске команды 1. Внутри скрипта конструкция **`$(команда2)`** может использоваться, например, для передачи результатов работы команды 2 в параметры цикла **`for`** … **`in`**.

Работа со строками (внутренние 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** – регулярное выражение (см. ниже).

${string##substring}** – удаляет самую длинную из найденных подстрок **$substring в строке **$string.** Поиск ведется с начала строки. **$substring** – регулярное выражение.

${string/substring/replacement}** – замещает первое вхождение **$substring строкой **$replacement**. $substring – регулярное выражение.

${string//substring/replacement}** – замещает все вхождения **$substring строкой **$replacement**. **$substring** – регулярное выражение.

Работа со строками (внешние команды)

Для каждой команды доступно управление с помощью передаваемых команде параметров. Рекомендуем ознакомиться с документацией по этим командам с помощью команды man.

sort – сортирует поток текста в порядке убывания или возрастания, в зависимости от заданных опций.

uniq – удаляет повторяющиеся строки из отсортированного файла.

cut – извлекает отдельные поля из текстовых файлов (поле – последовательность символов в строке до разделителя).

head – выводит начальные строки из файла на **stdout.**

tail – выводит последние строки из файла на **stdout.**

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.

grep pattern [file...] – утилита поиска участков текста в файле(ах), соответствующих шаблону **pattern**, где **pattern** может быть как обычной строкой, так и регулярным выражением.

Sed – неинтерактивный "потоковый редактор". Принимает текст либо с устройства **stdin**, либо из текстового файла, выполняет некоторые операции над строками и затем выводит результат на устройство **stdout** или в файл. **Sed** определяет, по заданному адресному пространству, над какими строками следует выполнить операции. Адресное пространство строк задается либо их порядковыми номерами, либо шаблоном. Например, команда **3d** заставит **sed** удалить третью строку, а команда **/windows/d** означает, что все строки, содержащие "**windows**", должны быть удалены. Наиболее часто используются команды **p** – печать (на **stdout**), **d** – удаление и **s** – замена.

awk – утилита контекстного поиска и преобразования текста, инструмент для извлечения и/или обработки полей (колонок) в структурированных текстовых файлах. **Awk** разбивает каждую строку на отдельные поля. По умолчанию поля – это последовательности символов, отделенные друг от друга пробелами, однако имеется возможность назначения других символов в качестве разделителя полей. **Awk** анализирует и обрабатывает каждое поле в отдельности.

Регулярные выражения – это набор символов и/или метасимволов, которые наделены особыми свойствами.

Их основное назначение – поиск текста по шаблону и работа со строками. При построении регулярных выражений используются нижеследующие конструкции (в порядке убывания приоритета), некоторые из которых могут быть использованы 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

\n Строка, соответствующая **n**–му выражению **\(...\)**

r* Ноль или более вхождений символа **r**

r+ Одно или более вхождений символа **r**

r? Нуль или одно вхождение символа **r**

\<...\> 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

[:class:] альтернативный способ указания диапазона символов.

[:alnum:] соответствует алфавитным символам и цифрам. Эквивалентно выражению [**A-Za-z0-9**].

[:alpha:] соответствует символам алфавита. Эквивалентно выражению [**A-Za-z**].

[:blank:] соответствует символу пробела или символу табуляции.

[:cntrl:] соответствует управляющим символам

[:digit:] соответствует набору десятичных цифр. Эквивалентно выражению [**0-9**].

[:lower:] соответствует набору алфавитных символов в нижнем регистре. Эквивалентно выражению [**a-z**].

[:space:] соответствует пробельным символам (пробел и горизонтальная табуляция).

[:upper:] соответствует набору символов 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 свой каталог 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) Создать файл **errors.log**, в который поместить все строки из всех доступных для чтения файлов директории /**var/log**/, начинающиеся с последовательности символов ACPI, без указания имени файла, в котором встретилась строка. Вывести на экран те строчки из получившегося файла, которые содержат полные имена каких-либо файлов.

   ii) Создать **full.log**, в который вывести строки файла /**var/log/Xorg.0.log**, содержащие предупреждения и информационные сообщения, заменив маркеры предупреждений и информационных сообщений на слова **Warning**: и **Information:**, чтобы в получившемся файле сначала шли все информационные сообщения, а потом все предупреждения. Вывести этот файл на экран.

   iii) Создать файл **emails.lst**, в который вывести через запятую все адреса электронной почты, встречающиеся во всех файлах директории /**etc**.

   iv) Найти в директории /**bin** все файлы, которые являются сценариями, и вывести на экран полное имя файла с интерпретатором, наиболее часто используемым в этих сценариях (только полное имя файла).

   v) 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 – двоеточие.

   vi) Подсчитать общее количество строк в файлах, находящихся в директории /**var/log**/ и имеющих расширение **log**.

   vii) 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.
4. После защиты лабораторной работы удалите созданный каталог со всем его содержимым

   (**rm –R lab2**)

Лабораторная работа №3. Мониторинг процессов в ОС GNU/Linux

Рассматриваемые вопросы

1. Получение информации о запущенных процессах
2. Получение информации об используемых процессами ресурсах
3. Представление результатов в различном виде

Методические рекомендации:

Процесс – это совокупность набора исполняемых команд, ассоциированных с ним ресурсов и контекста выполнения управляемая операционной системы. Процесс может содержать несколько потоков исполнения. Потоки являются самостоятельными наборами исполняемых команд, но имеют доступ к общим ресурсам своего процесса. Как правило, диспетчирование операционная система выполняет именно на уровне потоков, но основной единицей управления является все же процесс.

Идентификация процессов

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

Все процессы, работающие в системе GNU/Linux, организованы в виде дерева. Корнем этого дерева является **init** – процесс системного уровня, запускаемый во время загрузки. Для каждого процесса хранится идентификатор его родительского процесса (**PPID,** Parent Process ID). У процесса **init PPID** равен 0.

Получение общих сведений о запущенных процессах

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

Запуск **ps** без аргументов покажет только те процессы, которые были запущены Вами и привязаны к используемому Вами терминалу.

