Нижегородский государственный технический университет им. Р.Е. Алексеева

МОДУЛЬ 1

АДМИНИСТРИРОВАНИЕ LINUX

ВВЕДЕНИЕ

Нижний Новгород, 2021г.

РАЗНОВИДНОСТИ ОС

Операционные Системы

Windows

Microsoft corp.

Mac OS

Apple corp.

Linux

ИСТОРИЯ СОЗДАНИЯ LINUX

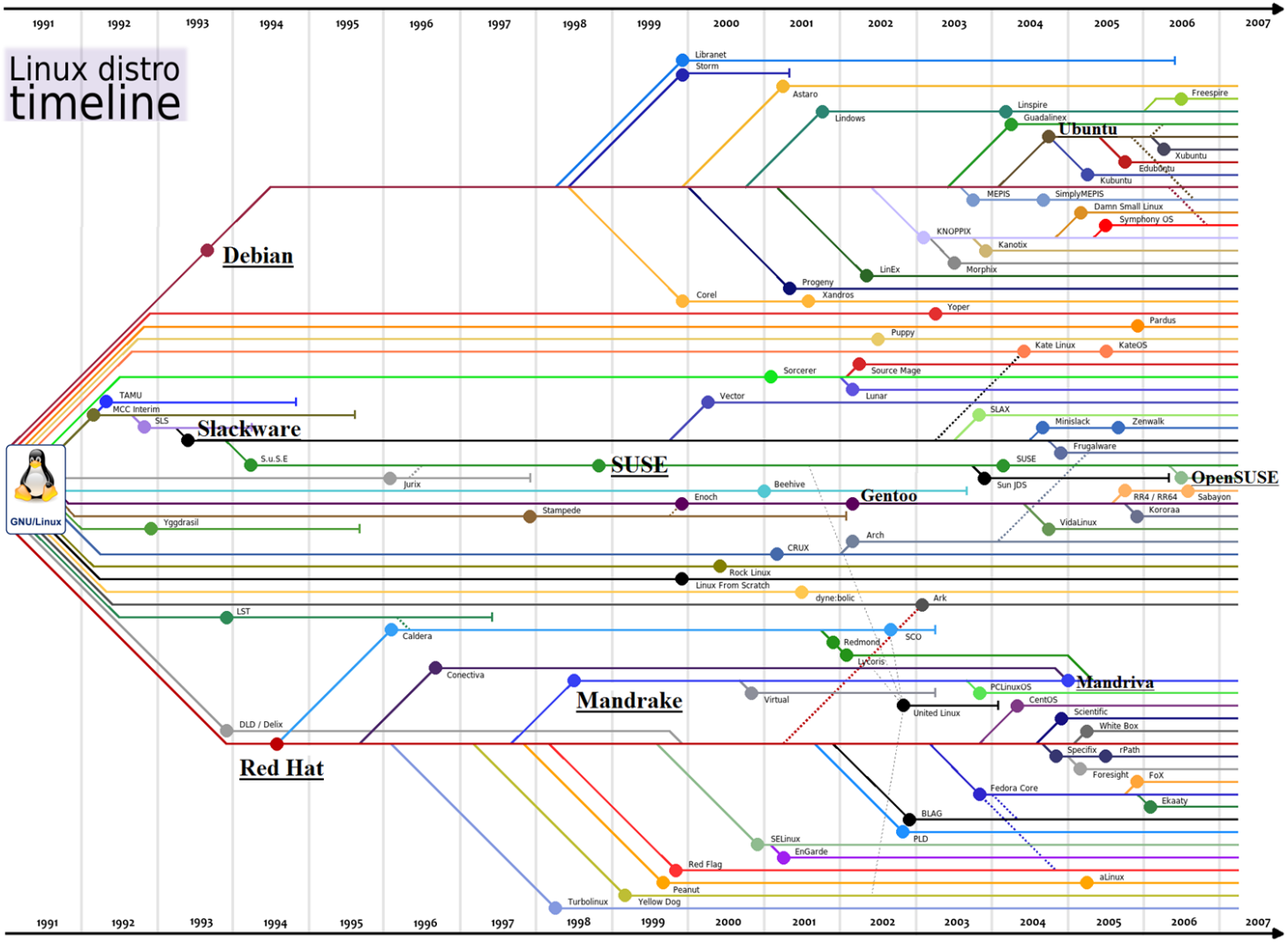
ХРОНОЛОГИЯ СОБЫТИЙ

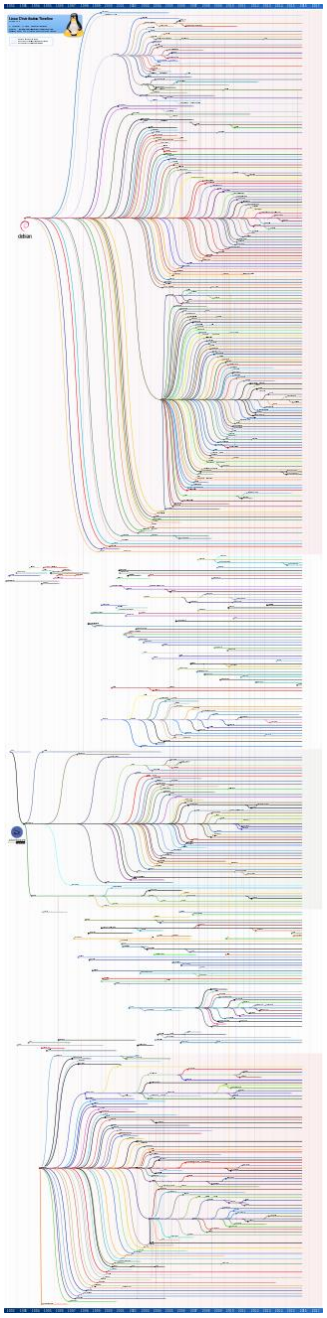
- Первые версии Linux создавались на базе Minix
- Файлы первой версии Linux (Linux 0.01) были опубликованы в Internet 17 сентября 1991г.
- Рабочая версия Linux 0.02 была опубликована 5 октября 1991г.

Фонд Свободного Программного Обеспечения

- 1984г. – основание Фонда Свободного Программного обеспечения (Free Software Foundation)
- 1985г. – начало разработки проекта GNU
- 1988г. – создание в рамках GNU лицензии GPL

ДИСТРИБУТИВЫ





ЛИЦЕНЗИИ ПРОЕКТА GNU\LINUX

GPL (***G**eneral **P**ublic **L**icense*)

Предоставляет права на копирование, модификацию и распространение; наследуется.

BSD (***B**erkeley **S**oftware **D**istribution*)

Предоставляет права на копирование, модификацию и распространение; наследуют Berkeley TM.

MIT (*MIT License*)

Предоставляет права на копирование, модификацию и распространение без обязательств; не предполагает наследование лицензии.

СТАНДАРТ POSIX

POSIX – набор стандартов и рекомендаций, описывающих унифицированный интерфейс взаимодействия приложений с ОС.

Структура:

- **Часть 1.** Основные определения
- **Часть 2.** Системные интерфейсы
- **Часть 3.** Shell и утилиты
- **Часть 4.** Пояснения и дополнения

- **IEEE Std 1003.1b-1993.** Расширения реального времени
- **IEEE Std 1003.1c-1995.** Интерфейсы потоков
- **IEEE Std 1003.1d-1999.** Доп. расширения реального времени
- **IEEE Std 1003.1j-2000.** Доп. расширения реального времени
- **IEEE Std 1003.1q-2000.** Трейсинг

ФИЛОСОФИЯ LINUX

Принципы Linux Foundation и сообщества:

- Пишите программы, которые делают одну вещь и делают ее хорошо
- Пишите совместимые программы
- Пишите программы, ориентированные прежде всего на командную среду
- Используйте проверенные методы и алгоритмы
- Используйте верифицированный и принятый в сообществе инструментарий
- Базируйтесь на ПО с открытыми лицензиями

