# Лабораторные работы по курсу
# Операционные системы

## Лабораторная работа 1
## Основы взаимодействия с операционной системой Linux

2021

**Оглавление**

**Список сокращений**

ОС – Операционная Система

ПК – Персональный Компьютер

GUI – Graphic User Interface – Графический интерфейс пользователя

CLI – Command Line Interface – Интерфейс командной оболочки

GNU – GNU Not Unix

IPC – Inter Process Commutation

POSIX – Portable Operating System Interface

## Введение

Уже несколько десятков лет *Linux* является основной операционной системой (ОС) для применения в сферах построения надежных высокопроизводительных вычислительных комплексов, телекоммуникационного и сетевого оборудования, разработки аппаратного и программного обеспечения (ПО), создания встраиваемых систем и т. д. Понимание внутреннего устройства ОС *Linux* и умение эффективно использовать ее возможности является неотъемлемыми для ведения профессиональной деятельности в этих и многих других сферах.

Целью лабораторной работы является освоение первичных навыков взаимодействия с ОС *Linux* через командную оболочку и разработки скриптов на языке *bash* с применением лабораторного стенда на базе одноплатного компьютера *Raspberry Pi*.

## 1. Общие сведения об операционной системе Linux

### 1.1. Краткая история

В середине 70-х годов прошлого века активное распространение получила ОС под названием «*Unix*» [1]. В то время она выделялась удобной средой для пользователя и позволяла организовывать совместную работу в прикладных программах. ОС *Unix* распространялась бесплатно, и каждая организация могла самостоятельно дополнять и модифицировать ее исходный код, создавая собственный дистрибутив для коммерческой реализации.

К 80-м годам из-за наличия большого количества коммерческих дистрибутивов на базе *Unix* появился ряд проблем. Во-первых, из-за вносимых изменений каждой организацией в свою версию ОС исчезла совместимость ПО. Во-вторых, коммерчески распространяемые дистрибутивы и программы для них имели закрытый исходный код, что не позволяло сторонним разработчикам воспользоваться уже имеющимися наработками и вынуждало их реализовывать существующие программы или отдельные функции заново.

В 1983 году Ричард Столлман основал проект *GNU* [2], целью которого было создание свободно распространяемой ОС с открытым исходным кодом. Для этой цели была создана лицензия *GPL*, которая позволяет разработчику, сохраняя за собой авторство, передавать в общественное пользование исходные коды своих программ. При этом каждый разработчик, модифицируя исходные коды под лицензией *GPL*, обязан так же распространять свои результаты под лицензией *GPL*.

В начале 90-х годов Линукс Торвальдс изучал ОС *Minix*, которая представляла собой *Unix*-подобную ОС (созданную на основе *Unix*, но не использующую ее исходный код), разработанную Эндрю Таненбаумом для обучения студентов [3]. На основе ОС *Minix* в 1991 году Линукс создал собственно ядро ОС. Оно было реализовано на языке *C* и так же представляло собой *Unix*-подобную систем. В последствии работа Линукса легла в основу проекта *GNU*, став основным ядром для множества

дистрибутивов, распространяемых под лицензией *GPL*. Итоговую ОС принято называть *GNU/Linux* или просто ОС *Linux*.

ОС *Linux* получила все преимущества от передовой на тот момент времени ОС *Unix* – поддержка сетевого стека *TCP/IP*, многопользовательский режим, многопроцессность. При этом за счет распространения под лицензией *GPL* она получила быстрое развитие усилиями программистов со всего мира, став широко применяемой ОС во многих прикладных областях. Например, среди производителей микропроцессоров является стандартной практикой портирование ОС *Linux* на вновь выпускаемые продукты.

В наши дни разработка ПО высокого и низкого уровня, и в частности, разработка ПО для встраиваемых систем, непосредственно связана с работой в одном из дистрибутивов ОС *Linux*. Чем лучше разработчик понимает возможности и особенности дистрибутива, тем более совершенное и стабильное программное обеспечение он способен реализовывать.

### 1.2. Архитектура

В общем смысле под ОС *Linux* понимается некоторый программный дистрибутив, который содержит как ядро *Linux*, так и набор программ и утилит из проекта *GNU*. Среди программ присутствуют как сервисы ОС, так и пользовательские приложения, такие как текстовые редакторы, офисные пакеты, графические редакторы и т. д. Общая архитектура любого дистрибутива представлена на Рисунке Рисунок 1 [4].

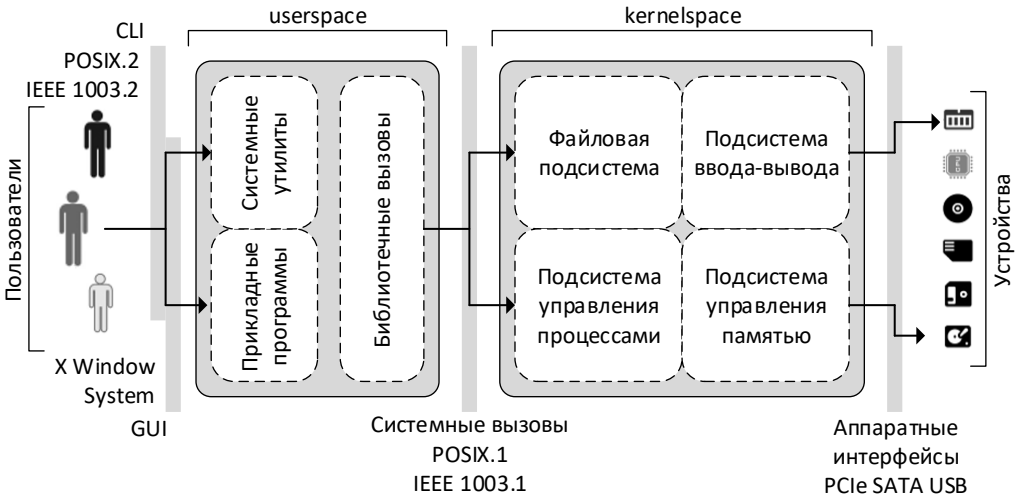



*Рисунок 1. Архитектура ОС Linux*

Ядро *Linux* – это основной компонент ОС, который состоит из набора бинарных файлов, реализующих функции управления процессами, памятью, организует доступ к файловой системе и к подсистеме ввода-вывода. В ОС *Linux* ядро имеет монолитную организацию – все его компоненты являются составными частями одной программы, исполняемой в адресном пространстве, которое принято называть «*kernelspace*». При

необходимости возможно подключение и отключение дополнительных модулей ядра в процессе работы. Адресное пространство *kernelspace* соответствует адресному пространству микропроцессора, на котором функционирует ОС, что дает ей возможность напрямую обращаться ко всем доступным периферийным устройствам. Пользовательские программы располагаются в адресном пространстве «*userspace»* и их взаимодействие с файловой системой или аппаратным обеспечением происходит через системные вызовы ядра ОС (Рисунок  Рисунок 1).

Для взаимодействия пользователя с ОС *Linux* предусмотрено два интерфейса:

- графический интерфейс (*GUI – Graphic User Interface*) и
- командная оболочка (*CLI – Command line Interface*),

о которых более подробно будет рассказано в п. 2.1.

Для обеспечения совместимости ПО при его переносе из дистрибутива в дистрибутив, в ОС *Linux* поддерживается набор стандартов, называемый *POSIX – Portable Operating System Interface* [5]. Эти стандарты описывают как интерфейс взаимодействия между ядром и программами, так и интерфейс взаимодействия между пользователями и основными системными утилитами и программами.

Серия стандартов *POSIX.1* описывает *API* (*Application Programming Interface,* программный интерфейс) между прикладными программами и ядром. В его разделах содержатся описания механизмов создания и удаления процессов, механизмов межпроцессного взаимодействия, сигналов, операций с файлами и т. д. Актуальной редакцией данного стандарта на текущий момент является *POSIX.1-2017* (*IEEE Std 1003.1-2017*).

Стандарт *POSIX.2* описывает требования к командной оболочке и системными утилитам, которые в обязательном порядке включаются в дистрибутив. Стандартизация взаимодействия пользователя с командной оболочкой и однозначное понимание разработчиком результатов работы системных утилит дает возможность гибко адаптировать разработанное ПО под различные дистрибутивы.