Часто используемые параметры (указываются без "-"):

a – вывод процессов, запущенные всеми пользователями;

x – вывод процессов без управляющего терминала или с управляющим терминалом, но отличающимся от используемого Вами;

u – вывод для каждого из процессов имя запустившего его пользователя и времени запуска.

Обозначения колонок в типовом выводе команды **ps**:

PID, **PPID** – идентификатор процесса и его родителя.

%CPU – доля процессорного времени, выделенная процессу.

%MEM – процент используемой оперативной памяти.

VSZ – виртуальный размер процесса.

TTY – управляющий терминал, из которого запущен процесс.

STAT– статус процесса:

START – время запуска процесса.

TIME – время исполнения на процессоре.

Обозначения состояний процессов (в колонке **STAT**)

R – процесс выполняется в данный момент

S – процесс ожидает (т.е. спит менее 20 секунд)

I – процесс бездействует (т.е. спит больше 20 секунд)

D – процесс ожидает ввода/вывода (или другого недолгого события), непрырываемый

Z – zombie-процесс

T – процесс остановлен

Команда **pstree**

Команда **pstree** выводит процессы в форме дерева: можно сразу увидеть родительские процессы.

Часто используемые параметры:

-p – вывод **PID** всех процессов

-u – вывод имени пользователя, запустившего процесс.

Команда **top**

top – программа, используемая для наблюдения за процессами в режиме реального времени. Полностью управляется с клавиатуры. Вы можете получить справку, нажав на клавишу **h**. Наиболее полезные команды для мониторинга процессов:

Shift+M – эта команда используется для сортировки процессов по объему занятой ими памяти (поле **%MEM**);

Shift+P – эта команда 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. **top** спросит у вас его имя. Вам необходимо ввести имя пользователя, а не его UID. Если вы не введете никакого имени, будут показаны все процессы;

i – по умолчанию выводятся все процессы, даже спящие. Эта команда обеспечивает вывод информации только о работающих в данный момент процессах (процессы, у которых поле **STAT** имеет значение **R**, Running). Повторное использование этой команды вернет Вас назад к списку всех процессов.

Получение детальных сведений о запущенных процессах

/proc – псевдо-файловая система, которая используется в качестве интерфейса к структурам данных в ядре. Большинство расположенных в ней файлов доступны только для чтения, но некоторые файлы позволяют изменять переменные ядра.

Каждому запущенному процессу соответствует подкаталог с именем, соответствующим идентификатору этого процесса (его **PID**). Каждый из этих подкаталогов содержит следующие псевдо-файлы и каталоги (указаны наиболее часто использующиеся для мониторинга процессов).

Внимание! Часть из этих файлов доступна только в директориях процессов, запущенных от имени данного пользователя или при обращении от имени **root**.

cmdline – файл, содержащий полную командную строку запуска процесса.

cwd – ссылка на текущий рабочий каталог процесса.

environ – файл, содержащий окружение процесса. Записи в файле разделяются нулевыми символами, и в конце файла также может быть нулевой символ.

exe – символьная ссылка, содержащая фактическое полное имя выполняемого файла.

fd – подкаталог, содержащий одну запись на каждый файл, который в данный момент открыт процессом. Имя каждой такой записи соответствует номеру файлового дескриптора и является символьной ссылкой на реальный файл. Так, **0** – это стандартный ввод, **1** – стандартный вывод, **2** – стандартный вывод ошибок и т. д.

maps – файл, содержащий адреса областей памяти, которые используются программой в данный момент, и права доступа к ним. Формат файла следующий:

```
address           perms offset   dev   inode pathname
08048000-08056000 r-xp 00000000 03:0c 64593 /usr/sbin/gpm
08056000-08058000 rw-p 0000d000 03:0c 64593 /usr/sbin/gpm
08058000-0805b000 rwxp 00000000 00:00 0
40000000-40013000 r-xp 00000000 03:0c 4165 /lib/ld-2.2.so
bffff000-c0000000 rwxp 00000000 00:00 0
```

где **address** -- адресное пространство, занятое процессом; **perms** -- права доступа к нему:

r = можно читать

w = можно писать

x = можно выполнять

s = можно использовать несколькими процессами совместно

p = личная (копирование при записи);

offset -- смещение в файле, **dev --** устройство (старший номер : младший номер); **inode --** индексный дескриптор на данном устройстве: **0** означает, что с данной областью памяти не ассоциированы индексные дескрипторы;

stat – детальная информация о процессе в виде набора полей;

status – предоставляет бóльшую часть информации из **stat** в более лёгком для прочтения формате.

sched – предоставляет информацию о процессе, использующуюся планировщиком задач.

statm – предоставляет информацию о состоянии памяти в страницах как единицах измерения. Список полей в файле:

size общий размер программы

resident размер резидентной части

share разделяемые страницы

trs текст (код)

drs данные/стек

lrs библиотека

dt "дикие" (dirty) страницы

Обработка данных о процессах

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

Получение данных об оперативной памяти

free - возвращает информацию о свободной и используемой памяти в системе, как физической, так и виртуальной (в разделе подкачки на жестком диске).

Поля вывода команды:

total – общее количество доступной физической памяти. Некоторая область оперативной памяти может быть зарезервирована ядром, поэтому показатель total может быть меньше реального объема оперативной памяти.

used – объем используемой памяти (used=total-free).

free - свободная память.

shared - память, распределенная между процессами.

buffers - память используемая в буферах.

cached - память используемая для кэширования.

-/+ buffers/cache - использованная память без учета буферов и кэшей/свободная память с учётом буферов и КЭШей.

swap - использование раздела подкачки.

