

МИНИОБРАЗОВАНИЯ РОССИИ
ВЛАДИВОСТОКСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ИНСТИТУТ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ И АНАЛИЗА ДАННЫХ
КАФЕДРА ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ И СИСТЕМ

ОТЧЕТ ПО УЧЕБНОЙ ОЗНАКОМИТЕЛЬНОЙ ПРАКТИКЕ

Студент
гр. БПИ-24-2



М.А. Федянин

Руководитель,
доцент,
кнд. хим. наук,
старший преподаватель каф.
ИТС



Б.К. Васильев

Владивосток 2026

МИНИОБРАЗОВАНИЯ РОССИИ
ВЛАДИВОСТОКСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ИНСТИТУТ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ И АНАЛИЗА ДАННЫХ
КАФЕДРА ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ И СИСТЕМ

ИНДИВИДУАЛЬНОЕ ЗАДАНИЕ
на учебную ознакомительную практику

Студенту гр. БПИ-24-2 Федянину Максиму Андреевичу

Срок сдачи: 18.07.2026

Задание 1. Перенести данные действующих пользователей из начальных компьютеров на эталонный компьютер.

Задание 2. Выбрать язык для написания скриптов и среду разработки, а затем установить их на эталонный компьютер.

Задание 3. Собрать и подключить новые компьютеры, клонировав в них систему с эталонного компьютера.

Задание 4. Оформить отчет и документы практики в печатном и электронном виде и представить на защиту в соответствии с требованиями и в установленные графиком практики сроки.

Руководитель,
доцент,
кнд. хим. наук,
старший преподаватель каф.
ИТС


Б.К. Васильев

Задание получил:


М.А. Федянин

КАЛЕНДАРНЫЙ**ПЛАН-ГРАФИК****(ДНЕВНИК)**

прохождения учебной практики БПИ-24-2 группы студента ВВГУ

Студент Федянин Максим Андреевич направляется для прохождения учебной практики «Учебная ознакомительная практика» в ВВГУ, кафедра ИТС, г. Владивосток.

С 15.06.2026 по 18.07.2026 г.

п/п	Содержание выполняемых работ по программе	Сроки выполнения		Заключение и оценка руководителя	Подпись руководителя
		Начало	Окончание		
1	Поиск литературы	15.06.2026	17.06.2026		
2	Выбор языка программирования и среды разработки	18.06.2026	20.06.2026		
3	Копирование пользователей на эталонный компьютер	22.06.2026	27.06.2026		
4	Установка выбранного ПО, сборка компьютеров и клонирование системы	29.06.2026	04.07.2026		
5	Подключение новых ПК к локальной сети	06.07.2026	11.07.2026		
6	Написание отчёта и остальной документации	13.07.2026	17.07.2026		
7	Защита отчета	18.07.2026	18.07.2026		

Согласовано:

Студент-практикант



М.А. Федянин

подпись

Руководитель,

доцент,

кнд. хим. наук,

старший преподаватель каф.

ИТС



Б.К. Васильев

подпись

Содержание

Введение.....	3
1 Теоретическая часть.....	4
1.1 История и архитектурные принципы работы скриптовых языков.....	4
1.2 Сравнительный анализ и выбор языка для написания скриптов.....	7
1.3 Анализ архитектуры и функционала сред разработки (IDE).....	11
1.4 Сравнительный анализ и выбор среды разработки (IDE).....	13
1.5 ПО для клонирования системы.....	15
2 Практическая часть.....	17
2.1 Подготовка эталонного компьютера.....	17
2.2 Удаление неактивных пользователей и перенос на жесткий диск.....	19
2.3 Установка языков для написания скриптов.....	20
2.4 Клонирование системы с помощью Clonezilla.....	21
2.5 Смена MAC-адресов.....	23
2.6 Развертывание кластера.....	24
Заключение.....	26
Список используемых источников.....	27

Введение

Целью учебной ознакомительной практики является формирование и развитие профессиональных навыков и умений в области исследовательской работы, формирование компетенций поиска, критического анализа и синтеза информации с применением системного подхода для решения поставленных задач. А также формирование и развитие практических навыков работы с компьютерами на операционной системе Debian Linux, созданию образа системы и её клонированию, работ по переносу пользователей, сборке и подключения пользовательского компьютера, установке различного программного обеспечения.