Среди серии *POSIX* так же существует множество более специализированных стандартов. Например *POSIX.4,* описывающий такие расширения реального времени, как планировка приоритетов, семафоры, таймеры и т. д.

В совокупности стандарты *POSIX* призваны целиком и полностью описать поведение ОС для программиста как «черного ящика» и содействовать облегчению разработки и переноса кода прикладных программ между платформами и дистрибутивами. Знание и понимание разработчиком данной серии стандартов является основной для создания высокоэффективных и стабильных приложений.

Однако так же стоит понимать, что зачастую стандарты *POSIX* могут поддерживаться в различных дистрибутивах не полностью. Например, в каждом дистрибутиве могут быть по-разному реализованы отдельные модули, драйверы периферийных устройств и т. д.  В связи с этим помимо знания стандартов каждому разработчику все-таки необходимо знать особенности дистрибутива, с которым он

работает. В особенности это важно при разработке ПО для встраиваемых систем ввиду больших расхождений между дистрибутивами различных производителей.

В рамках данного лабораторного практикума в качестве ОС *Linux* мы будем использовать дистрибутив *Raspberry Pi OS*, который был создан на основе дистрибутива *Debian* для одноплатного компьютера *Raspberry Pi* [6].

## 2. Взаимодействие с ОС

### 2.1. Способы доступа

Информационное взаимодействие между пользователем и ОС возможно двумя способами:

- через графический интерфейс (*GUI*);
- через командную оболочку (*CLI*).

Наиболее простым способом является *GUI*, отображающий на мониторе набор окон программ и использующий для ввода такие устройства, как клавиатура и мышь. В ОС *Linux* для создания графического интерфейса и управления устройствами ввода-вывода используется специальный системный программный пакет *X Windowing System*, часто называемый *X11* или просто *X*. Визуальное оформление элементов рабочего стола и окон создается, как правило, с помощью программных пакетов *GNOME* или *KDE*. Взаимодействие через *GUI* наиболее просто и удобно для рядовых пользователей, повседневно использующих приложения. Однако он неудобен для администрирования ОС и задач разработки. В связи с этим среди разработчиков более актуальным способом взаимодействия с ОС является *CLI*.

В *Linux* командную оболочку принято называть *shell*. Однако так же часто используют схожие понятие, такие как «командная строка» или «терминал». *Shell* представляет собой одно интерактивное окно, в котором пользователь может последовательно в текстовом виде вводить команды, которые возвращают результат своей работы в это же окно. За счет текстового ввода *CLI* является более гибким и предоставляет больше возможностей при меньшем количестве действий. Поэтому в каждом *GUI* в ОС *Linux* возможен вызов терминала.

Физическое взаимодействие пользователя с ОС возможно двумя способами:

- через периферийные устройства аппаратной платформы;
- удаленно по одному из цифровых интерфейсов.

В качестве периферийных устройств традиционно используются монитор как средство отображения информации и клавиатура и компьютерная мышь как средства ввода информации. Такой способ удобен при работе на персональных или одноплатных компьютерах, если существует возможность их подключения.

Однако зачастую более простым способом получить доступ к ОС, не используя дополнительное оборудование, является подключение к аппаратной платформе через один из цифровых интерфейсов. Такой способ часто применяется при работе с серверным оборудованием или при разработке встраиваемых систем. С помощью

специальных программ можно удаленно получить управление над ОС через, например, сеть *Ethernet* или *UART*. При соединении через *Ethernet* может использоваться один из стандартных протоколов, таких как *telnet* или *SSH*. Наиболее популярной программой-терминалом, запускаемой на хост-машине, является *PuTTY* [7]. Она позволяет осуществить подключение к удаленной ОС *Linux* и управлять ей из-под командной оболочки. Так же стоит отметить, что существует возможность подключения и к удаленному рабочему столу с поддержкой *GUI* через сеть *Ethernet*. Такую возможность дает, например, система *VNC* (*Virtual Network Server*). Она использует протокол *RBF* (*Remote FrameBuffer*) [8], ретранслируя отображение рабочего стола с удаленной ОС на хост-машину и перехватывает сигналы с устройств ввода от хост-машины к удаленной ОС. Однако в практике разработки встраиваемых систем данный способ используется реже ввиду его избыточности.

### 2.2. Пользователи

ОС *Linux* является многопользовательской системой – в один и тот же момент времени множество пользователей могут работать с одной ОС на одном и том же аппаратном обеспечении. Ввиду физической невозможности доступа нескольких пользователей к ОС с помощью периферийных устройств, многопользовательское взаимодействие происходит через удаленное подключение. Однако вне зависимости от способа доступа к ОС перед ее эксплуатацией пользователю необходимо пройти процедуру авторизации – ввести логин и пароль учетной записи, к которой в дальнейшем привязываются все действия в системе.

Всего в ОС *Linux* существует три типа пользователей:

1. пользователь *root*;
2. системные (фиктивные) пользователи;
3. обычные пользователи.

Пользователь *root* считается владельцем системы, и он обладает максимально возможными правами по ее администрированию и модификации. Системные пользователи создаются ОС автоматически для системных процессов, которые должны иметь права на доступ к файлам и каталогам. Обычные пользователи – это учетные записи, созданные администратором системы, которые предназначены для работы пользователей.

При подключении к ОС *Linux* через терминал первым действием является ввод имени пользователя и его пароль. При вводе пароля из соображений безопасности символы на экране терминала не отображаются. В дальнейшем при работе в терминале при каждом вводе или выводе команды в начале строки будет отображаться префикс формата «*имя_пользователя@имя_машины:*».

### 2.3. Файлы и каталоги

Все файлы и каталоги в ОС *Linux* объединены в логическую древовидную структуру. В отличии от ОС *Windows*, где может существовать несколько деревьев,

корнем каждого из которых является логических диск, в ОС *Linux* существует только один корневой каталог, который именуется «/» и называется «корень».

Всего в ОС *Linux* можно выделить два типа файлов – это регулярные файлы, содержащие информацию (текст, видео, устройства, исходный код и т.д.), и директории, которые служат для создания иерархических уровней хранения информации и в которые включаются регулярные файлы. Регулярные файлы разделяются на исполняемые и неисполняемые. К исполняемым относятся программы, которые могут быть запущены на исполнение. К неисполняемым относятся все остальные файлы. Любому файлу должно быть присвоено имя. В зависимости от дистрибутива ОС ограничение на имя файла могут быть различными. В *POSIX* в явном виде определены три правила:

1. имя файла не может быть больше длины, предусмотренной ОС (для *Linux* это 255 байт),
2. нельзя использовать символ NUL,
3. нельзя использовать символ «/».

Также на практике в именах файлов нежелательно использование символов «*», «?», «’», «”», « », «\». Все файлы, расположенные в рамках одной директории, должны иметь уникальные имена.

Для наименования и назначения директорий, расположенных в корневом, и некоторых подкаталогах, существует стандарт FHS (Filesystem Hierarchy Standard) [9]. В Таблице Таблица 1 приведено наименование основных каталогов и их назначение для дистрибутива *Raspberry Pi OS*.