Задание на лабораторную работу

1. Создайте свой каталог в директории /**home/user/** Все скрипты и файлы для вывода результатов создавайте внутри этого каталога или его подкаталогов. (**mkdir lab3**)
2. Напишите скрипты, решающие следующие задачи:

   i) Посчитать количество процессов, запущенных пользователем **user**, и вывести в файл пары
   PID:команда для таких процессов.
   ii) Вывести на экран **PID** процесса, запущенного последним (с последним временем запуска).
   iii) Вывести в файл список **PID** всех процессов, которые были запущены командами, расположенными в **/sbin/**
   iv) Для каждого процесса посчитать разность резидентной и разделяемой части памяти процесса (в страницах). Вывести в файл строки вида **PID:разность**, отсортированные по убыванию этой разности.
   v) Для всех зарегистрированных в данный момент в системе процессов выведите в один файл строки
   ProcessID=PID : Parent_ProcessID=PPID : Average_Time=avg_atom.
   Значения **PPid** и **Pid** возьмите из файлов **status**, значение **avg_atom** из файлов **sched**, которые находятся в директориях с названиями, соответствующими **PID** процессов в **/proc** .
   Отсортируйте эти строки по идентификаторам родительских процессов.
   vi) В полученном на предыдущем шаге файле после каждой группы записей с одинаковым идентификатором родительского процесса вставить строку вида
   Average_Sleeping_Children_of_ParentID=N is M,
   где **N** = **PPID**, а **M** – среднее, посчитанное из **SleepAVG** для данного процесса.

3. Предъявите скрипты преподавателю и получите вопрос или задание для защиты лабораторной работы.
4. После защиты лабораторной работы удалите созданный каталог со всем его содержимым (**rm -R lab3**)

Лабораторная работа №4. Управление процессами в ОС GNU/Linux

Рассматриваемые вопросы

1. Директивы объединения команд
2. Команды для управления процессами
3. Планирование времени запуска процессов
4. Передача данных и управления между процессами

Методические рекомендации:

Основными задачами управления процессами в ОС GNU/Linux является управление приоритетами процессов, планирование запуска процессов по расписанию и организация обмена данными между процессами, например с помощью сигналов. Для автоматизации управления системные администраторы создают управляющие скрипты. Последовательности команд в управляющих скриптах могут быть построены с помощью традиционных операторов процедурного программирования (условный оператор, оператор цикла), но часто используются специальные директивы объединения команд.

Директивы (команды) объединения команд

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

`команда1 | команда2` – перенаправление стандартного вывода,

`команда1 ; команда2` – последовательное выполнение команд,

`команда1 && команда2` – выполнение команды при успешном завершении предыдущей,

`команда1 || команда2` – выполнение команды при неудачном завершении предыдущей,

`команда1 $(команда2)` – передача результатов работы команды 2 в качестве аргументов запуска команды 1,

`команда 1 > файл` – направление стандартного вывода в файл (содержимое существующего файла удаляется),

`команда 1 >> файл` – направление стандартного вывода в файл (поток дописывается в конец файла).

```
{
команда1
команда 2
}
```
– объединение команд после директив `||` , `&&` или в теле циклов и функций.

`команда1 &` – запуск команды в фоновом режиме (стандартный вход и стандартный выход не связаны с консолью, из которой запускается процесс; управление процессом возможно в общем случае только с помощью сигналов).

Команды для управления процессами
(с подробным описанием возможностей и синтаксисом команд можно ознакомиться в документации, доступной по команде **`man команда`**)

`kill` – передает сигнал процессу. Сигнал может передаваться в виде его номера или символьного обозначения. По умолчанию (без указания сигнала) передает сигнал завершения процесса. Идентификация процесса для команды **`kill`** производится по PID. Перечень системных сигналов, доступных в GNU/Linux, с указанием их номеров и символьных обозначений можно получить с помощью команды **`kill -l`**;

`killall` – работает аналогично команде kill, но для идентификации процесса использует его символьное имя, а не PID;

`pidof` – определяет PID процесса по его имени;

`pgrep` – определяет PID процессов с заданными характеристиками (например, запущенные конкретным пользователем);

`pkill` – позволяет отправить сигнал группе процессов с заданными характеристиками;

`nice` – запускает процесс с заданным значением приоритета. Уменьшение значения (повышение приоритета выполнения) 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 отсроченный запуск команды.

`cron` – демон, который занимается планированием и выполнением команд, запускаемых по определенным датам и в определенное время. Команды, выполняемые периодически, указываются в файле **`/etc/crontab`** (не через команду **`cron`**, а путем внесения строк в файл **`crontab`** или с

использованием одноименной команды **crontab**). Команды, которые должны быть запущены лишь однажды, добавляются при помощи **at.** Синтаксис строки в **crontab** подробно описан здесь:

Каждая команда в файле crontab занимает одну строку и состоит из шести полей:

минута час день_месяца месяц день_недели команда

Допустимые значения:

минута	**от 0 до 59**	
час	**от 0 до 23**	
день_месяца	**от 1 до 31**	
месяц	**от 1 до 12**	**(или три буквы от jan до dec, независимо от регистра)**
день_недели	**от 0 до 6**	**(0 это воскресенье или три буквы от sun до sat)**

Если в соответствующее поле поместить символ * это будет соответствовать любому возможному значению. Для полей можно указывать диапазоны значений, разделенных дефисом, например:

0 11 6-9 1-3 * echo "Hello World!" – вывод "Hello World!" в 11:00 в 6,7,8,9 дни января, февраля и марта.

0 */2 * * mon echo "Hello World!" – вывод "Hello World!" каждый четный час каждого понедельника

tail – не только выводит последние n строк из файла, но и позволяет организовать "слежение" за файлом – обнаруживать и выводить новые строки, появляющиеся в конце файла.

sleep – задает паузу в выполнении скрипта.

Организация взаимодействия двух процессов

Существует несколько вариантов организации взаимодействия процессов. Поскольку суть взаимодействия состоит в передаче данных и/или управления от одного процесса к другому, рассмотрим два распространенных варианта

организации такого взаимодействия: передачу данных через файл и передачу управления через сигнал.