Для достижения цели практики были поставлены следующие задачи:

- выбрать средство для создания образа системы и её клонирования;
- проанализировать языки для написания скриптов (bash, tcl, perl, python, java);
- проанализировать интегрированные среды разработки (IDE) для написания и отладки скриптов;
- провести сравнительный анализ и обосновать выбор конкретного языка и среды разработки;
- отсортировать и скопировать пользователей на эталонный компьютер;
- установить выбранный язык и среду разработки на эталонный компьютер;
- провести сборку и подключение новых ПК;
- создать образ системы эталонного компьютера и клонировать его на остальные машины;
- проверить работоспособность новых компьютеров.

Объектом исследования в рамках отчета являются языки программирования сценариев и специализированные программные интерфейсы, обеспечивающие автоматизацию вычислений и системного администрирования.

Предметом исследования выступают архитектурные паттерны, синтаксические особенности, механизмы типизации, модели исполнения кода и инструментальные среды разработки, определяющие качества создаваемых скриптовых решений.

Отчет по практике оформляется в соответствии с СК-СТО-ТР-04-1.005-2015 «Требования к оформлению текстовой части выпускных квалификационных работ, курсовых работ (проектов), рефератов, контрольных работ, отчётов по практикам, лабораторным работам» [1].

1 Теоретическая часть

1.1 История и архитектурные принципы работы скриптовых языков

Генезис языков для написания скриптов неразрывно связан с эволюцией операционных систем и парадигм человеко-компьютерного взаимодействия. На заре компьютерной эры, в эпоху пакетной обработки данных, управление вычислительными процессами осуществлялось посредством перфокарт и жестко заданных инструкций, что исключало интерактивность.

Ситуация коренным образом изменилась с появлением операционной системы UNIX в начале 1970-х годов, внедрившей революционную концепцию командной строки и конвейера передачи данных (pipes). Первым полноценным командным интерпретатором, заложившим основы скриптового программирования, стала оболочка Томпсона (Thompson shell), замененная в 1979 году на Bourne shell (sh), разработанную Свеном Борном в Bell Labs. Оболочка Борна привнесла в командный интерфейс основные конструкции алгоритмических языков, используемые практически во всех программах – циклы, ветвления и переменные, превратив операционную среду в программируемую экосистему. Дальнейшее развитие этого направления привело к созданию в 1989 году оболочки Bash (Bourne Again Shell) в рамках проекта GNU, которая по сей день остается стандартом де-факто для автоматизации низкоуровневых системных задач в POSIX-совместимых системах [2].

Параллельно с развитием системных оболочек возникала потребность в языках, способных манипулировать сложными структурами данных и текстовой информацией без жесткой привязки к конкретным утилитам ОС. В 1987 году Ларри Уолл создал язык Perl (Practical Extraction and Report Language), который сочетал в себе синтаксис Си, семантику утилит sed и awk, и мощнейший аппарат регулярных выражений [3]. Perl ознаменовал собой эпоху «интернет-скриптинга», став главным инструментом для написания CGI-сценариев на раннем этапе развития Всемирной паутины.

В 1988 году Джон Оустерхаут представил язык Tcl (Tool Command Language), изначально задуманный как встраиваемый командный язык для систем автоматизированного проектирования (САПР) [4]. Tcl предложил уникальную концепцию «все есть строка», что обеспечило беспрецедентную простоту интеграции скриптов с бинарными библиотеками на Си. В 1991 году Гвидо ван Россум опубликовал первую версию языка Python, противопоставив избыточному синтаксису Perl философию читаемости, лаконичности и строгого структурного разделения кода, что впоследствии вывело Python в лидеры среди языков общего назначения и анализа данных [5].