*Таблица 1. Наименование и назначение каталогов в Raspberry Pi OS*

| **Каталог** | **Описание** |
|---|---|
| */* | Корневой каталог, содержащий всю файловую иерархию |
| */bin* | Основные программы и утилиты (например: *cat, ls, cp*) |
| */boot* | Загрузочные файлы (в том числе файлы загрузчика, ядро, *initrd, System.map*) |
| */dev* | Основные файлы устройств |
| */etc* | Общесистемные конфигурационные файлы (имя происходит от лат. *et cetera*) |
| */home* | Содержит домашние каталоги пользователей, которые в свою очередь содержат персональные настройки и данные пользователя |
| */lib* | Основные библиотеки, необходимые для работы программ из */bin* и */sbin* |
| */media* | Точки монтирования для сменных носителей, таких как *CD-ROM, DVD-ROM* |
| */mnt* | Содержит временно монтируемые файловые системы |
| */opt* | Дополнительное программное обеспечение |
| */proc* | Виртуальная файловая система, представляющая состояние ядра ОС и запущенных процессов в виде файлов |
| */root* | Домашний каталог пользователя *root* |
| */run* | Информация о системе с момента её загрузки, в том числе данные, необходимые для работы демонов (*pid*-файлы, *UNIX*-сокеты и т.д.) |

| | |
|---|---|
| */sbin* | Основные системные программы для администрирования и настройки системы, например, init, *iptables*, *ifconfig* |
| */srv* | Данные для сервисов, предоставляемых системой (например, *www* или *ftp*) |
| */sys* | Содержит информацию об устройствах, драйверах, а также некоторых свойствах ядра |
| */tmp* | Временные файлы (см. также */var/tmp*) |
| */usr* | *Вторичная иерархия* для данных пользователя. Содержит большинство пользовательских приложений и утилит, используемых в многопользовательском режиме. Может быть смонтирована по сети только для чтения и быть общей для нескольких машин |
| */var* | Изменяемые файлы, такие как файлы регистрации, временные почтовые файлы, файлы спулеров |

Многие дистрибутивы соответствуют данному стандарту, однако всегда могут быть исключения. Например, в *Red Hat Linux* (версии 7.3 и 8.0) каталог «*/etc/opt*» создан, но пуст, а конфигурационные каталоги пакетов размещаются непосредственно в «*/etc*».

Для понимания назначения некоторых каталогов в таблице, например «*/dev*» и «*/proc*», требуется пояснение. В ОС *Linux* принята идеология, при которой пользователь может обращаться к любой части аппаратного или программного обеспечения как к файлу. Например, внутри директории «*/dev*» представлены поддиректории и файлы, соответствующие физическим устройствам, доступным пользователю – *COM*-порт, *USB*-устройства и т. д. При обращении к файлу из «*/dev*» обмен данными будет происходить не с жестким диском, а с устройством, которому соответствует данный файл. В директории «*/proc*» содержатся файлы, доступные на чтение, которые хранят информацию о запущенных процессах и некоторые соответствующие им статистические данные. Физически файлов в «*/proc*» не существует на диске, они формируются ОС при каждом обращении.

### 2.4. Работа с командной оболочкой

После входа в ОС *Linux* через *CLI* на экране отображается текстовая строка с префиксом «*имя_пользователя@имя_машины:*». По умолчанию рабочим каталогом в котором буду создаваться новые файлы и вызываться программы на исполнение без указания пути, является «*/home/имя_пользователя/*». Взаимодействие с командной оболочкой сводится к написанию текстовых команд для вызова программ и получению их результата. Каждая программа может быть вызвана в формате

```
[символ_начала_программы]имя_программы [-опция1] […] [-опцияN] [аргумент1] […] [аргументN]
```

Вначале указывается символ начала программы. Для запуска программы в рабочем каталоге указывается «./», а для запуска программы из другого каталога указывается ее относительный или абсолютный путь в файловой системе. Затем указывается имя программы, а после него через пробел указываются опции и аргументы, если их ввод поддерживается программой. Запуск команды осуществляется нажатием клавиши «*Enter*». Обмен данными с запущенной программой возможен с помощью стандартных потоков ввода-вывода: если программа запросит данные из

стандартного потока *stdin* [10], то данные будут перехвачены из терминала, вывод программы в стандартные потоки вывода *stdout* [11] и *stderr* [12] так же буду направляться в терминал и отображаться на экране.

В *CLI* реализованы удобные для пользователя способы быстрого ввода. Например, для выбора ранее введенных программ в командную строку можно использовать клавиши «стрелка вверх» и «стрелка вниз». Для выбора имени каталога, файла или программы, начинающегося с введенных символов, можно использовать нажатие клавиши «*Tab*».

Стандарт *POSIX* определяет параметры вызова и логику работы около 150 программ и утилит. По умолчанию все стандартные программы располагаются в каталоге «/*bin*» и могут быть вызваны без символа начала программы. Не обязательно все стандартизированные программы будут содержаться в дистрибутиве ОС *Linux*. Наиболее популярные стандартные программы и утилиты указаны в Таблице Таблица 2.

*Таблица 2. Часто используемые программы*

| **Программа** | **Описание** | **Синтаксис вызова** |
|---|---|---|
| *cat* | Читает данные из файлов и выводит их в терминал | **cat** -опции файл1 файл2 |
| *cd* | Изменяет директорию с текущей на указанную | **cd** /путь/к/папке |
| *chmod* | Изменяет права/действия/пользователей для файла | **chmod** -опции -права /путь/к/файлу |
| *chown* | Передает права на файлы другим пользователям | **chown** -пользователь -опции /путь/к/файлу |
| *cp* | Копирует из файлы из одного каталога в другие | **cp -опции** /откуда /куда |
| *date* | Извлекает дату в различных форматах | **date** -опции -формат |
| *echo* | Выводит сообщение на экран | **echo** сообщение |
| *grep* | Фильтрует вывод других команд, позволяет искать по содержимому файловой системы | **grep** -опции 'поиск' /где_искать |
| *halt* | Остановка ОС | halt |
| *help* | Выводит информацию о встроенных программах | **help** имя_программы |
| *less* | Позволяет перематывать текст вперёд/назад, осуществлять поиск в обоих направлениях, переходить сразу в конец или в начало файла | **less** -опции файл |
| *locate* | Используется для поиска файлов, расположенных на машине пользователя или на сервере. Фактически она выполняет ту же работу, что и команда *find*, однако, ведёт поиск в собственной базе данных. *find* же шаг за шагом проходит через всю иерархию директорий. | **locate** опции шаблон_для_поиска |
| *ls* | Используется в командной оболочке Linux для вывода содержимого каталогов и информации о файлах | **ls** -опции /путь/к/папке |
| *man* | Формирует и выводит справочные страницы | **man** <имя_программы> |
| *mkdir* | Создает пустой каталог | **mkdir** -опции директория |
| *mv* | Переносит файлы из одного каталога в другие | **mv** -опции /откуда /куда |
| *passwd* | Задает пароль пользователя | **passwd** пароль |

| | | |
|---|---|---|
| *ps* | Вывод отчета о запущенных процессах в ОС | **ps** |
| *pwd* | Выводит полный путь до рабочей директории, в которой находится пользователь | **pwd** |
| *reboot* | Перезагрузка ОС | reboot |
| *rm* | Удаляет файлы из каталога | **rm -опции файл(ы)** |
| *rmdir* | Удаляет каталоги | **rmdir** -опции директория |
| *sleep* | Задержка выполнение на указанное время в секундах | **sleep** 5 |
| *sudo* | Выполнение программы от имени пользователя *root* | **sudo** <имя_программы> |
| *tail* | Выводит заданное количество строк с конца файла (по умолчанию 10 строк) | **tail** -опции файл |
| *tar* | Архивирует и записывает вывод в файл | **tar** -опции архив.tar файлы_для_архивации |
| *touch* | Создает файл, если его не существовало | **touch** <имя_файла> |
| *wc* | Считает количество строк или слов в тексте из стандартного ввода или файла | **wc** -опции файл |

Например, если вызвать программу «*pwd*», то на экране отобразиться путь до текущего рабочего каталога пользователя. По умолчанию для *Raspberry Pi OS* это «*/home/pi*». С помощью команды «*echo*» можно выводить на экран произвольный текст. С помощью «*cat*» можно считать содержимое файла и вывести на экран.

Одной из наиболее полезных программ является программа «*help*». Она выводит информацию о других встроенных программах, имеющихся в системе. Например, команда «*help echo*» выведет информацию о том, что делает программа *echo*, какие доступны для нее ключи и аргументы. Так же практически у каждой встроенной программы есть опциональный ключ «*[-h]*» (или «*[-help]*» или «*[--help]*»), который так же выведет на экран данную информацию. С помощью программы «*man*» можно вывести на экран справочное руководство по работе с дистрибутивом.