ПРИЧИНЫ ПОПУЛЯРНОСТИ LINUX

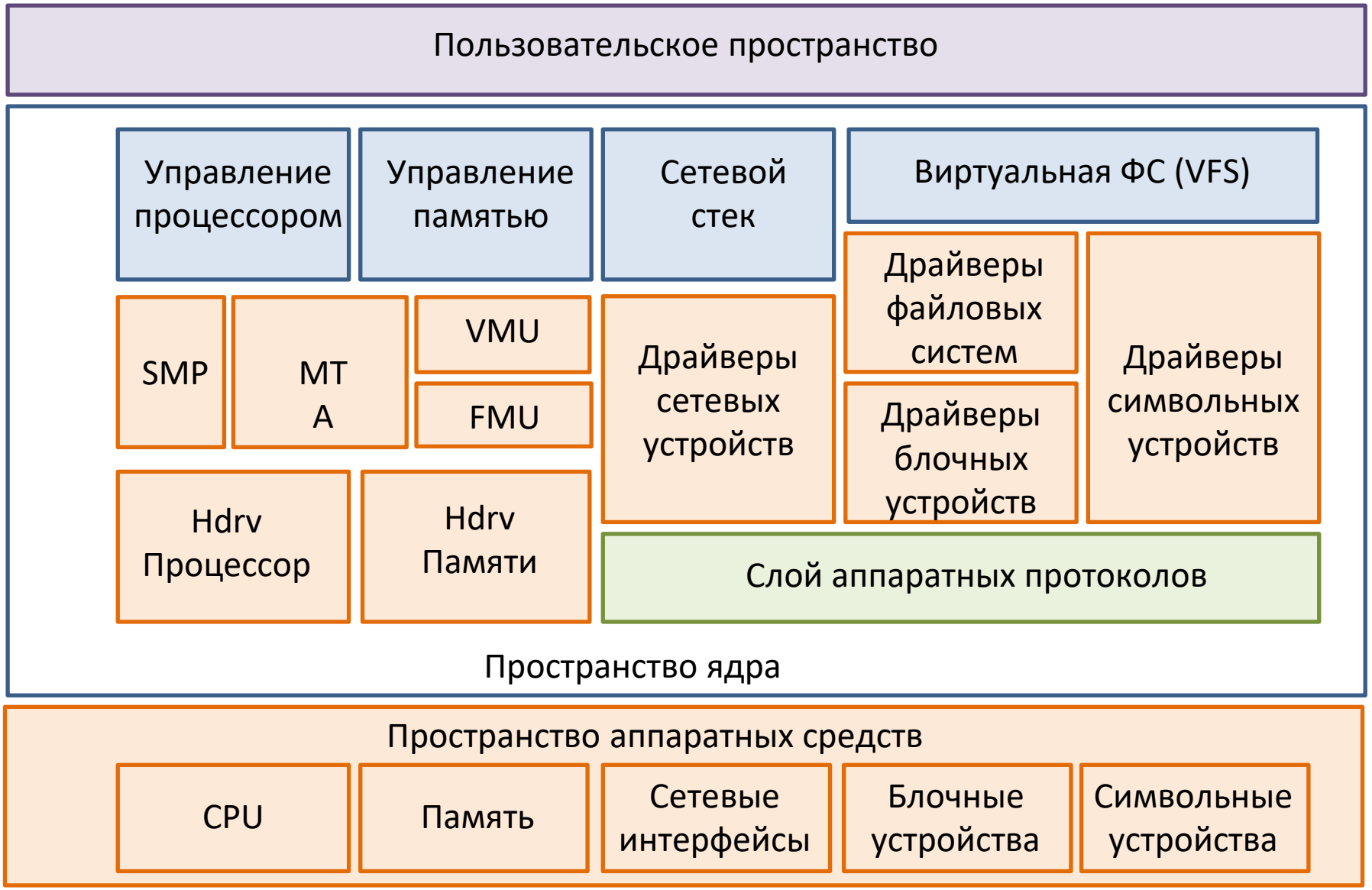
- Наличие развитого пользовательского интерфейса
- Возможность быстрого создания сложных программ из простых
- Иерархическая системная структура
- Единообразный доступ к периферийным устройствам
- Поддержка многопользовательского и однопользовательского режима
- Высокая скорость реконфигурации
- Одновременная поддержка многозадачного и однозадачного режимов
- Переносимость прикладного ПО
- Наличие унифицированных стандартов

ПРИЧИНЫ ПОПУЛЯРНОСТИ LINUX

- Наличие развитого пользовательского интерфейса
- Возможность быстрого создания сложных программ из простых
- Иерархическая системная структура
- Единообразный доступ к периферийным устройствам
- Поддержка многопользовательского и однопользовательского режима
- Высокая скорость реконфигурации
- Одновременная поддержка многозадачного и однозадачного режимов
- Переносимость прикладного ПО
- Наличие унифицированных стандартов