Особое место в этой эволюционной цепочке занимает язык Java, представленный компанией Sun Microsystems в 1995 году. Изначально позиционируемый как компилируемый язык для создания независимых кроссплатформенных приложений, Java в течение длительного времени воспринимался как антипод скриптовых языков из-за тяжеловесного синтаксиса, строгой статической типизации и необходимости явной компиляции [6]. Однако с развитием платформы Java SE ситуация радикально изменилась. Архитекторы экосистемы Java последовательно внедряли инструменты, сближающие язык со скриптовой парадигмой. Введение спецификации JSR 223 (Scripting for the Java Platform) обеспечило бесшовную интеграцию внешних скриптовых движков внутрь Java-приложений. Революционным шагом стало появление в Java 9 утилиты JShell – интерактивной среды REPL (Read-Eval-Print Loop), позволившей исполнять выражения Java на лету без объявления классов и методов. Окончательное закрепление Java в статусе мощного инструмента для написания скриптов произошло с принятием JEP 330 (Launch Single-File Source-Code Programs) в Java 11, что позволило запускать исходные файлы .java напрямую через интерпретатор одной командой, минуя стадию ручного создания байт-кода, аналогично традиционным сценариям Bash или Python.

Для глубокого понимания природы скриптовых языков необходимо преодолеть поверхностное представление о них как о простых текстовых оболочках над системными вызовами. Архитектура современного скриптового языка представляет собой сложный программный комплекс, включающий подсистемы лексического и синтаксического анализа, генерации промежуточного представления (байт-кода), виртуальную машину исполнения, менеджер динамической памяти с автоматическим сборщиком мусора, а также обширную библиотеку базовых типов данных и системных интерфейсов [7].

Простая командная оболочка (например, базовый sh) функционирует в парадигме прямой трансляции: она считывает строку текста, разбивает её на токены, сопоставляет первый токен с системным исполняемым файлом в переменной окружения PATH, порождает новый дочерний процесс через системный вызов `fork()` и передает управление функции `exec()`. В такой модели оболочка выполняет роль лишь диспетчера процессов, а накладные расходы на постоянное создание новых процессов ОС и межпроцессное взаимодействие (IPC) делают её неприменимой для сложной алгоритмической обработки.

В отличие от примитивных оболочек, развитые скриптовые языки (Python, Perl, Java, Tcl) используют двухфазную или многофазную модель исполнения. При запуске скрипта исходный текст подвергается глубокому синтаксическому анализу (парсингу), в процессе которого строится абстрактное синтаксическое дерево (AST – Abstract Syntax Tree). На основе

AST транслятор генерирует внутреннее низкоуровневое представление – байт-код. Этот байт-код не зависит от архитектуры центрального процессора и исполняется специализированной программной средой – виртуальной машиной (например, PVM в Python или JVM в Java). Виртуальная машина инкапсулирует в себе управление регистрами, стеком вызовов и структурами данных, обеспечивая полную изоляцию исполняемого скрипта от специфики операционной системы. Благодаря этому скрипт получает возможность манипулировать сложными абстракциями данных непосредственно внутри единого адресного пространства одного процесса, без необходимости порождения тяжелых дочерних процессов ОС.

Важнейшим архитектурным отличием развитых скриптовых систем является глубинная интеграция механизмов управления памятью для предотвращения возникновения сбоев и ошибок, связанных с её переполнением. В то время как системная оболочка оперирует исключительно строковыми переменными, выделяя под них память через примитивные буферы, виртуальные машины скриптовых языков реализуют развитые стратегии динамического выделения памяти в куче (heap) и автоматического уничтожения неиспользуемых объектов. В Python для этого применяется комбинированный метод подсчета ссылок (Reference Counting) и периодического поиска циклических зависимостей генерационным сборщиком мусора. В платформе Java (JVM) задействованы сложнейшие промышленные алгоритмы сборки мусора (такие как G1, ZGC), которые минимизируют паузы в работе программы (Stop-the-World) и динамически оптимизируют распределение памяти на основе анализа времени жизни объектов. Такой подход гарантирует, что скрипт, выполняющий обработку огромных потоков корпоративных данных, защищен от утечек памяти и порчи указателей, свойственных низкоуровневым языкам разработки.