С помощью текстового вызова программ и получения их результата возможно полное управление ОС и пользовательскими процессами. Однако на практике возникает множество задач по последовательному вызову одних и тех же программ и анализу их результата, которые могли бы быть частично или полностью автоматизированы. Поэтому командная оболочка в ОС *Linux* представляет собой не просто строку для ввода программ, а полноценный интерпретатор специального языка создания скриптов. Существует несколько разновидностей интерпретаторов, которые могут использоваться в ОС *Linux*: «*bash*», «*dash*», «*zsh*», «*ash*» и т.д. Все они достаточно сильно схожи между собой, но традиционно самым популярным и наиболее часто используемым командным интерпретатором является *bash* [13].

Благодаря использованию интерпретатора прямо в командной строке можно вычислять арифметические выражения, обрабатывать строки, задавать условия ветвления и циклы. Если код скрипта достаточно объемный и его вызов требуется несколько раз, то скрипты записывают в специальные файлы с расширением «*.sh*».

Файлу скрипта устанавливают атрибут на исполнение и вызывают его в командной строке точно так же, как и другие исполняемые файлы.

На практике с помощью скриптового языка *bash* реализуют сценарии автозагрузки пользовательских приложений, узкопрофильные утилиты по работе с файлами, специальные процессы для мониторинга работы ОС и другие вспомогательные программы. В определенных случаях возможно применение *bash* как основного языка программирования для создания программы.

## 3. Разработка скриптов командной оболочки

### 3.1. Система контроля версий *git*

Разработка скриптов и ПО всегда представляет собой постоянный процесс внесения изменений в исходные коды. Зачастую исходный код для одной программы разрабатывается сразу несколькими людьми. Отсутствие истории и контроля вносимых изменений при работе над сложными программами может быть фатально для всего процесса разработки.

При работе над исходным кодом ядра ОС Линукс Торвальдс разработал систему контроля версий «*git*» [14], которая в настоящее время де-факто стала стандартным инструментом при разработке программного кода как для персональных компьютеров, так и встраиваемых систем. *Git* была разработана и выпущена под лицензией *GNU GPL v.2* в 2005 году. *Git*-репозиторий представляет собой директорию, в которой размещаются исходные коды. Набор утилит системы *git* позволяет вести историю изменений файлов, храня журнал изменений и комментарии к ним. За счет наличия структурированного порядка всегда имеется возможность вернуться к одной из предыдущих версий программ. При этом существует возможность создавать отдельные ветки изменений программного кода для проверки тех или иных нововведений.

Первичная инициализация репозитория *git* может быть сделана двумя способами. Первый из них – это команда

```
git init
```

Она создает в текущей директории скрытую поддиректорию «*.git*», в которой в дальнейшем будут храниться все служебные файлы. Второй способ заключается в клонировании существующего репозитория с помощью команды

```
git clone url_к_проекту
```

Данная команда скопирует все файлы из репозитория в текущую директорию.

Добавление файлов в версионный контроль осуществляется командой

```
git add файл1 файл2 … файлN
```

После добавления файлов для их фиксации необходимо выполнить команду

```
git commit -m "comment"
```

Оставленный при фиксации комментарий «*comment*» будет отображаться в истории изменений. Малоинформативный комментарий будет являться помехой при просмотре истории и поиске необходимой версии.

Остальные наиболее часто используемые команды *git* показаны в Таблице Таблица 3.

*Таблица 3. Часто используемые команды git*

| **Команда** | **Описание** |
|---|---|
| *git add* | добавление файлов в отслеживаемый индекс |
| *git status* | вывод состояния изменений в отслеживаемых файлах |
| *git diff* | вычисляет разницу между двумя деревьями |
| *git commit* | сохраняет слепок файлов из индекса в базе данных |
| *git reset* | сброс отслеживаемых файлов в индексе |
| *git rm* | удаление отслеживаемых файлов |
| *git mv* | перемещает файл, выполняя git add для нового файл и git rm для старого |
| *git clean* | удаляет «мусорные» файлы в рабочей директории |

Более подробно руководство по работе с *git* приведено в [15]. Так же применение данной системы будет подробнее рассмотрено в ходе упражнений в п. 4.

### 3.2. Основы языка *bash*

Дальнейшее повествование дает описание основных возможностей языка программирования скриптов *bash* для начального ознакомления. Более полные возможности и особенности языка приведены в официальном документе *Bash Reference Manual* [13], доступном на сайте *www.gnu.org*.

#### 3.2.1. Создание скрипта

Скрипт – последовательность действий, описанных с помощью языка программирования. Простейший скрипт на языке *bash* может быть введен прямо в командной строке с помощью разделения вызываемых программ через символ «;». Например, скрипт

**Листинг 1** – Простейший скрипт на *bash*

```
1.  pwd; whoami
```

после запуска сначала выполнит команду «*pwd*», которая выведет путь до рабочей директории, а затем выполнит команду «*whoami*», который выведет информацию о пользователе.

Данный скрипт может быть оформлен в виде отдельного файла. Для создания файла возможно использовать команду «*touch*»:

```
touch myscript.sh
```

Редактирование скрипта удобно производить с помощью одного из текстовых редакторов, например, «*vim*» или «*nano*»:

```
nano myscript.sh
```

В текстовом редакторе «*nano*» сочетание клавиш «*Ctrl+X*» и последующее нажатие клавиши «*Y*» позволяют выйти из текстового редактора с сохранением внесенных изменений. Нажатие клавиши «*N*» после сочетания «*Ctrl+X*» позволяет выйти из текстового редактора без сохранения изменений.

В Листинге 2 приведен пример исходного кода *bash*-скрипта из Листинга 1:

**Листинг 2** – Простейший скрипт на *bash*

```
1.  #!/bin/bash
2.  # This is a comment
3.  pwd
4.  whoami
```

Первой строкой указывается командная оболочка, которая будет использоваться ОС для интерпретации скрипта. Вторая строка является примером комментария в коде. Он начинается с символа «#» и продолжается до переноса строки. Строки 3 и 4 представляют собой последовательный запуск программ «*pwd*» и «*whoami*». Вызовы данных программ так же могли бы быть размещены на одной строке с использованием разделителя «*;*» между ними.

После сохранения исходного кода для запуска скрипта предварительно необходимо изменить тип файла на «исполняемый». Это можно сделать с помощью «*sudo*» и «*chmod*»:

```
sudo chmod +x myscript.sh
```

Использование «*sudo*» необходимо, потому что только пользователь *root* может изменять атрибут файла на исполнение. Теперь возможен запуск скрипта по аналогии с другими программами в системе:

```
./myscript.sh
```

После запуска скрипта ОС будет построчно исполнять исходный код, вызывая указанные в нем программы и выводя их результат на экран пользователя.

### 3.2.2. Потоки ввода-вывода

Обмен данными между программой и пользователем или программой и файловой системой или программой и программой в ОС *Linux* осуществляется с помощью «дескрипторов». Дескриптор представляет собой целое неотрицательное число, которое закрепляется ОС за потоком ввода-вывода или файлом. ОС *Linux* для каждого процесса резервирует до 9 открытых дескрипторов файлов. Первые три

дескриптора со значениями «0», «1» и «2» используются для стандартных потоков ввода-вывода:

- 0 – *stdin*, стандартный поток ввода, в него направлены вводимые символы с помощью клавиатуры,
- 1 – *stdout*, стандартный поток вывода на экран, в него направлен вывод процесса,
- 2 – *stderr*, поток для вывода на экран диагностических и отладочных сообщений.

Потоки с номерами от «3» до «9» по умолчанию не используются и могут быть задействованы пользователем по своему усмотрению.

Все потоки ввода-вывода, включая стандартные, могут быть легко перенаправлены в *bash*-скрипте. Это бывает весьма удобно в ситуациях, когда необходимо получать данные не от пользователя, а из ранее сохраненного файла, или сохранять вывод программы в файл. В Таблице Таблица 4 представлен синтаксис перенаправления потоков и даны примеры.