АРХИТЕКТУРА ОС LINUX

АРХИТЕКТУРА LINUX



КОМАНДНАЯ ОБОЛОЧКА

ЗАДАЧИ КОМАНДНОЙ ОБОЛОЧКИ

- **Часть операционной системы**
  - Скрипты инициализации
  - Системные утилиты
- **Программирование**
  - Создание пользовательских скриптов
  - Создание элементов ПО
- **Решение системных и прикладных задач**
  - Пользовательский CUI-интерфейс
  - Автоматизация команд

РАЗНОВИДНОСТИ КОМАНДНЫХ ОБОЛОЧЕК

- **sh** (Bourne shell) – ранняя командная оболочка для UNIX-систем
- **bash** (Bourne-again shell) – переработанная и модифицированная версия sh
- **csh** (C-shell) – оболочка с C-подобным синтаксисом
- **tcsh** (tiny C-shell) – версия csh с минимальным функционалом
- **pdksh** (public domain Korn shell) – bash-совместимая оболочки, применяемая в некоторых Unix-системах (AIX, BSD, QNX)
- **sash** (Stand-alone shell) – автономная оболочка, содержащая в себе все необходимые библиотеки

ОБЩИЙ ФОРМАТ КОМАНД BASH

Формат команд Bash:

```
<command> [<key-list>] [<params>]
```

- <command> – название команды, например «cat»
- <key-list> – список первичных и вторичных ключей
- <params> – список параметров команды

Пример:

```
ls -l -1 /etc
cat /etc/shadow
passwd
```

ФОРМАТЫ КЛЮЧЕЙ BASH

- Упрощенный UNIX-формат (short):

```
-<key> [value1, value2, …]
```

Пример:

```
ls -l
ls -w 10
```

- GNU-формат (full-word):

```
--<full_key_name> [=value1 value2 …]
--<full_key_name> [value1 value2 …]
```

Пример:

```
ls --long-format
ls --width=10
ls --width 10
```

ФОРМАТЫ КЛЮЧЕЙ BASH (2)

- NU-формат:

```
<key>=[value1, value2, …]
```

Пример:

```
dd if=/dev/zero
```

Ключи разных форматов могут находится одновременно в одной команде:

```
ls -l --width=10 /etc
```

Ключи без параметров упрощенного UNIX-формата могут склеиваться вместе:

```
ls -l -1 /etc    ⇔    ls -l1 /etc
```

ЭКРАНИРОВАНИЕ СИМВОЛОВ

- “ ” (*двойные кавычки*) – все символы между кавычками понимаются буквально, кроме:
  - $ – подстановка значения переменной

```
echo “CWD = ${cwd}”        CWD = /etc
```

  - ``` (*апострофы*) – подстановка результатов команд

```
echo “CUR.PATH: `pwd`”     CUR.PATH: /etc
```

- ‘ ’ (*одинарные кавычки*) – все символы между кавычками понимаются буквально
- \ (*обратный слэш*) – последующий символ экранируется

```
cat my\ file\ txt      вывести “my file txt”
```

ГРУППИРОВКА КОМАНД

- Объединение в список (;)
- Объединение в конвейер (|)
- Выделение в отдельный поток (&)
- Положительное условное выполнение (&&)
- Отрицательное условное выполнение (||)
- Объединение в атомарную группу
- Выделение в функцию
- Подстановка команд (``)

ГРУППИРОВКА КОМАНД (2)

- Объединение в список (;)

  - Объединяет команды в список с помощью символа ‘**;**’
  - Команды выполняются последовательно
  - В случае ошибки прекращается выполнение списка

Пример:

```
echo “ETC CONTENT” ; ls -l /home ; echo “===========”
```

```
linux-ge03:/etc # echo "ETC CONTENT" ; ls -l /home ; echo "==============="
ETC CONTENT
total 24
drwx------  4 root    root   4096 Jun  5 16:28 .Trash-0
drwx------  2 root    root  16384 May 31 22:43 lost+found
drwxr-xr-x 22 student users  4096 Jun  5 16:28 student
===============
```

Задание:

- Создать список команд, создающих файл командой touch и открывающих его
- Создать список команд, который создает директорию и входит в нее

ГРУППИРОВКА КОМАНД (2)

- Объединение в конвейер (|)
  - Объединяет команды с помощью символа ‘|’
  - Команды выполняются последовательно слева направо
  - Вывод предыдущей команды подается как ввод последующей как параметр команды

Пример:

```
ls /etc | grep .conf | wc -l
```

```
linux-ge03:/etc # ls /etc | grep .conf | wc -l
75
linux-ge03:/etc #
```

ВСТРОЕННЫЕ КОМАНДЫ BASH

Задание:

- Создать конвейер команд, считающий количество символов в файле
- Создать конвейер, считающий количество процессов, присоединенных к терминалам
- Вывести на экран данные о состоянии http-демона в формате ps

Список используемых команд:

- **ps –ef** – вывод состояния демонов
- **grep Str**– фильтрация строк, содержащих подстроку Str
- **head –n N** – вырезать первые N строк

ГРУППИРОВКА КОМАНД (2)

- Выделение в отдельный поток (&)
  - Команда, которая выделяется символом ‘**&**’ выделяется в отдельный поток
  - Команда завершается асинхронно

Пример:

Рекурсивный обход всей файловой системы и запись всех имен файлов в 1.txt

```
ls -R / > 1.txt &
```

```
linux-ge03:/etc # ls -R  / > 1.txt &
[2] 9776
linux-ge03:/etc # cat 1.txt | wc -l
391260
```

ГРУППИРОВКА КОМАНД (2)

Задание:

Записать в файл размеры всех файлов в файловой системе

Список используемых команд:

- **du --human-readable –total <Folder>** – вывод размера файлов

ГРУППИРОВКА КОМАНД (2)

- Положительное условное выполнение(&&)
  - Команды объединяются друг с другом с помощью символов &&
  - Последующая команда выполняется только в случае успешного завершения предыдущей

Пример:

Создание папки и переход в нее с выводом на экран сообщения «ОК»

```
mkdir ~/mydir && cd ~/mydir && echo “OK”
```

```
student@linux-ge03:~> mkdir ~/mydir && cd ~/mydir && echo "OK"
OK
student@linux-ge03:~/mydir>
```

ГРУППИРОВКА КОМАНД (2)

- Отрицательное условное выполнение(| |)
  - Команды объединяются друг с другом с помощью символов | |
  - Последующая команда выполняется только в случае не успешного завершения предыдущей

Пример:

Рекурсивный обход всей файловой системы и запись всех имен файлов в 1.txt

```
rm –r ~/mydir || echo “Is not deleted”
```

```
student@linux-ge03:~/mydir> rm -r ~/mydir || echo "Is not deleted"
rm: cannot remove '/home/student/mydir': No such file or directory
Is not deleted
student@linux-ge03:~/mydir>
```

ГРУППИРОВКА КОМАНД (2)

- Объединение в атомарную группу
  - Команды объединяются с помощью символов “**(**” и ”**)**”
  - Объединенные команды выполняются совместно как одна команда

Пример:

Создание папки ~/2. С выводом сообщений на экран

```
mkdir ~/2 && (echo “Dir created” ; echo “OK”)
```

ГРУППИРОВКА КОМАНД (2)

Задание:

- Создать группу команд, которая удаляет заданную директорию и выводит результат на экран
- Каков будет результат команды?

```
touch /root/1.txt && echo “Created” ; cat /root/1.txt
```

ГРУППИРОВКА КОМАНД (2)

- Выделение в функцию

  - Команды объединяются с помощью кл. последовательности

    “function <Func_Name> { <Func_Body> }”

  - Вызов команд производится с помощью имени <Func_Name>
  - Все команды отделяются друг от друга символами «;»

Пример:

Вывод на экран состояния процессов httpd и cron.