Кроме того, концепция современных скриптовых языков опирается на развитую модель выполнения контекстов и обработки исключительных ситуаций. Вместо анализа кодов возврата процессов (как это реализовано в Bash, где успешность операции проверяется через переменную \$?), развитые языки используют объектно-ориентированную модель исключений (try-catch-finally). При возникновении системной ошибки (например, отсутствии прав на чтение файла или сбое сетевого сокета) виртуальная машина прерывает линейный поток выполнения, собирает полный снимок стека вызовов (stack trace) и передает его по цепочке обработчиков. Это превращает скрипт из хрупкого набора команд в надежное отказоустойчивое приложение, способное корректно реагировать на любые инфраструктурные аномалии, закрывать дескрипторы файлов, освобождать сетевые порты и осуществлять детальное журналирование инцидентов. Эти механизмы значительно облегчают разработку программного обеспечения, а также повышают его надёжность работы.

1.2 Сравнительный анализ и выбор языка для написания скриптов

Перед установкой на эталонный компьютер в рамках учебной практики требуется проведение сравнительного анализа доступного инструментария. Для объективной оценки языков написания скриптов необходимо выработать критерии, отражающие как сугубо технические характеристики сред выполнения, так и эксплуатационные метрики, влияющие на совокупную стоимость владения (ТСО) программным комплексом. В качестве базовых критериев анализа в рамках данного исследования выдвинуты: вычислительная производительность (скорость выполнения алгоритмов и эффективность утилизации ресурсов), безопасность типов (выявление ошибок на этапе компиляции/анализа против времени выполнения), масштабируемость кодовой базы (архитектурные возможности для построения крупных систем), кроссплатформенность и поддержка экосистемы (наличие библиотек, драйверов и средств интеграции).

Рассматривая язык Bash, можно констатировать его абсолютное превосходство в узкой нише первичной инициализации операционных систем и оркестрации базовых утилит POSIX. Bash предоставляет лаконичный синтаксис для перенаправления ввода-вывода (stdout, stdin, stderr) и связывания программ в конвейеры. Однако за пределами этой ниши Bash демонстрирует критические недостатки. Полное отсутствие типизации (все переменные по сути являются строками) и специфический синтаксис делают написание сложных математических или логических алгоритмов крайне трудоемким процессом. Отсутствие встроенных механизмов модульности и структур данных (в Bash поддерживаются лишь простейшие одномерные массивы) превращает скрипты размером более нескольких сотен строк в монолитный, нечитаемый код, поддержание которого силами команды разработчиков становится экономически нецелесообразным.

Язык Tcl, несмотря на свою элегантную концепцию минимизации синтаксических правил и мощный потенциал встраивания в Си-приложения, в современных реалиях автоматизации общего назначения утерял былые позиции. Парадигма, в которой любая инструкция и любая структура данных интерпретируется как текстовая строка, накладывает колоссальную перегрузку на вычислительные процессы. Выполнение математических расчетов в Tcl требует постоянного явного преобразования типов на уровне интерпретатора, что катастрофически снижает производительность. Кроме того, экосистема Tcl весьма ограничена и сосредоточена преимущественно вокруг специфических задач автоматизации тестирования аппаратного обеспечения (сетевое оборудование, интегральных схем через расширение Exprest) и старых графических интерфейсов (Tk), что делает его непригодным для создания современных облачных или веб-ориентированных скриптов автоматизации.

Perl, долгое время носивший титул «швейцарского армейского ножа» системных администраторов, на современном этапе переживает глубокий кризис востребованности. Обладая непревзойденной скоростью обработки регулярных выражений и текстовых файлов, Perl страдает от избыточной синтаксической гибкости, сформулированной в девизе языка: «Есть более одного способа сделать это» (TMTOWTDI). Данный подход привел к тому, что код, написанный одним инженером, практически невозможно прочитать и верифицировать другому специалисту без детального полинейного разбора. Динамическая слабая типизация Perl и запутанная контекстная модель (скалярный, списочный контексты) являются источником трудно обнаруживаемых логических ошибок, проявляющихся исключительно в специфических условиях времени выполнения (runtime), что недопустимо для критически важных инфраструктурных скриптов предприятия.