*Таблица 4. Синтаксис перенаправления потоков*

| **Команда** | **Описание** | **Пример** |
|---|---|---|
| *program* > <*filename*><br>*program* 1> <*filename*> | Перенаправление потока *stdout* в файл с именем *filename*. Если файла не существовало, то он создается. Если файл существовал и в нем были данные он отчищается перед записью | *echo* “*Hello*!” > *testfile*<br>*echo* “*Hello*!” 1> *testfile* |
| *program* >> <*filename*><br>*program* 1>> <*filename*> | Перенаправление потока *stdout* в файл с именем *filename*. Данные в файл дописываются в конец | *echo* “*First line*” >> *testfile*<br>*echo* “*Second line*” 1>> *testfile* |
| *program* 2> <*filename*> | Перенаправление потока *stderr* в файл с именем *filename*. Если файла не существовало, то он создается. Если файл существовал и в нем были данные он отчищается перед записью | *whoami* 2> *debugfile* |
| *program* 2>> <*filename*> | Перенаправление потока *stderr* в файл с именем *filename*. Данные в файл дописываются в конец | *pwd* 2>> *debugfile*<br>*whoami* 2>> *debugfile* |
| *program* &> <*filename*> | Перенаправление потоков *stdout* и *stderr* в файл с именем *filename* | *ps* &> *testfile* |
| *program i*>&*j* | Перенаправление файла с дескриптором *i* в дескриптор *j*. | *script.sh* 4>&5 |
| *program* < *filename*<br>*program* 0< *filename* | ввод в программу данных из файла с именем filename | *grep word* < *file_of_words*<br>*grep word* 0< *file_of_words* |
| *exec* [*i*]<>*filename* | Открыть и связать дескриптор *i* с файлом с именем *filename* на чтение и запись. | *exec* 3 <> *tmpfile* |
| *exec* [*i*] >&- | Закрыть ранее открытый дескриптор файла | *exec* 3 >&- |
| *program1* \| *program2* \| … | Неименованный канал. Передает стандартный вывод *stdout* из *program1* на стандартный ввод *stdin program2*. | *ps* \| *grep bash* |

Часто встречается комбинирование перенаправления потока ввода и потока вывода одновременно:

```
program < input-file > output-file
```

Вызываемая программа будет в стандартном входе *stdin* получать данные из «*input-file*», а свой стандартный поток *stdout* записывать в «*output-file*».

В командной строке и в *bash*-скриптах, оформленных в виде отдельного файла, поток *stdin* может быть перехвачен в переменную с помощью встроенной команды «*read*». Вызов в скрипте

```
read a
```

остановит исполнение скрипта до тех пор, пока пользователь не введет данные, которые затем будут сохранены в переменную с именем «*a*». С помощью команды «*read*» так же возможно получение данных из файла:

```
read a < somefile.txt
```

Отдельно стоит отметить передачу данных через потоки ввода-вывода с помощью неименованных каналов. Такой способ позволяет создавать длинные цепочки вызова программ, в которых выходные данные от первой программы будут входными данными для второй программы и т. д. В указанном в Таблице Таблица 4 примере первой будет исполнена программа «*ps*», которая сформирует список существующих в ОС процессов и передаст их программе «*grep*». Программа «*grep*» оставит в списке только те строки, которые содержат слово «*bash*», после чего результат будет выведен на экран.

### 3.2.3. Переменные

Язык *bash* позволяет использовать переменные для хранения информации. Все доступные переменные можно разделить на два типа:

- переменные среды;
- пользовательские переменные.

Переменные среды – это набор переменных, определенных ОС, которые доступны в *bash*-скрипте. При исполнении скрипта они будут заменены на системную информацию. Например, команда

```
echo "Home for the current user is: $HOME"
```

выведет на экран сообщение, указанное в кавычках, но при этом заменит «*$HOME*» на строку, определенную в этой переменной. Все доступные системные переменные можно посмотреть с помощью команды «*env*».

Пользовательские переменные могут объявляться, инициализироваться и использоваться в рамках кода скрипта. В Листинге 3 приведен пример инициализации, объявления и использования переменных с именами «*grade*» и «*person*». Пробелы при инициализации до и после знака «=» должны отсутствовать.

**Листинг 3** – Пример скрипта на *bash*

```
#!/bin/bash
# testing variables
grade=5
person="Adam"
echo "$person is a good boy, he is in grade $grade"
```

Пользовательские переменные так же могут перехватывать результат вывода программ в стандартный поток *stdout*. Для этого существует два способа:

1. использовать специальный символ «`» (обратный апостроф). Например,

```
mydir=`pwd`
```

2. Использовать конструкцию «*$()*». Например,

```
mydir=$(pwd)
```

После одного из таких вызов в переменной *mydir* будет храниться значение, возвращаемое программой «*pwd*».

Так же во время выполнения скрипта доступны переменные, автоматически создаваемые ОС. Эти переменные имеют имена «*$0*», «*$1*», …, «*$9*». В переменной «$0» содержится строка с именем скрипта, в переменных «*$1*» - «*$9*» содержатся аргументы, переданные при запуске скрипта. В переменной «*$#*» содержится количество переданных в скрипт аргументов.

### 3.2.4. Математические операции

В *bash* существует несколько способов выполнить математические операции. Первым способом является использование оператора «(())». Математическое выражение, помещенное внутри двойных скобок, будет вычислено и возвращено в переменную. В Листинге 4 приведены примеры:

**Листинг 4** – Пример скрипта на *bash* с использованием математических вычислений

```
A=$((1+2))                  # A = 3
B=$(( 3 + 4 ))              # B = 7
C=$(( $B + 4 ))             # C = 11
(( A++ ))                   # A = 4
(( C += 3 ))                # C = 14
D=$(( 5*6 ))                # D = 30
```

При присваивании результата операции в переменную перед двойными скобками ставится символ «*$*». При операциях, присваивающих значение в изменяемую переменную, символ «*$*» не ставится. Внутри двойных скобок допускается писать математическое выражение как с использованием, так и без использования пробелов.

Еще один способом описания математических выражений является использование встроенной функции под названием «*let*». Пример представлен в Листинге 5:

**Листинг 5** – Пример скрипта на *bash* с использованием функции let

```
1.  let A=2+3                    # A = 5
2.  let “B = $A * 2 + 4”         # B = 14
```

При отсутствии кавычек все символы выражения должны быть написаны без использования пробелов. При добавлении кавычек возможно написание выражение с пробелами.

Еще один способ описания математических выражений – это использование функции «*expr*». Она аналогична функции *let*, за исключением того, что сохраняет результат выражение не в переменную, а выводит в *stdout*. Математическое выражение после *expr* брать в кавычки не следует. При написании с кавычками выражение будет воспринято и выведено как строка. Примеры использования функции *expr* приведены в Листинге 6:

**Листинг 6** – Пример скрипта на *bash* с использованием функции expr

```
1.  expr 1 + 2                   # to stdout: 3
2.  expr “1 + 2”                 # to stdout: 1 + 2
3.  A = $( expr 1 + 2 )          # A=3
```

Обратите внимание, что в строке 3 идет перехват вывода «*expr*» из *stdout* и его сохранение в переменную точно так же, как и для других программ.

В Таблице Таблица 5 приведен полный список допустимых математических операций.

*Таблица 5. Математические операции bash*

| № | Операция | Описание |
|---|---|---|
| 1 | id++ id-- | пост-инкремент и пост-декремент переменной id |
| 2 | ++id --id | пред-инкремент и пред-декремент переменной id |
| 3 | - + | унарный минус, унарный плюс (изменение знака числа) |
| 4 | ! ~ | логическое НЕ |
| 5 | ** | возведение в степень |
| 6 | * / % | умножение, деление, остаток от деления (целочисленные операции) |
| 7 | + - | сложение, вычитание |
| 8 | << >> | битовый сдвиг влево и вправо |
| 9 | <= >= < > | операции сравнения чисел |
| 10 | == != | операции проверки равенства или неравенства чисел |
| 11 | & ^ \| | побитовое И, побитовое НЕ и побитовое ИЛИ |
| 12 | && \|\| | логическое И, логическое ИЛИ |
| 13 | a ? b : c | условный оператор выбора (если a истинно выбирается b, в противном случае c) |
| 14 | = *= /= %= += -= <<= >>= &= ^= \|= | операторы присваивания |
| 15 | a , b | запятая (поэтапное выполнение операций интерпретатором) |

Вне зависимости от выбранного способа вычисления математического выражения («(())», «*let*» или «*expr*») все операции выполняются целочисленно. Для выполнения операций с плавающей точкой может использоваться утилита «*bc*».