Взаимодействие процессов через файл

Для демонстрации передачи информации через файл рассмотрим два скрипта – «Генератор» и «Обработчик». Требуется считывать информацию с консоли с помощью процесса «Генератор» и выводить ее на экран с помощью процесса «Обработчик», причем таким образом, чтобы считывание генератором строки «QUIT» приводило к завершению работы обработчика. Каждый скрипт запускается в своей виртуальной консоли. Переключаясь между консолями, можно управлять скриптами и наблюдать результаты их работы.

<table>
<tr><th>Генератор</th><th>Обработчик</th></tr>
<tr><td><pre>#!/bin/bash
while true; do
        read LINE
        echo $LINE >> data.txt
done</pre></td><td><pre>#!/bin/bash
(tail -n 0 -f data.txt) |
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 дописывает их в конец файла data.txt.

Скрипт «Обработчик» рассмотрим подробнее.

Команда **`tail`** позволяет считывать последние **`n`** строк из файла. Но один из наиболее распространенных вариантов ее использования – организация «слежения» за файлом. При использовании конструкции **`tail -f`** считывание из файла будет происходить только в случае добавления информации в этот

файл. При этом ключ **`-n 0`** предотвращает чтение из файла, пока его содержимое не обновилось после запуска команды **`tail.`** Поскольку необходимо передавать выход команды **`tail`** на вход скрипта «Обработчик», используем конструкцию **`(команды)|`** Круглые скобки позволяют запустить независимый подпроцесс (дочерний процесс) внутри родительского процесса «Обработчик», а оператор конвейера в конце позволит направить выход этого подпроцесса на вход родительского процесса. Таким образом, команда **`read`** в этом скрипте читает выход команды **`tail.`** Остальная часть скрипта основывается на конструкциях, изученных в предыдущих лабораторных работах, и не требует детального рассмотрения. Исключение составляет только команда **`killall tail.`** С ее помощью завершается вызванный в подпроцессе процесс **`tail`** перед завершением родительского процесса. Использование **`killall`** в этом случае используется для упрощения кода, но не всегда является корректным. Лучше 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. Существуют «стандартные» (системные) сигналы, имеющие фиксированные имена и названия (например, SIGTERM, SIGKILL и т.д.), но существует возможность передавать процессу и вновь создаваемый, пользовательский сигнал.

Таблица 1. Часто используемые сигналы

№	**Имя**	**Описание**	**Можно перехватывать**	**Можно блокировать**	**Комбинация клавиш**
1	HUP	Hangup. Отбой. Получение этого сигнала как правило означает, что завершил работу терминал из которого был запущен процесс и следовательно процесс тоже должен быть завершен.	Да	Да	
2	INT	Interrupt. В случае выполнения простых команд вызывает прекращение выполнения, в интерактивных программах - прекращение активного процесса	Да	Да	<Ctrl>+<C> или <Del>
3	QUIT	Как правило, сильнее сигнала Interrupt	Да	Да	<Ctrl>+<\>

4	ILL	Illegal Instruction. Центральный процессор столкнулся с незнакомой командой (в большинстве случаев это означает, что допущена программная ошибка). Сигнал отправляется программе, в которой возникла проблема	Да	Да	
8	FPE	Floating Point Exception. Вычислительная ошибка, например, деление на ноль	Да	Да	
9	KILL	Всегда прекращает выполнение процесса	Нет	Нет	
11	SEGV	Segmentation Violation. Доступ к недозволенной области памяти	Да	Да	
13	PIPE	Была предпринята попытка передачи данных с помощью конвейера или очереди FIFO, однако не существует процесса, способного принять эти данные	Да	Да	
15	TERM	Software Termination. Требование закончить процесс (программное завершение)	Да	Да	
17	CHLD	Изменение статуса порожденного процесса	Да	Да	
18	CONT	Продолжение выполнения приостановленного процесса	Да	Да	
19	STOP	Приостановка выполнения процесса	Нет	Нет	
20	TSTR	Сигнал останова, генерируемый клавиатурой. Переводит процесс в фоновый режим	Да	Да	<Ctrl>+<Z>

В случае системных сигналов, как правило, процесс имеет обработчик этого сигнала – код, который выполнится в случае получения процессом этого сигнала. Для использования пользовательских сигналов необходимо написать свой обработчик.

Для обработки сигналов в **sh** (**bash**) используется встроенная команда **trap** с форматом

```
trap action signal
```

Команде нужно передать два параметра: действие при получении сигнала и сигнал, для которого будет выполняться указанное действие. Обычно в качестве действия указывают вызов функции, описанной выше в коде скрипта.

С помощью команды **trap** можно не только задать обработчик для пользовательского сигнала, но и подменить обработчик для некоторых из системных сигналов (кроме тех, перехват которых запрещен). В этом случае обработка сигнала перейдет к указанному в **trap** обработчику.

Для демонстрации передачи управления от одного процесса к другому 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 случае скрипт «Генератор» будет в бесконечном цикле считывать строки с консоли и бездействовать (используется оператор **:**) для любой входной строки, кроме строки STOP, получив которую, он отправит пользовательский сигнал **USR1** процессу «Обработчик». Поскольку процесс «Генератор» должен знать PID процесса «Обработчик», передача этого идентификационного номера осуществляется через скрытый файл. В процессе «Обработчик» определение PID процесса производится с помощью системной переменной **$$.**

Процесс «Обработчик» выводит на экран последовательность натуральных чисел до момента получения сигнала **USR1**. В этот момент

запускается обработчик **usr1()**, который меняет значение переменной MODE. В результате на следующем шаге цикла будет выведено сообщение о прекращении работы в связи с появлением сигнала, и работа скрипта будет завершена.