Python на сегодняшний день является самым популярным языком динамического скриптинга в мире. Его сильные стороны общеизвестны: чистота синтаксиса, высокая скорость написания кода и гигантская экосистема, покрывающая сферы от системного администрирования до машинного обучения. Однако при использовании Python для построения масштабных корпоративных систем автоматизации инженеры сталкиваются с фундаментальными барьерами. Главным из них является динамическая типизация. В крупных проектах, где кодовая база скриптов пишется и модифицируется десятками сотрудников в течение нескольких лет, отсутствие жестких контрактов данных приводит к тому, что банальная опечатка в имени свойства объекта или передача неверного типа аргумента в функцию обнаруживается только в момент непосредственного выполнения данной ветки кода в продакшн-среде. Внедрение аннотаций типов (Type Hinting) и внешних статических анализаторов (муру) лишь частично решает проблему, не гарантируя абсолютной безопасности на уровне самой среды выполнения.

В этом контексте платформа Java раскрывает свой колоссальный скрытый потенциал как идеальный инструмент для написания высоконадежных, масштабируемых и производительных скриптов автоматизации. Современная Java успешно преодолела исторические стереотипы о медлительности и синтаксической громоздкости. Благодаря внедрению механизма JEP 330, позволяющего исполнять файлы .java напрямую без ручной предварительной компиляции, процесс написания и запуска скрипта на Java стал столь же быстрым и итеративным, как и на Python или Bash. При этом разработчик получает в свое распоряжение все преимущества одного из самых совершенных языков со статической типизацией. Строгая статическая типизация Java гарантирует, что более 90% потенциальных семантических и синтаксических ошибок, несоответствий типов и нарушений интерфейсов

отсекаются на этапе компиляции, выполняемой прозрачно в момент запуска скрипта виртуальной машиной.

Вычислительная производительность Java, обеспечиваемая многоуровневыми JIT-компиляторами (Just-In-Time) внутри JVM, превосходит производительность интерпретируемых языков (Python, Bash, Perl, Tcl) на несколько порядков. Виртуальная машина Java динамически анализирует профиль выполнения скрипта, выявляет «горячие точки» кода (hot spots) и компилирует их непосредственно в оптимизированный машинный код целевого процессора. Это позволяет использовать Java-скрипты для выполнения сложнейших вычислительных задач в реальном времени – агрегации гигабайтных логов, криптографической обработки потоков данных, параллельного опроса тысяч сетевых эндпоинтов – без риска упереться в ограничения производительности одного потока, как это происходит в Python из-за глобальной блокировки интерпретатора (GIL – Global Interpreter Lock).

Архитектурная модель Java предоставляет возможности для масштабирования кодовой базы автоматизации. Наличие четких абстракций (интерфейсов, абстрактных классов), пакетов и модулей позволяет структурировать скрипты автоматизации в полноценные надежные библиотеки многократного использования. Поддержка экосистемы Java (библиотеки Apache Commons, Google Guava, драйверы для любых существующих СУБД, брокеров сообщений и облачных API) гарантирует, что автоматизатор защищен от необходимости самостоятельной реализации низкоуровневых протоколов. Корпоративная поддержка и долгосрочные циклы релизов (LTS) гарантируют стабильность API: скрипт, написанный на Java сегодня, гарантированно продолжит корректно функционировать и через десять лет, что невозможно для динамически развивающихся экосистем, склонных к частым обратно-несовместимым изменениям. Сравнение вышеописанных языков представлено в таблице 1.

Таблица 1 – Сравнение языков для написания скриптов

Язык	Производительность	Типизация	Масштабируемость	Кроссплатформенность	Интеграция с ОС	Экосистема и библиотеки
Bash	Низкая для вычислительных задач	Отсутствует (переменные – строки по умолчанию)	Затруднительна из-за отсутствия ряда структур (многомерные массивы и т.д.)	Работает в большинстве платформ, иногда лишь с доп.системами	Отличная в Unix-системах: прямой доступ к файлам и окружению	Нет единого пакетного менеджера