### 3.2.5. Условный оператор *if*

Помимо условного оператора выбора, приведенного в Таблице Таблица 5, в *bash* доступен оператор условного перехода «*if*». Его поведение аналогично поведению одноименных операторов в других языках программирования (например, *C*). В Листинге 7 приведен общий вид условного оператора:

**Листинг 7** – Пример общего вида исходного оператора

```
1.  if условие
2.  then
3.     команды
4.  else
5.     команды
6.  fi
```

Если условие истинно, то выполняются команды, заключенные между «*then*» и «*else*». Если условие ложно, то выполняются команды между «*else*» и «*fi*». Ключевое слово «*else*» может отсутствовать, тогда в случае ложности условия команды внутри «*then*» и «*fi*» исполнены не будут.

Пример использования условного оператора «*if*» приведен в Листинге 8:

**Листинг 8** – Пример использования условного оператора «*if*»

```
1.  #!/bin/bash
2.  user=anotherUser
3.  if grep $user /etc/passwd
4.  then
5.     echo "The user $user exists"
6.  else
7.     echo "The user $user doesn't exist"
8.  fi
```

В примере если программа *grep* выдаст в *stdout* какие-то данные, то на экран выведется сообщение о том, что пользователь существует. В противном случае на экран выведется сообщение о несуществовании данного пользователя.

Числовые значения в операторе «*if*» сравниваются отлично от математических операций сравнения, приведенных в Таблице Таблица 5. В Таблице Таблица 6 приведены выражения для сравнения числовых переменных, используемые в условном операторе:

*Таблица 6. Операции сравнения числовых переменных*

| № | Выражение | Описание |
|---|---|---|
| 1 | $n1 -eq $n2 | Возвращает истинное значение, если $n1 равно $n2 |

| | | |
|---|---|---|
| 2 | $n1 -ge $n2 | Возвращает истинное значение, если $n1больше или равно $n2 |
| 3 | $n1 -gt $n2 | Возвращает истинное значение, если $n1 больше $n2 |
| 4 | $n1 -le $n2 | Возвращает истинное значение, если $n1меньше или равно $n2 |
| 5 | $n1 -lt $n2 | Возвращает истинное значение, если $n1 меньше $n2 |
| 6 | $n1 -ne $n2 | Возвращает истинное значение, если $n1не равно $n2 |

Используемое в операторе выражение должно быть помещено в скобки «[]». Пример приведен в Листинге 9:

**Листинг 9** – Пример использования условного оператора «*if*»

```
1.  #!/bin/bash
2.  val1=6
3.  if [ $val1 -gt 5 ]
4.  then
5.      echo "The test value $val1 is greater than 5"
6.  else
7.      echo "The test value $val1 is not greater than 5"
8.  fi
```

Использование операторов сравнения, указанных в Таблице Таблица 5, будет воспринято интерпретатором *bash* как сравнение строк: операторы «>», «<» будут сравнивать переменные по величине *ASCII*-кодов. При этом данные операторы необходимо экранировать в тексте скрипта с помощью символа «\» («\>», «\<») ввиду того, что по умолчанию интерпретатор воспринимает их как перенаправление потока ввода-вывода.

Для проверки длины строки используются префиксы «*-n*» (возвращает истину, если длина строки больше нуля) и «*-z*» (возвращает истину, если длина строки равна нулю). Пример использования условного оператора «*if*» со сравнением строк приведен в Листинге 10:

**Листинг 10** – Пример использования условного оператора «*if*» со сравнением строк

```
1.  #!/bin/bash
2.  login ="pi"
3.  if [-n $login]
4.  then
5.            if [$login = $USER]
6.            then
7.                      echo "The user $user is the current logged in user"
8.            fi
9.  fi
```

В первом операторе «*if*» происходит проверка длинны строки в переменной «*$login*», во втором операторе «*if*» ее значение сравнивается со значением переменной среды «*$USER*».

### 3.2.6. Оператор выбора

В *bash* присутствует оператор выбора «*case*», который заменяет собой многократно вложенную конструкцию «*if-else-if-else-…*». Его синтаксис приведен в Листинге 11:

**Листинг 11** – Пример использования оператора «*case*»

```
1.  case “переменная” in
2.          “значение1”)
3.                  команды
4.          ;;
5.          “значение2”)
6.                  команды
7.          ;;
8.  ecas
```

В поле «значение» могут использоваться логические и регулярные выражения.

### 3.2.7. Циклы

В скриптах *bash* возможно создание циклов для вызова команд внутри тела цикла. Для этого по аналогии с другими языками программирования используются операторы «*for*» и «*while*». В Листинге 12 приведен общий синтаксис использования оператора «*for*»:

**Листинг 12** – Пример использования оператора «*for*»

```
1.  for (( начальное значение переменной ; условие окончания цикла; изменение переменной ))
2.  do
3.          команды
4.  done
```

В Листинге 13 приведен пример цикла от 1 до 10:

**Листинг 13** – Пример использования оператора «*for*»

```
1.  #!/bin/bash
2.  for (( i=1; i <= 10; i++ ))
3.  do
4.     echo "number is $i"
5   done
```

Синтаксис оператора «*while*» приведен в Листинге 14:

**Листинг 14** – Синтаксис использования оператора «*while*»

```
1.  while условие
2.  do
3.          команды
4.  done
```

В Листинге 15 приведен пример использования оператора «*while*»:

**Листинг 15** – Пример использования оператора «*while*»

```
1.  #!/bin/bash
2.  var=5
3.  while [ $var -gt 0 ]
4.  do
5.     echo $var
6.     (( --var ))
7.  done
```

Он будет выполняться до тех пор, пока переменная «*$var*» больше нуля. На каждой итерации значение переменной будет выведено на экран.

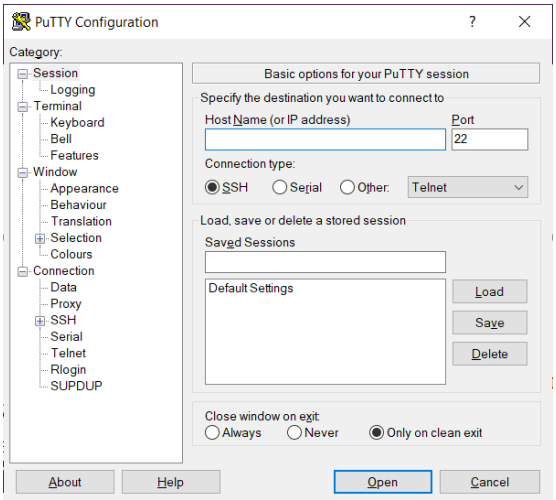
## 4. Упражнения

4.1. Убедитесь, что лабораторный стенд на базе Raspberry Pi подключен к ПК, на лабораторный стенд подано питание.