```
function func                    #Определение группы
{
echo “START”;
ps –ef | head –n 1;
ps –ef | grep httpd;
ps –ef | grep cron;
}
Func                             #Вызов группы
```

ГРУППИРОВКА КОМАНД (2)

```
student@linux-ge03:~> function func
> {
> echo "STRT";
> ps -ef | head -n 1;
> ps -ef | grep httpd;
> ps -ef | grep cron;
> }
student@linux-ge03:~> func
STRT
UID        PID  PPID  C STIME TTY          TIME CMD
student  10457 10353  0 20:17 pts/1    00:00:00 grep --color=auto httpd
root      1552     1  0 Jun04 ?        00:00:00 /usr/sbin/cron -n
student  10459 10353  0 20:17 pts/1    00:00:00 grep --color=auto cron
student@linux-ge03:~>
```

ГРУППИРОВКА КОМАНД (2)

Задание:

- Создать группу команд, которая бы выводила данные о демоне httpd и crond в форматирванном виде и считала их количество

ГРУППИРОВКА КОМАНД (2)

- Подстановка команд

  - Команды вкладываются одна в другую и с помощью блока ` . . .`
  - Команда, находящаяся в блоке `` выполянется в дочерней оболочке
  - Вложенность может быть неограниченной

Пример:

Вывести содержимое всех *.conf-файлов на экран

```
cat `ls /etc | grep .conf`
```

ПЕРЕМЕННЫЕ ОКРУЖЕНИЯ

Переменные

Локальные

- Не доступны дочерним процессам
- Выставляются текущим процессом

Глобальные

- Доступны дочерним процессам
- Выставляются родительскими процессами

Назначение переменных окружения:

- Хранение локальных значений
- Передача значений и параметров дочерним процессам
- Упрощение работы с командной строкой

Переменные могут перемещаться из локального контекста в глобальный с помощью команды **export**

РАБОТА С ПЕРЕМЕННЫМИ BASH

- **Инициализация и определение**

Формат присвоения:

```
<Var_Name>=<Value>
```

Пример:

```
cwd="/home/student"
```

Пробелы между именем переменной и значением не ставятся:

```
#1. Ошибка
MyVar = MyValue

#2. Попытка выполнить команду "MyVar" с параметром "MyValue"
MyVar =MyValue

#3. Присвоение MyVar="" и выполенение команды "MyValue"
MyVar= MyValue
```

РАБОТА С ПЕРЕМЕННЫМИ BASH (2)

Разыменование переменной

Формат присвоения:

```
$<Variable_Name>
${<Variable_Name>}
```

Пример:

```
cwd="/etc"                    #Инициализация cwd
echo "CWD-> $cwd"             #Выведет: CWD-> /etc
ls ${cwd}                     #Выполнит: ls /etc
```

Для обращения всегда используется символ разыменования **$**:

```
echo cwd              #Выведет имя переменной "cwd"
echo $cwd             #Выведет значение /etc
```

РАБОТА С ПЕРЕМЕННЫМИ BASH (3)

Изменение контекста переменной

Формат:

```
export <Var_Name>
export <Var_Name>=<Value>
```

Пример:

```
cwd=/etc              #Инициализация cwd
export cwd            #Перевод cwd глоб. контекст
export cwd=/etc       #Инициализация cwd и перевод в
                      #глоб. контекст
```

Переменную можно убрать из глобального контекста с помощью команды unset.

РАБОТА С ПЕРЕМЕННЫМИ BASH (3)

- **Закрепление значения переменной**

Формат:

```
readonly <Var_Name>
readonly <Var_Name>=<Value>
```

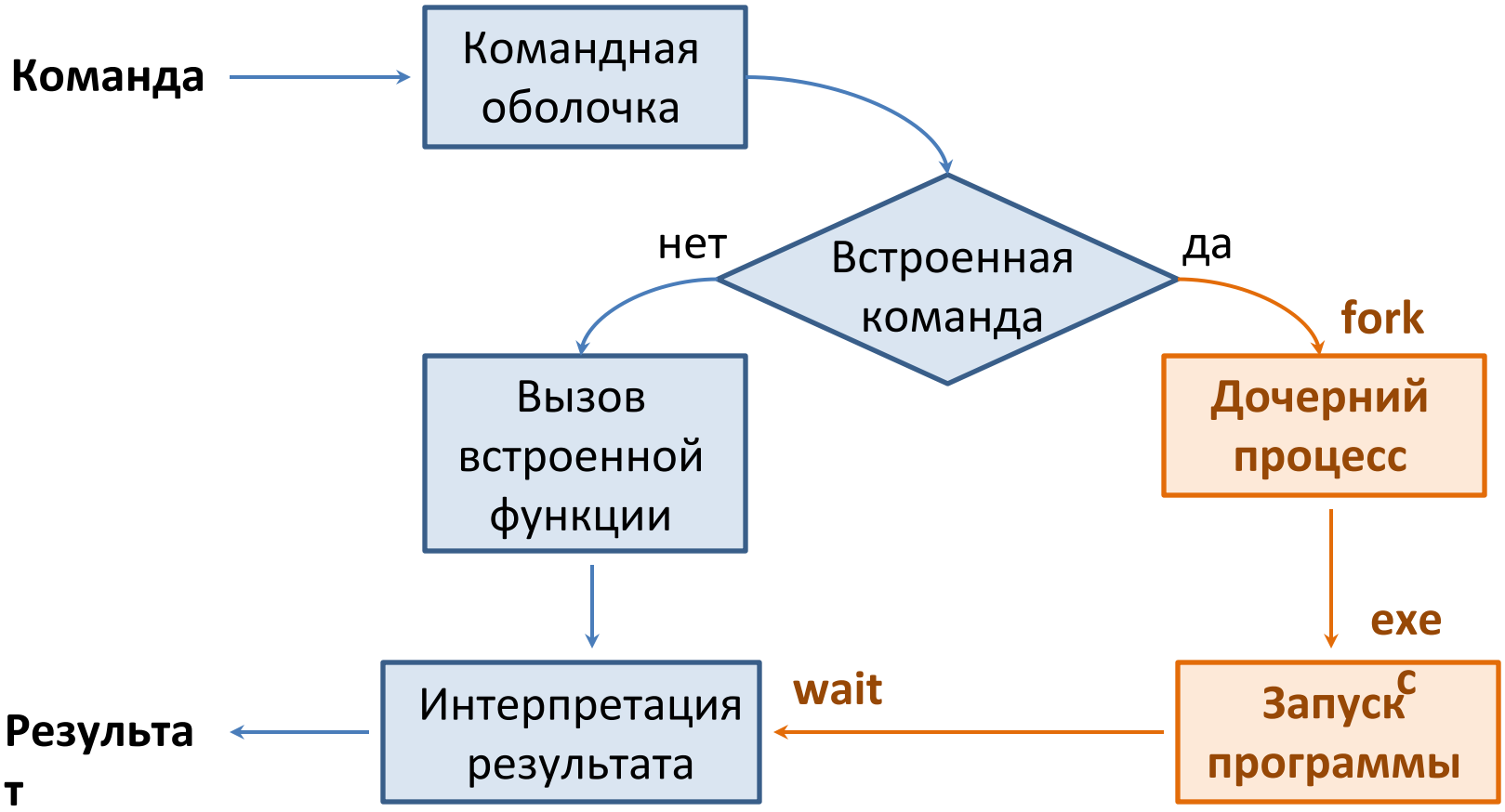
Пример:

```
cwd=/etc              #Инициализация cwd
readonly cwd          #закрепление значения cwd
readonly cwd=/etc     #Инициализация cwd и закрепление
                      #ее значения
```

ТИПЫ КОМАНД BASH

Встроенные команды (echo, pwd, alias, …)

Внешние команды



ВСТРОЕННЫЕ КОМАНДЫ BASH

- **Alias** – назначение псевдонима для команды или группы команд

Формат:

```
alias <Alias_Name>=<Long-Long-Comand>
```

Пример:

```
alias rmdir='rm -i --recursive'
rmdir /myfolder
alias -p
```

- **Echo** – вывод последовательности на экран

Формат:

```
echo <Screen_message>
```

Пример:

```
echo "Hello USER"
```

Задание:

- Создать псевдоним для команды alias, блокирующий создание псевдонимов
- Создать псевдоним для команды ls, чтобы она выводила данные о файлах в форматированном виде
- Создать псевдоним для команды echo, заменив ее на print

ВСТРОЕННЫЕ КОМАНДЫ BASH (3)

- **Source** – вставить в точку вызова команды, содержащиеся в аргументе; в отличие от `bash <Command_File>` не создает fork

Формат:

```
source <Command_File>
```

Пример:

```
source /bin/exec/myscript
.bin/exec/myscript
```

- **Enable** – выполнить вместо встроенной команды выполнить одноименную внешнюю

Формат:

```
enable <Command_Name>
```

Пример:

```
enable -n echo
```

ВСТРОЕННЫЕ КОМАНДЫ BASH (4)

- **Eval** – конструирование и выполнение команды из кэша

Формат:

```
eval <Command_String>
```

Пример:

```
eval /home/myscript
./home/myscript
eval `echo ls -l` /etc
```

- **Let** – вычисление выражения и присвоение результата переменной

Формат:

```
let <Variable>=<Expression>
```

Пример:

```
let var=1+2
```

ПЕРЕНАПРАВЛЕНИЕ ПОТОКОВ ВВОДА/ВЫВОДА

Перенаправление потоков ввода/вывода – это передача входных и выходных байтовых потоков между файлам и процессами.

Для каждой оболочки всегда открыты 3 файла:

- 0 – stdin – дескриптор стандартного ввода
- 1 – stdout – дескриптор стандартного вывода
- 2 – stderr – дескриптор стандартной ошибки

Для открываемых файлов номера дескрипторов начинаются с 10. Дескрипторы 3-9 зарезервированы для операций с дескрипторами стандартного ввода-вывода.

ПЕРЕНАПРАВЛЕНИЕ ПОТОКОВ ВВОДА/ВЫВОДА (2)

Перенаправление вывода (>):

- Перенаправление вывода команды в файл с перезаписью

Формат:

```
<Command> > <File>
```

Пример:

```
ls -l > 1.txt
```

- Очистка содержимого

Формат:

```
: > <File>
  > <File>
```

Пример:

```
: > 1.txt
```

ПЕРЕНАПРАВЛЕНИЕ ПОТОКОВ ВВОДА/ВЫВОДА (2)

Перенаправление вывода с добавлением(>>):

- Перенаправление вывода команды в файл с добавлением

Формат:

```
<Command> >> <File>
```

Пример:

```
ls -l >> 1.txt
```

- Перенаправление stdout и stderr

Формат:

```
&> j
&> <File>
```

Пример:

```
&> 1.txt
```

ПЕРЕНАПРАВЛЕНИЕ ПОТОКОВ ВВОДА/ВЫВОДА (2)

Пример:

```
# Перенаправление вывода (stdout) в файл "filename“
1>filename
# Перенаправление вывода (stdout) в файл "filename", файл
#открывается в режиме добавления
1>>filename
# Перенаправление stderr в файл "filename“
2>filename
# Перенаправление stderr в файл "filename", файл
#открывается в режиме добавления
2>>filename
# Перенаправление stdout и stderr в файл "filename“
&>filename
```

ПЕРЕНАПРАВЛЕНИЕ ПОТОКОВ ВВОДА/ВЫВОДА (2)

Перенаправление потоков между собой:

- Перенаправление ввода от i в j

Формат:

```
i >& j
```

Пример:

```
1 >& 2
```

- Перенаправление вывода из файла

Формат:

```
<Command> < j
<Command> < <File>
```

Пример:

```
grep .d < 1.txt
```

ПЕРЕНАПРАВЛЕНИЕ ПОТОКОВ ВВОДА/ВЫВОДА (2)

Перенаправления могут соединяться между собой в одной конструкции

```
<Command> < <Input_File> > <Output_File>
```

ПЕРЕНАПРАВЛЕНИЕ ПОТОКОВ ВВОДА/ВЫВОДА (2)

Открытие и закрытие дескрипторов:

- Открытие дескриптора

  Формат:

```
i <> <File>
```

  Пример:

```
4 <> 1.txt
```

- Закрытие входного дескриптора i

  Формат:

```
i <n-
```

  Пример:

```
1 <n-
```

ПЕРЕНАПРАВЛЕНИЕ ПОТОКОВ ВВОДА/ВЫВОДА (2)

- Закрытие выходного дескриптора i

Формат:

```
i >n-
```

Пример:

```
2 >n-
```

ПЕРЕНАПРАВЛЕНИЕ ПОТОКОВ С ПОМОЩЬЮ EXEC

Используется для постоянного перенаправления потоков ввода/вывода внутри командной оболочки

```
exec <Forward_Instructions>
```

Пример:

```
exec 5 <> outFile.txt
exec 4 >& 1
exec 1 >& 5
```

ПЕРЕНАПРАВЛЕНИЕ ПОТОКОВ ВВОДА/ВЫВОДА (2)

```
exec 4 >& 1          //сохранение stdout в 4-м дескрипторе
exec 1 > outFule     //перенаправление всего вывода оболочки

. . . .

exec 1 >& 4          //восстановление stdout из 4-го дескр-ра
```