Задание на лабораторную работу

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

1. Создайте и однократно выполните скрипт (в этом скрипте нельзя использовать условный оператор и операторы проверки свойств и значений), который будет пытаться создать директорию **test** в домашней директории. Если создание директории пройдет успешно, скрипт выведет в файл **~/report** сообщение вида "**catalog test was created successfully**" и создаст в директории **test** файл с именем **Дата_Время_Запуска_Скрипта**. Затем независимо от результатов предыдущего шага скрипт должен опросить с помощью команды **ping** хост www.net_nikogo.ru и, если этот хост недоступен, дописать сообщение об ошибке в файл **~/report.**
2. Задайте еще один однократный запуск скрипта из пункта 1 через 2 минуты. 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 каждые 5 минут каждого часа в день недели, в который вы будете выполнять работу.
4. Создайте два фоновых процесса, выполняющих одинаковый бесконечный цикл вычисления (например, перемножение двух чисел). После запуска процессов должна сохраниться возможность использовать виртуальные консоли, с которых их запустили. Используя команду **top**, проанализируйте процент использования ресурсов процессора этими процессами. Добейтесь, чтобы тот процесс, который был запущен первым, использовал ресурс процессора не более чем на 20%.
5. Процесс «Генератор» передает информацию процессу «Обработчик» с помощью файла. Процесс «Обработчик» должен осуществлять следующую обработку новых строк в этом файле: если строка содержит единственный символ «**+**», то процесс «Обработчик» переключает режим на *сложение* и ждет ввода численных данных. Если строка содержит единственный символ «*****», то обработчик переключает режим на *умножение* и ждет ввода численных данных. Если строка содержит целое число, то обработчик осуществляет текущую активную операцию (выбранный режим) над текущим значением вычисляемой переменной и считанным значением (например, складывает или перемножает результат

предыдущего вычисления со считанным числом). При запуске скрипта режим устанавливается в сложение, а вычисляемая переменная приравнивается к 1. В случае получения строки **`QUIT`** скрипт выдает сообщение о плановой остановке и завершает работу. В случае получения любых других значений строки скрипт завершает работу с сообщением об ошибке входных данных.

6. Процесс «Генератор» считывает строки с консоли, пока ему на вход не поступит строка **`TERM`**. В этом случае он посылает системный сигнал **`SIGTERM`** процессу обработчику. Процесс «Обработчик» (как и в примере, выводящий в бесконечном цикле натуральное число каждую секунду) должен перехватить системный сигнал **`SIGTERM`** и завершить работу, предварительно выведя сообщение о завершении работы по сигналу от другого процесса.
7. Процесс «Генератор» считывает с консоли строки в бесконечном цикле. Если считанная строка содержит единственный символ «**+**», он посылает процессу «Обработчик» сигнал **`USR1`**. Если строка содержит единственный символ «*****», генератор посылает обработчику сигнал **`USR2`**. Если строка содержит слово **`TERM`**, генератор посылает обработчику сигнал **`SIGTERM`**. Другие значения входных строк игнорируются. Обработчик добавляет 2 или умножает на 2 текущее значение обрабатываемого числа (начальное значение принять на единицу) в зависимости от полученного пользовательского сигнала и выводит результат на экран. Вычисление и вывод производятся один раз в секунду. Получив сигнал **`SIGTERM`**, «Обработчик» завершает свою работу, выведя сообщения о завершении работы по сигналу от другого процесса.

Лабораторная работа №5. Работа с файлово-каталожной системой в ОС GNU/Linux

Рассматриваемые вопросы

1. Основные команды для работы с файлами и каталогами
2. Использование механизма ссылок
3. Прямая и косвенная адресация каталогов

Методические рекомендации

Понятие файла является фундаментальным понятием для *nix операционных систем. Кроме простых (регулярных) файлов с данными в этом семействе операционных систем принято реализовывать с помощью специальных типов файлов интерфейсы доступа к внешним устройствам,

отображения данных о процессах и ресурсах операционной системы, настроек компонентов операционной системы и пользовательских приложений и т.п. Следует отметить, что каталог (директорий) в *nix операционных системах также представляет собой специальный тип файла, хранящий имена и номера дескрипторов входящих в него подкаталогов и файлов.

Основные команды для работы с файлами и каталогами

`cd` - смена каталога

`cp` - копирование файлов

`ls` - выводит список файлов и каталогов текущей директории

`file` - указывает тип указанного файла

`find` - поиск файлов

`ln` - создание ссылок

`mkdir` - создание каталога

`mv` - перемещение файла или каталога

`pwd` – вывод имени текущего каталога

`rm` - удаления файла

`rmdir` - удаление каталога

`cat` - слияние и вывод файлов

Ссылки на файлы

В Linux существует два вида ссылок, обычно называемых жесткие ссылки и символьные, или "мягкие" ссылки.

Жесткая ссылка является всего лишь именем какого-либо файла – записью в соответствующем каталоге со ссылкой на индексный дескриптор этого файла. Таким образом, файл может иметь одновременно несколько имен в различных каталогах. Он будет удален с диска только тогда, когда будет удалено последнее из его имен. Нет такого понятия, как "настоящее" имя: все имена имеют одинаковый статус.

Мягкая ссылка (или символьная ссылка, или symlink) полностью отличается от жесткой ссылки: она является специальным файлом, который содержит путь к другому файлу. Таким образом, мягкая ссылка может указывать на файлы, которые находятся на других файловых системах, и не нуждается в наличии того файла, на который она указывает. Когда происходит попытка доступа к файлу, ядро операционной системы заменяет ссылку на тот путь, который она содержит. Однако команда **rm** удаляет саму ссылку, а не файл, на который она указывает. Для чтения состояния символьной ссылки, а также имени файла, на который она указывает, используется команда **readlink**.

Полное имя файла может задаваться как с использованием абсолютного пути, например, **/home/user/file**, так и с помощью относительного пути – пути, заданного относительно текущего каталога. Это особенно часто применяется в скриптах. Для этого в каждом каталоге есть два служебных каталога:

.. – указывает на родительский каталог

. – указывает на текущий каталог

Например, команда **cd ..** позволит перейти на уровень выше, а команда **cd .** ничего не изменит.

Другой пример: команда **./script.bash** запускает скрипт именно из текущего каталога.

Наконец, если мы находимся в домашнем каталоге пользователя user, то путь к файлу

./../../home/user/file

будет соответствовать пути к файлу в домашнем каталоге, как и описанный выше пример абсолютного пути.