**4.2. Установка удаленного подключения между ПК и лабораторным стендом**

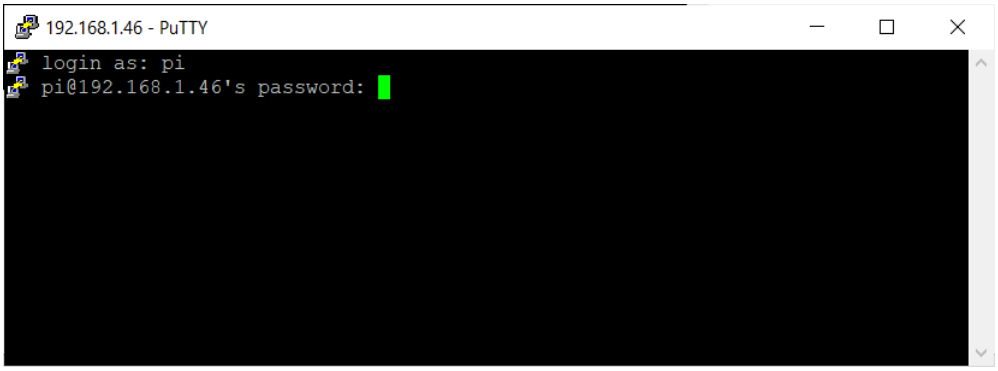
4.2.1. на ПК запустите программу-терминал PuTTY с помощью ярлыка на рабочем столе или с помощью меню Пуск;


4.2.2. в появившемся окне *PuTTY* введите *IP*-адрес *Raspberry Pi*, который сообщит преподаватель. Выберете тип подключения «*SSH*», порт «22» и нажмите кнопку «Соединиться» («Open»);

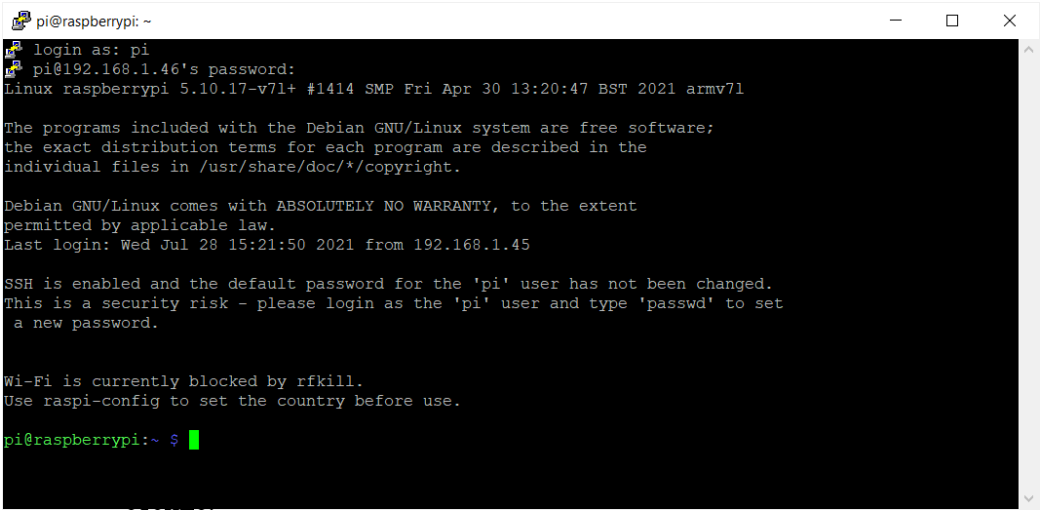



4.2.3. при первом подключении появится всплывающее окно, которое уведомит об отсутствии данного подключения в памяти. Нажмите кнопку «*Accept*» («Принять»);

4.2.4. в появившемся окне введите логин «*pi*», пароль «*raspberry*». Обратите внимание, что пароль при вводе отображаться не будет.




4.2.5. После успешного ввода логина и пароля терминал *PuTTY* станет командной оболочкой для *Raspberry Pi*, установленного в лабораторном стенде.




### 4.3. Ввод простых команд

4.3.1. введите команду «*pwd*», она выведет текущий рабочий каталог:

4.3.2.


4.3.3. введите команду

```
ls /
```

для вывода всех поддиректорий коревой директории.

4.3.4. введите команду

```
ps
```

для просмотра процессов, запущенных пользователем «*pi*», а затем команду

```
ps -aux
```

для просмотра всех процессов, запущенных в ОС;

4.3.5. Введите команду

```
ps --help
```

чтобы ознакомится с используемыми аргументами.

4.4. В рабочем каталоге создайте подкаталог для лабораторных работ с помощью команды «*mkdir*» и перейдите в него с помощью команды «*cd*», название каталога должно содержать номер группы, имя и фамилию:

```
mkdir IVT31_Ivanov_Ivan
cd IVT31_Ivanov_Ivan/
```

После перехода название каталога будет отображаться в командной строке:

```
pi@raspberrypi:~/IVT31_Ivanov_Ivan $
```

4.5. Создайте *git* репозиторий в рабочей папке и настройте данные об авторе:

```
git init
git config --global user.name "Ivan Ivanov"
git config --global user.email "pi@raspberry.pi"
```

4.6. Создайте в рабочем каталоге подкаталог с названием «*lab1*» и перейдите в него:

```
mkdir lab1
cd lab1/
```

4.7. Создайте свой первый *bash*-скрипт в командной строке:

```
echo "Hello, $USER!"
```

Программа *echo* выводит в стандартный вывод фразу, переданную в кавычках в качестве аргумента. Командная оболочка подставляет значение системной переменной *$USER*. На экране должна появиться строчка:

```
«Hello, pi!»
```

4.8. Напишите еще один скрипт в командной строке

```
whoami ; pwd
```

Данный скрипт последовательно запускает программы «*whoami*» и «*pwd*». Стандартный вывод (stdout) данных программ отображается в терминале построчно:

```
pi
/home/pi/IVT31_Ivanov_Ivan/lab1
```

4.9. Создайте с помощью команды «*touch*» файл «*draft.sh*» для *bash*-скрипта и присвойте ему права на исполнения:

```
touch draft.sh
sudo chmod +x draft.sh
```

4.10. Добавьте созданный файл в репозиторий *git* и сделайте коммит:

```
git add draft.sh
git commit -m "Initial commit"
```

В терминале появился следующая информация:

```
[master (root-commit) 7e617be] Initial commit
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100755 lab1/draft.sh
```

4.11. Убедитесь, что коммит имеется в репозитории с помощью команды:

```
git log
```

Ее вывод будет отображать сделанный коммит:

```
commit 7e617bec98224ebffdd7941ecc7f6702e0fb170f (HEAD -> master)
Author: Ivan Ivanov <pi@raspberry.pi>
Date:   Wed Jul 28 19:34:23 2021 +0100

    Initial commit
```

4.12. Откройте созданный файл «*draft.sh*» с помощью текстового редактора «*nano*»

```
nano draft.sh
```

и добавьте в него следующие строчки:

```
#!/bin/bash

whoami
pwd
```

После этого закройте файл с сохранением изменений (*Ctrl+X -> Y -> Enter*).

4.13. Убедитесь, что данные сохранились в файле с помощью команды

```
cat draft.sh
```

Если сохранение было успешно, то содержимое файла будет выведено на экран.

4.14. Выполните скрипт с помощью команды

```
./draft.sh
```

и убедитесь, что результат его работы соответствует

```
pi
/home/pi/IVT31_Ivanov_Ivan/lab1
```

4.15. Обновите измененный файл в системе контроля версий *git*:

```
git add draft.sh
git commit -m "First script"
```

На экране появится:

```
[master 8600ece] First script
 1 file changed, 5 insertions(+)
```

```
pi@raspberrypi:~/IVT31_Ivanov_Ivan/lab1 $ git log
commit 8600ecee323c24256fe16f31a869166df0aed041 (HEAD -> master)
Author: Ivan Ivanov <pi@raspberry.pi>
Date:   Wed Jul 28 19:41:00 2021 +0100

    First script

commit 7e617bec98224ebffdd7941ecc7f6702e0fb170f
Author: Ivan Ivanov <pi@raspberry.pi>
Date:   Wed Jul 28 19:34:23 2021 +0100

    Initial commit
```

4.16. Обновите скрипт «*draft.sh*», заменив его исходный код на:

```
#!/bin/bash

dir=`pwd`
curdate=`date`
echo ""
echo "Hello, $USER! You are in $dir directory"
echo "My name is $0"
echo "Now $curdate"
echo ""

if [ $# -eq 2 ]
then
	let sum=$1+$2
	let diff="$1 - $2"
	echo "First input argument is $1"
	echo "Second input argument is $2"
	echo "Their sum=$sum, their diff=$diff"
else
	echo "error: input argument must be greater than 2" 1>&2
	exit 1
fi

echo ""

for (( i=$1 ; i<$2 ; i+=1 ))
do
	echo "It is $i line..."
done
```

В приведенном выше исходном коде строки 3-9 являются примером использования системной переменной «*$USER*», локальной переменной «*$0*», содержащей название скрипта, и способа перехвата вывода программ «*pwd*» и «*date*» в переменные. Эти строки с помощью «*echo*» выводят в *stdout* информационное сообщение, которое появится на экране после запуска скрипта. Строки 11 – 21 представляют собой пример использования условного оператора «*if*». В условии проверяется количество входных аргументов, если оно не равно 2, то выводится сообщение об ошибке. Обратите внимание, что в сообщении об ошибке стандартный вывод *stdout* «*echo*» с помощью конструкции «1>&2» перенаправляется в стандартный вывод *stderr*. Если количество аргументов соответствует двум, то в *stdout* выводится их сумма и разность. Строки 25 – 28 представляют собой пример использования цикла «*for*». В данном цикле последовательно выводятся пронумерованные строки.