Продолжение таблицы 1

Язык	Производительность	Типизация	Масштабируемость	Кроссплатформенность	Интеграция с ОС	Экосистема и библиотеки
TCL	Средняя; хорошо оптимизирован для встраивания и обработки событий	Динамическая	Может быть затруднительна для крупных систем ввиду средней производительности	Работает на всех основных платформах	Хорошая через расширения; часто используется как встроенный язык	Пакеты через менеджер teapot или вручную; экосистема ограничена
Perl	Средняя; уступает компилируемым языкам	Динамическая (слабо), с возможностью строгой проверки через модули	Затруднительна из-за очень гибкого синтаксиса	Работает на всех основных платформах	Отличная; исторически создан для системного администрирования	менеджер CPAN (огромный репозиторий модулей)
Python	Средняя; уступает компилируемым языкам	Динамическая с возможностью строгой проверки через модули	Могут возникать трудности из-за нестрогой типизации	Работает на всех основных платформах	Хорошая; имеются модули для взаимодействия с ОС	PuPI, одна из крупнейших экосистем и наборов библиотек
Java	Высокая; JIT-компиляция (HotSpot), оптимизация рантайма	Статическая, строгая	Могут возникать трудности при синхронизации	Работает на всех основных платформах	Посредственная для низкоуровневых задач; требует JNI или внешних вызовов	Maven, Gradle; огромная экосистема библиотек

По итогу сравнения для установки на эталонный компьютер был выбран язык Java. Данный язык обладает большей производительностью, чем остальные представленные в исследовании, он имеет большое количество библиотек, что позволяет писать программы под решения большинства задач. А благодаря кроссплатформенности, написанные программы можно использовать и в других системах с установленным Java Development Kit. Особенностью данного языка по сравнению с остальными является строгая типизация, где необходимо чётко отслеживать взаимодействия различных объектов между собой. Это

позволяет заранее выявлять ошибки семантических и синтаксических ошибок, несоответствий типов и нарушений интерфейсов ещё на стадии разработки. Поэтому данный язык подходит для создания разнообразного программного обеспечения в учебных и рабочих целях.

1.3 Анализ архитектуры и функционала сред разработки (IDE)

Качество, скорость написания и надежность скриптов автоматизации детерминированы не только выбранным языком программирования, но и характеристиками инструментальной среды, в которой инженер осуществляет проектирование, отладку и сопровождение кодовой базы. Современный инструментарий разработчика прошел длительный эволюционный путь от простейших консольных текстовых редакторов до тяжеловесных монолитных интегрированных сред разработки (IDE) и далее к гибким, компонентным, расширяемым архитектурам нового поколения. В контексте написания скриптов автоматизатору приходится работать в полиглотном режиме: один и тот же проект может включать Bash-скрипты для конфигурирования контейнеров, Python-скрипты для обработки данных, Java-файлы для основной логики автоматизации и конфигурационные файлы YAML/JSON. Это накладывает специфические требования к средам разработки, которые должны обладать мультиплатформенностью, гибкостью конфигурирования и низким порогом входа при сохранении аналитического потенциала.

Исторический пласт инструментов представлен консольными редакторами общего назначения, такими как Vim и Emacs. Базируясь на текстовом интерфейсе (CLI), эти редакторы предоставляют разработчику уникальные возможности для сверхбыстрого редактирования кода без использования манипулятора мышь, благодаря развитым системам клавиатурных комбинаций и модалным режимам работы [8]. Обладая минимальными требованиями к аппаратным ресурсам, Vim и Emacs незаменимы для экстренного редактирования скриптов непосредственно на удаленных серверах через сессии SSH. Тем не менее, превращение Vim или Emacs в полноценную среду разработки с подсветкой синтаксиса, автодополнением и возможностями отладки требует от инженера глубоких специфических знаний, написания сотен строк конфигурационных файлов (например, `.vimrc` или скриптов на Emacs Lisp) и ручной поддержки хрупких цепочек плагинов, что неэффективно в условиях дефицита времени реального производства.