Для того, чтобы перейти к корню файловой системы можно использовать команду **cd /**

Для обозначения домашнего каталога активного пользователя можно использовать символ **~**. Тогда запись **cd ~** будет эквивалентна записи **cd $HOME**.

Задание на лабораторную работу

Создайте скрипты для перечисленных заданий и предъявите их преподавателю.

1. Скрипт **`rmtrash`**
   a. Скрипту передается один параметр – имя файла в текущем каталоге вызова скрипта.
   b. Скрипт проверяет, создан ли скрытый каталог **`trash`** в домашнем каталоге пользователя. Если он не создан – создает его.
   c. После этого скрипт создает в этом каталоге жесткую ссылку на переданный файл с уникальным именем (например, присваивает каждой новой ссылке имя, соответствующее следующему натуральному числу) и удаляет файл в текущем каталоге.
   d. Затем в скрытый файл **`trash.log`** в домашнем каталоге пользователя помещается запись, содержащая полный исходный путь к удаленному файлу и имя созданной жесткой ссылки.
2. Скрипт **`untrash`**
   a. Скрипту передается один параметр – имя файла, который нужно восстановить (без полного пути – только имя).
   b. Скрипт по файлу **`trash.log`** должен найти все записи, содержащие в качестве имени файла переданный параметр, и выводить по одному на экран полные имена таких файлов с запросом подтверждения.
   c. Если пользователь отвечает на подтверждение положительно, то предпринимается попытка восстановить файл по указанному полному пути (создать в соответствующем каталоге жесткую ссылку на файл из **`trash`** и удалить соответствующий файл из **`trash`**). Если каталога, указанного в полном пути к файлу, уже не существует, то файл восстанавливается в домашний каталог пользователя с выводом соответствующего сообщения.
3. Скрипт **`backup`**
   a. Скрипт создаст в **`/home/user/`** каталог с именем **`Backup-YYYY-MM-DD`**, где YYYY-MM-DD – дата запуска скрипта, если в **`/home/user/`** нет каталога с именем, соответствующим дате, отстоящей от текущей менее чем на 7 дней. Если в **`/home/user/`** уже есть «действующий» каталог резервного копирования (созданный не ранее 7 дней от даты запуска скрипта), то новый каталог не создается. Для определения текущей даты можно воспользоваться командой **`date`**.

   b. Если новый каталог был создан, то скрипт скопирует в этот каталог все файлы из каталога **/home/user/source/** (для тестирования скрипта создайте такую директорию и набор файлов в ней). После этого скрипт выведет в режиме дополнения в файл **/home/user/backup-report** следующую информацию: строка со сведениями о создании нового каталога с резервными копиями с указанием его имени и даты создания; список файлов из **/home/user/source/**, которые были скопированы в этот каталог.
   c. Если каталог не был создан (есть «действующий» каталог резервного копирования), то скрипт должен скопировать в него все файлы из **/home/user/source/** по следующим правилам: если файла с таким именем в каталоге резервного копирования нет, то он копируется из **/home/user/source**. Если файл с таким именем есть, то его размер сравнивается с размером одноименного файла в действующем каталоге резервного копирования. Если размеры совпадают, файл не копируется. Если размеры отличаются, то файл копируется с автоматическим созданием версионной копии, таким образом, в действующем каталоге резервного копирования появляются обе версии файла (уже имеющийся файл переименовывается путем добавления дополнительного расширения «**.YYYY-MM-DD**» (дата запуска скрипта), а скопированный сохраняет имя). После окончания копирования в файл **/home/user/backup-report** выводится строка о внесении изменений в действующий каталог резервного копирования с указанием его имени и даты внесения изменений, затем строки, содержащие имена добавленных файлов с новыми именами, а затем строки с именами добавленных файлов с существовавшими в действующем каталоге резервного копирования именами с указанием через пробел нового имени, присвоенного предыдущей версии этого файла.
4. Скрипт **upback**
   a. Скрипт должен скопировать в каталог **/home/user/restore/** все файлы из актуального на данный момент каталога резервного копирования (имеющего в имени наиболее свежую дату), за исключением файлов с предыдущими версиями.
5. Все скрипты должны корректно обрабатывать любые передаваемые им входные параметры и значения. Не допускается вывод сообщений об ошибках от отдельных команд, использующихся в скрипте. В случае некорректных входных данных или невозможности выполнить операцию,

пользователю должно выводиться отдельное сообщение, формирующееся в скрипте. Перед предъявление результатов выполнения лабораторной работы преподавателю необходимо провести тестирование разработанных скриптов.

Лабораторная работа №6. Консольный интерфейс ОС Microsoft Windows

Рассматриваемые вопросы

1. Познакомиться с командной строкой ОС Microsoft Windows.
2. Получить навыки использования команд для работы с файлово-каталожной системой.
3. Научиться управлять службами и драйверами.

Методические рекомендации

В отличие от *nix операционных систем в операционных системах семейства Microsoft Windows основным является графический пользовательский интерфейс, предоставляющий доступ ко всем возможным для изменения параметрам операционной системы и приложений. Однако в операционных системах этого семейства есть возможность использовать командный интерпретатор и создавать скрипты для автоматизации задач администрирования.

Запуск командного интерпретатора (оболочки командной строки) производится исполняемым файлом cmd.exe, который находится в каталоге **`%SystemRoot%\SYSTEM32`**, где **`%SystemRoot%`** - переменная окружения, в которой записан путь к системному каталогу Windows (например, C:/Windows). Командный интерпретатор может выполнять команды двух типов: внутренние и внешние. Внутренние команды распознаются непосредственно интерпретатором. Любую команду, не относящуюся к внутренним, интерпретатор пытается рассматривать как внешнюю. Внешние команды реализуются отдельными утилитами, исполняемые файлы которых, как правило, располагаются в том же каталоге, что и cmd.exe. Вызов как внутренних, так и внешних команд нечувствителен к регистру.