4.17. Сохраните скрипт и добавьте в репозиторий *git*.

4.18. Запустите скрипт с помощью команды:

```
./draft.sh
```

В терминале появится вывод:

```
Hello, pi! You are in /home/pi/IVT31_Ivanov_Ivan/lab1 directory
My name is ./draft.sh
Now Wed 28 Jul 20:35:59 BST 2021

error: input argument must be greater than 2
```

4.19. Выполните команду запуска скрипта с перенаправлением вывода в файлы. Стандартный поток *stdout* будет перенаправлен в файл «*result.txt*», а стандартный поток *stderr* в файл «*errlog.txt*»:

```
./draft.sh 1> result.txt 2> errlog.txt
```

4.20. С помощью команд

```
cat result.txt
cat errlog.txt
```

выведите содержимое файлов на экран и убедитесь, что они содержат разные части стандартных выводов.

4.21. Запустите скрипт с помощью команды

```
./draft.sh 1 5
```

Проверьте вывод скрипта:

```
Hello, pi! You are in /home/pi/IVT31_Ivanov_Ivan/lab1 directory
My name is ./draft.sh
Now Wed 28 Jul 20:54:11 BST 2021

First input argument is 1
Second input argument is 5
Their sum=6, their diff=-4

It is 1 line...
It is 2 line...
It is 3 line...
It is 4 line...
```

4.22. Для проверки работоспособности системы контроля версий выполните команду

```
git log
```

в выводе которой найдите число в шестнадцатеричной системе, которое соответствует второму коммиту:

```
commit 407cdc904dfea5ad8f717e43f1828b1725c54724 (HEAD -> master)
Author: Ivan Ivanov <pi@raspberry.pi>
Date:   Wed Jul 28 20:53:16 2021 +0100

    example code was added

commit 8600ecee323c24256fe16f31a869166df0aed041
Author: Ivan Ivanov <pi@raspberry.pi>
Date:   Wed Jul 28 19:41:00 2021 +0100

    First script

commit 7e617bec98224ebffdd7941ecc7f6702e0fb170f
Author: Ivan Ivanov <pi@raspberry.pi>
Date:   Wed Jul 28 19:34:23 2021 +0100
```

4.23. Скопируйте это число с помощью выделения мышью и выполните команду

```
git checkout 8600ecee323c24256fe16f31a869166df0aed041
```

4.24. С помощью команды

```
cat draft.sh
```

убедитесь, что содержимое файла соответствует п. 14

4.25. Вернитесь к последней версии скрипта с помощью команды

```
git checkout master
```

## 5. Индивидуальные задания

### 5.1. Общие требования

5.1.1. Задания из п. 5.2 – 5.4 должны быть выполнены последовательно.

5.1.2. Каждое выполненное задание должно быть сохранено с использованием системы контроля версий *git*. Допускается сохранение промежуточных вариантов.

5.1.3. Коммиты в *git* должны иметь содержательное описание произведенных изменений.

5.1.4. Для каждого выполненного задания должен быть создан файл «*README.MD*», располагающийся в одном каталоге с исходными кодами и сохраненный с помощью *git*. Данный файл должен описывать поведение работы скрипта и показывать пример запуска.

5.1.5. Конкретное задание основывается на двоичном представлении номера обучающегося в общем списке в восьмибитном формате *N*.

5.1.6. Продемонстрируйте преподавателю выполненное задание перед тем, как приступить к следующему.

5.1.7. Для ознакомления принципов работы с *GPIO* изучите Приложение 1.

### 5.2. Задание №1

Разработайте *bash*-скрипт, который имеет следующую логику работы. При нажатии на кнопку изменяется состояние светодиода:

1. если $N_0 = 0$, то при нажатой кнопке светодиод должен светиться, при отжатой кнопке гаснуть;
2. если $N_0 = 1$, то при нажатой кнопке светодиод должен гаснуть, при отжатой кнопке светиться.

### 5.3. Задание №2

Измените логику работы скрипта для реализации вывода бегущей строкой вашего номера *N* используя доступные светодиоды. Скорость смены значений – 1 секунда. Базовое направление движения – слева-направо. Номер *N* должен выводиться в двоичном формате. Реализуйте следующую логику реакции скрипта на нажатие по кнопкам 1 и 2:

1. если $N_1 = 0$, то нажатие на кнопку 1 увеличивает скорость смены значений в 2 раза, нажатие на кнопку 2 замедляет скорость смены значений в 2 раза;
2. если $N_1 = 1$, то нажатие на кнопку 1 устанавливает направление движения бит слева-направо, нажатие на кнопку 2 устанавливает направление движения бит справа-налево.

Соответствие светящихся диодов логическим уровням:

1. если $N_0 = 0$, то единичным битам в номере соответствуют светящиеся светодиоды;
2. если $N_0 = 1$, то единичным битам в номере соответствуют несветящиеся светодиоды.

### 5.4. Задание №3

Добавьте ввод аргументов в скрипт. Первый аргумент – начальная скорость переключения. Второй аргумент – начальная скорость движения. Предусмотрите контроль входных значений аргументов. При вызове скрипта без аргументов предусмотрите вывод справки на экран. Добавьте следующую логику реакции скрипта на однократное нажатие Кнопки 3:

1. если $N_2 = 0$, то вместо бегущего огня выводится счетчик, считающий с заданным ранее интервалом от *N* до 0.
2. если $N_0 = 1$, то вместо бегущего огня выводится счетчик, считающий с заданным ранее интервалом от до *N*.

## 6. Контрольные вопросы

1. В чем отличие между ядром и дистрибутивом Linux?
2. Что регламентируют стандарты *POSIX*?
3. Какие существуют способы взаимодействия с ОС Linux?
4. Какие существуют способы описания математических выражений на языке *bash*?
5. Для решения каких задач применяется создание скриптов на языке bash?
6. Для чего применяется *git*?

## 7. Список литературы


1. The Unix System [Электронный ресурс] // The UNIX System website: [сайт]. URL: https://unix.org/
2. The GNU official website [Электронный ресурс] URL: https://www.gnu.org/home.en.html
3. Таненбаум Э. Б.Х. Современные операционные системы. 4th ed. СПб.: Питер, 2015. 1120 pp.
4. Кетов Д.В. Внутреннее устройство Linux. СПб.: БВХ-Петербург, 2017. 320 pp.
5. The Austin Common Standards Revision Group [Электронный ресурс] URL: https://www.opengroup.org/austin/
6. Raspberry Pi OS Documentation [Электронный ресурс] URL: https://www.raspberrypi.org/documentation/raspbian/
7. Free client PuTTY [Электронный ресурс] URL: https://www.putty.org/
8. The Remote Framebuffer Protocol [Электронный ресурс] URL: https://datatracker.ietf.org/doc/html/rfc6143
9. Filesystem Hierarchy Standard [Электронный ресурс] URL: https://refspecs.linuxfoundation.org/FHS_3.0/fhs-3.0.pdf
10. Standard Input Definition [Электронный ресурс] URL: http://www.linfo.org/standard_input.html
11. Standard Output Definition [Электронный ресурс] URL: http://www.linfo.org/standard_output.html
12. Standard Error Definition [Электронный ресурс] URL: http://www.linfo.org/standard_error.html
13. Bash Reference Manual [Электронный ресурс] URL: https://www.gnu.org/software/bash/manual/bash.pdf
14. Официальный сайт Git [Электронный ресурс] URL: https://git-scm.com/
15. Самоучитель по git [Электронный ресурс] URL: https://git-scm.com/book/ru/v2