На противоположном полюсе инструментального ландшафта находятся классические монолитные интегрированные среды разработки, яркими представителями которых являются платформы компании JetBrains (IntelliJ IDEA, PyCharm), а также исторические среды Eclipse и NetBeans. Архитектурная парадигма тяжелых IDE строится вокруг создания глубокого внутреннего семантического индекса всего проекта [9]. Сразу после открытия директории IDE

запускает фоновые процессы сканирования, которые строят детальные графы зависимостей, абстрактные синтаксические деревья для каждого файла и таблицы символов. Это обеспечивает непревзойденное качество автодополнения кода (IntelliSense), контекстного анализа и мощнейших инструментов автоматического рефакторинга (переименование сущностей, извлечение методов, оптимизация импортов). Однако платой за такую интеллектуальность является экстремальное потребление оперативной памяти (зачастую измеряемое гигабайтами) и высокая нагрузка на центральный процессор, что делает монолитные IDE неоптимальными для быстрого открытия единичных скриптов или работы на ноутбуках инженеров с ограниченными ресурсами.

Революционным прорывом в архитектуре сред разработки стало создание в 2015 году корпорацией Microsoft редактора Visual Studio Code (VS Code). Архитектурная концепция VS Code перевернула индустрию, предложив идеальный баланс между легковесным текстовым редактором и мощной IDE. В основе VS Code лежит программная платформа Electron, объединяющая движок рендеринга Chromium и серверную среду выполнения Node.js [10]. Это позволяет интерфейсу функционировать с высокой скоростью нативных приложений, используя современные веб-технологии для отрисовки графических элементов. Ключевым технологическим достижением, заложенным в архитектуру VS Code, является разделение среды разработки на два изолированных контура: легковесную клиентскую оболочку интерфейса и независимые фоновые серверные процессы языкового анализа, взаимодействующие по стандартизированным протоколам.

Центральным элементом этой архитектуры выступает протокол LSP (Language Server Protocol), разработанный Microsoft совместно с лидерами индустрии. До появления LSP создатели каждого нового языка программирования или авторы сред разработки были вынуждены писать уникальные плагины синтаксического анализа под каждый конкретный редактор (m языков $\times$ n редакторов = $m \times n$ плагинов). LSP кардинально изменил эту схему, стандартизовав межпрограммное взаимодействие через JSON-RPC. Теперь разработчики языка программирования создают один официальный языковой сервер (Language Server), который инкапсулирует в себе всю логику парсинга, подсветки ошибок, автодополнения и поиска определений. VS Code выступает в роли универсального клиента, который запускает необходимый сервер в отдельном процессе и обменивается с ним стандартизированными сообщениями по протоколу LSP. Благодаря этому VS Code мгновенно приобретает глубокое интеллектуальное понимание любого языка программирования (будь то Java, Python или Bash) сразу после подключения соответствующего легковесного расширения, не раздувая при этом ядро самого редактора.

Интерфейс VS Code представлен на рисунке 1.