Синтаксис вызова команды включает в себя название команды, после которого через пробелы следуют передаваемые команде параметры и ключи. В ОС семейства Microsoft Windows традиционно ключи обозначаются **`/ключ`**, например, **`/A`** или **`/X`**. Любая команда поддерживает ключ **`/?`**, с которым на экран выводится справка по этой команде. По структуре она напоминает man в *nix системах. Обязательными элементами справочной информации являются

описание назначения команды, синтаксис ее вызова и описание назначения ключей.

Командный интерпретатор cmd.exe поддерживает конструкции для перенаправления ввода и вывода команд аналогично *nix операционным системам: **>**, **>>**, **<**, **|**.

Основные команды, необходимые для выполнения лабораторной работы:

cmd – запускает командный интерпретатор.

echo – выводит текстовое сообщение в консоль или отключает/включает режим отображения в консоли информации о работе команд.

copy – копирует один или группу файлов. Может использоваться для объединения нескольких текстовых файлов в один. Команда не позволяет копировать файлы в подкаталогах.

xcopy – копирует файлы и каталоги, включая подкаталоги.

dir – выводит список файлов и подкаталогов каталога. В случае вызова команды без параметров, она отображает метку тома и серийный номер тома, за которыми следует список подкаталогов и файлов в текущем каталоге с указанием их даты и времени последнего изменения. Для файлов также выводится размер в байтах. После списка выводится общее число перечисленных файлов и подкаталогов, их суммарный размер и свободное пространство (в байтах) на диске.

cd – изменяет текущий каталог.

md – создает каталог.

rd – удаляет пустой каталог.

rm – удаляет файл.

find – ищет заданную строку текста в файле или в группе файлов. Результатом поиска является вывод на экран всех строк файлов, содержащих заданный образец.

sort – сортирует входной поток данных.

mem – выводит сведения об используемой и свободной памяти.

diskpart – самостоятельный командный интерпретатор для управления структурами хранения данных (разделами, дисками, томами). Может управляться с помощью команд или сценариев. Файл сценария **diskpart** — это текстовый файл с расширением **.txt**. Для использования подготовленного сценария команду **diskpart** необходимо вызвать с ключом **/s**.

at – запускает программы и команды в заданное время. Команду at можно использовать только при запущенной службе расписаний. Вызванная без параметров команда at выводит список всех команд и программ, которые будут запущены с ее помощью. Для вызова команды at пользователь должен быть членом локальной группы администраторов.

sc – управляет службами. С помощью параметров этой утилиты можно настроить конкретную службу, отобразить текущее состояние службы, остановить и запустить службу и т.д.

call – вызывает один пакетный файл (скрипт) из другого без завершения выполнения первого скрипта.

if – условный оператор.

Синтаксис

if [not] errorlevel число команда [else команда]

Условие выполняется, если предыдущая команда, обработанная интерпретатором команд **cmd.exe**, завершилась с кодом, равным или большим числа.

if [not] строка1==строка2 команда [else команда]

Условие выполняется, если строки **Строка1** и **Строка2** совпадают. Строки могут являться символьными выражениями или пакетными переменными (например, **%1**). Явно заданные строки нет необходимости заключать в кавычки.

if [not] exist имя_файла команда [else команда]

Условие выполняется, если существует файл с именем **имя_файла**.

for – оператор итеративного цикла.

Синтаксис

for {%переменная | %%переменная} in (множество) do команда

{%переменная | %%переменная} – обязательный параметр | замещаемый параметр. Если команда for вызывается из командной строки, необходимо использовать **%переменная**. Если команда for вызывается из пакетного файла (скрипта), то необходимо использовать **%%переменная**. В переменных учитывается регистр и они могут быть представлены буквами, например **%A**, **%B** или **%C**.

(множество) – обязательный параметр. Задает один или несколько файлов, каталогов, диапазон значений или текстовых строк, подлежащих обработке заданной командой. Скобки являются обязательными.

команда – обязательный параметр. Задает команду, которая будет выполнена для каждого файла, каталога, диапазона значений или текстовой строки, включенной в указанный параметр **(множество)**.

Приведенный список команд неполон, возможно использование других команд.

Задание на лабораторную работу

1. *Работа с файлами и директориями*
   1. Создать каталог на диске **C:** с именем **LAB6.** В нем создать файлы с информацией о версии операционной системы, свободной и загруженной памяти, жестких дисках, подключенных в системе. Имена файлов должны соответствовать применяемой команде.
   2. Создать подкаталог **TEST,** в него скопировать содержимое каталога **LAB6.**
   3. Создать одной командой файл с содержимым всех файлов каталога **LAB6.**
   4. Удалить все файлы в текущем каталоге, кроме созданного последним, указав явно имена удаляемых файлов.
   5. Создать текстовый файл со списком использованных команд и параметрами, использованными для выполнения п.п. 1.1–1.4.
2. *Запуск и удаление процессов*
   1. В ручную узнать **имя_хостового_компьютера** (свойства компьютера).
   2. Создать исполняемый файл, производящий копирование любого файла из дериктории **C:\cd** объемом более 2 Мбайт на ресурс **\\имя_хостового_компьютера\temp** с поддержкой продолжения копирования при обрыве.
   3. Настроить запуск файла по расписанию через 1 минуту.
   4. Проверить запуск копирования; если процесс появился, принудительно завершить его.
   5. Сравнить исходный и конечный файл. Проверить их целостность.
   6. Продолжить копирование с места разрыва.
   7. Создать текстовый файл со списком использованных команд с

параметрами, использованными для выполнения п.п. 2.1–2.5.

3. *Работа со службами*
   1. Получить файл, содержащий список служб, запущенных в системе.
   2. Создать командный файл обеспечивающий:
      1. остановку служб **`DNS-client;`**
      2. с временной задержкой, создание файла, содержащего обновленный список служб, запущенных в системе;
      3. запуск другого командного файла, сравнивающего файлы, полученные в пп. 3.1 и 3.2, и создающего разностный файл;
      4. восстановление работы служб.
   3. Создать текстовый файл со списком использованных команд и параметрами, использованными для выполнения пп. 3.1–3.2.
4. *Поиск и сортировка информации в файлах*
   1. Поместить список всех имен драйверов, загруженных в системе, в файл **`DRIVERS,`** в табличной форме**.**
   2. Отсортировать полученные в п.п. 4.1 данные в обратном порядке по алфавиту.
   3. Создать текстовый файл со списком использованных команд и параметрами, использованными для выполнения п.п. 4.1–4.2.

Рекомендуемая литература

1. Дейтел Х.М., Дейтел П. Дж., Чофнес Д.Р. Операционные системы Изд. 3-е. — М.: Бином, 2011. — 1023 с.
2. Олифер В.Г., Олифер Н.А. Сетевые операционные системы Изд. 2-е. — СПб.: Питер, 2009 .— 668 с.
3. Таненбаум Э.С. Операционные системы. Разработка и реализация. Изд. 2-е. — СПб.: Питер, 2006. — 576 с.
4. 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.
5. 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.
6. Маслаков В.Г. Linux / В. Г. Маслаков .— СПб. [и др.] : Питер, 2009 .— 330 с. : ил.
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. Торчинский Ф. UNIX. Практическое пособие администратора. СПб.: Символ-плюс, 2005. – 400 с.
10. Тейнсли Д. Linux и Unix: программирование в shell. Руководство разработчика: Пер. с англ. – К.: Издательская группа BHV, 2001. – 464 с.
11. Скловская С.Л. Команды Linux. Справочник. СПб.: ДиаСофтЮП, 2004. – 848 с.
12. Костромин В. А. Самоучитель Linux для пользователя. — СПб.: БХВ-Петербург, 2003. - 672 с.
13. Купер. М. Искусство программирования на языке сценариев командной оболочки. Электронный ресурс. URL: http://www.opennet.ru:8101/docs/RUS/bash_scripting_guide/

Миссия университета – генерация передовых знаний, внедрение инновационных разработок и подготовка элитных кадров, способных действовать в условиях быстро меняющегося мира и обеспечивать опережающее развитие науки, технологий и других областей для содействия решению актуальных задач.

Кафедра информационных систем

Кафедра информационных систем основана в 2000 году для реализации образовательной программы специалитета «Информационные системы и технологии». Первым заведующим кафедрой был доктор технических наук, профессор Анатолий Абрамович Шалыто. С 2004 года кафедру возглавляет лауреат премий президента и правительства Российской Федерации, доктор технических наук, профессор Владимир Глебович Парфенов.

Кафедра осуществляет подготовку магистров и бакалавров по направлениям «Информационные системы и технологии», «Прикладная информатика» и «Бизнес-информатика» и ежегодно выпускает более 150 специалистов по разработке, внедрению и управлению информационными системами. С момента основания кафедры и до 2015 года непрерывно осуществлялась подготовка инженеров по специальности «Информационные системы и технологии». В 2004 году состоялся первый набор в бакалавриат по направлению подготовки «Бизнес-информатика», а в 2007 году началась подготовка магистров по этому направлению. С 2011 года проводится подготовка бакалавров и магистров направления «Прикладная информатика».

В 2013 году магистерская программа по направлению подготовки «Бизнес-информатика» стала лауреатом конкурса на лучшие программы Министерства образования и науки Российской Федерации. В 2014 году магистерская программа «Комплексная автоматизация предприятий» прошла международную аккредитацию и получила сертификат EUR-ACE® Master. Организовано взаимодействие с Университетом Ростока (Германия) по подготовке совместной образовательной программы магистров по направлению подготовки «Бизнес-информатика».

Кафедра информационных систем осуществляет тесное сотрудничество с ведущими IT-компаниями Санкт-Петербурга. Студенты проходят практику и работают над выпускными квалификационными работами в компаниях: ЗАО

"Транзас Технологии", ООО "ТОПС Консалтинг", ООО «Софтверке», ООО "САП Лабс", ЗАО "ПЕТЕР-СЕРВИС", ООО «Санкт-Петербургский Центр Разработок ЕМС», ООО "1С:Северо-Запад", ООО "ОпенВэй Сервис"и др. Специалисты из компаний ООО «ЯНДЕКС», ООО «Эксперт-система», ООО «Центр речевых технологи» участвовали в разработке образовательных стандартов подготовки магистров.

Профессорско-преподавательский состав кафедры участвует в научно-исследовательской и научно-методической деятельности. На базе кафедры в рамках реализации Программы повышения конкурентоспособности НИУ ИТМО среди ведущих мировых научно-образовательных центров на 2013-2020 гг. создана и успешно развивается международная научная лаборатория «Интеллектуальные технологии для социо-киберфизических систем» (научные руководители – д.т.н., профессор А.В. Смирнов и профессор К. Сандхул (Германия)). В течение 5 лет совместно с ООО «Санкт-Петербургский Центр Разработок ЕМС» проводились научные исследования, в которых участвовали преподаватели и студенты кафедры. Кафедра участвует в организации и проведении Открытой олимпиады школьников «Информационные технологии».

Кафедра ИС располагает двумя лекционными аудиториями и восемью компьютерными классами с проекционным оборудованием и доступом в интернет для всех пользователей. В учебном процессе используется лицензионное программное обеспечение, включая среды разработки, мультимедийное и офисное ПО, а также учебные версии платформ 1С, DocsVision, Microsoft Dynamics AX, EMC Documentum.

Кафедра использует в учебном процессе собственный вычислительный кластер в составе узла виртуализации и системы хранения данных, а также необходимого телекоммуникационного оборудования. Узел виртуализации под управлением гипервизора Hyper-V развернут на однnetwork сервере на котором выполняются виртуальные машины с серверами приложений, серверами баз данных, учебными средами для обеспечения образовательных задач программы и т.д. Вся информация размещается в выделенной системе хранения данных.

Зубок Дмитрий Александрович, Маятин Александр Владимирович

ОПЕРАЦИОННЫЕ СИСТЕМЫ

методические указания
по выполнению лабораторных работ

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