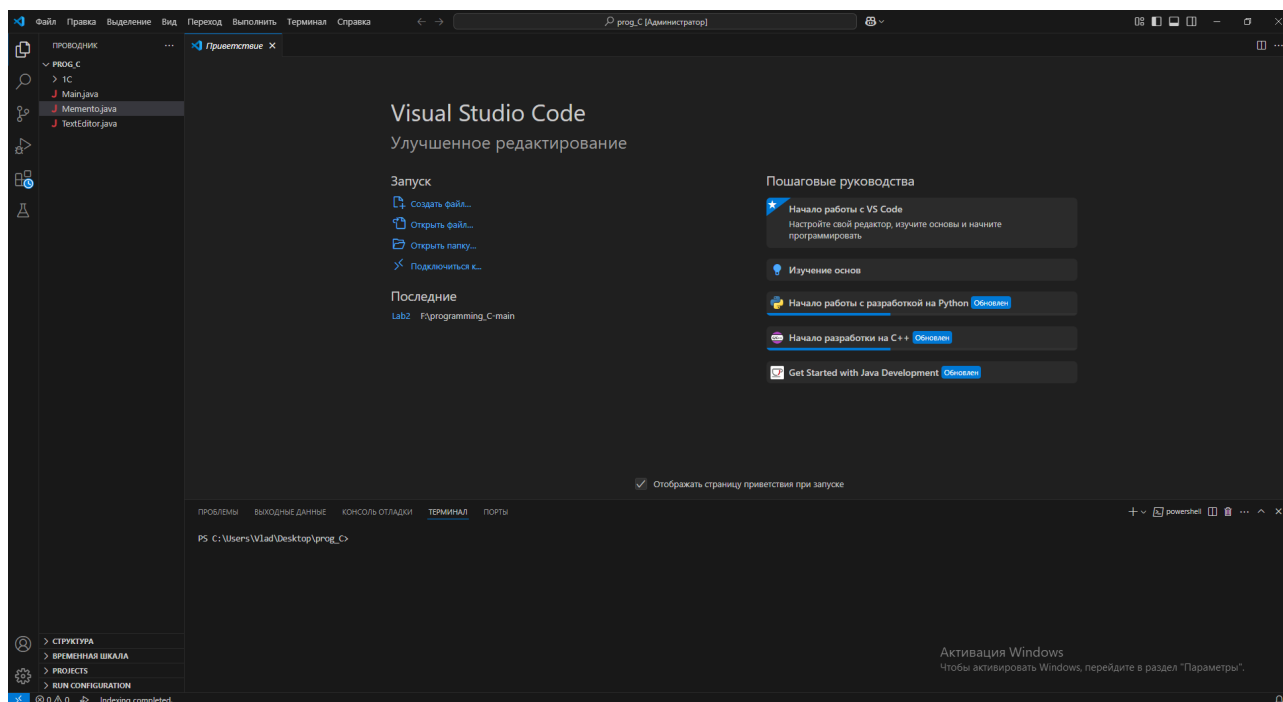


Рисунок 1 – Начальное окно VS Code

Аналогичным образом в архитектуре VS Code реализована подсистема отладки посредством протокола DAP (Debug Adapter Protocol). Стандартизация интерфейса отладки позволяет абстрагировать визуальные компоненты управления выполнением программы (точки останова, шаг с обходом, просмотр значений переменных, стек вызовов) от конкретного низкоуровневого дебаггера (GDB, PDB, движок отладки JVM). Универсальное клиентское окно отладки VS Code взаимодействует с языковым адаптером отладки через структурированные сообщения DAP, предоставляя автоматизатору сквозной, идентичный пользовательский опыт при отладке скриптов любой сложности, написанных на абсолютно разных языках программирования. Такой подход превращает среду разработки в гибкий конструктор, способный трансформироваться под сиюминутные задачи инженера без накопления системной избыточности.

1.4 Сравнительный анализ и выбор среды разработки (IDE)

Перед установкой на эталонный компьютер в рамках учебной практики требуется проведение сравнительного анализа доступного инструментария. В качестве базовых критериев анализа в рамках данного исследования выдвинуты: скорость инициализации среды и потребление системных ресурсов (RAM/CPU), уровень универсальности (способность одинаково эффективно работать с множеством разнородных языков в рамках одного окна), глубина интеллектуальной поддержки кода (качество автодополнения и рефакторинга),

интеграция с инструментами DevOps (Git, Docker, Kubernetes) и возможности удаленной разработки (Remote Development).

Анализируя тяжелые монолитные среды, такие как IntelliJ IDEA или PyCharm, необходимо признать их безупречное качество анализа кода для конкретных целевых языков. Однако в контексте написания скриптов автоматизации их монолитная природа оборачивается существенными издержками. Открытие смешанного проекта (например, репозитория инфраструктурных скриптов, где соседствуют Bash, Python и Java) приводит к избыточной нагрузке на аппаратную часть. Тяжеловесный процесс индексации может занимать длительное время при каждом обновлении веток в системе контроля версий, блокируя интерфейс и снижая оперативность работы инженера. Кроме того, лицензионная политика и разделение на коммерческие и бесплатные версии (Community/Ultimate) накладывают юридические и финансовые ограничения на их свободное развертывание в корпоративных контурах.

Консольные редакторы (Vim, Emacs), демонстрируя эталонные показатели скорости запуска и ничтожное потребление оперативной памяти, наталкиваются на жесткие ограничения человеческого фактора. Высокий порог входа и специфическая кривая обучения делают их труднодоступными для быстрой адаптации новыми сотрудниками команды. Отсутствие полноценного нативного графического интерфейса усложняет одновременную визуализацию сложных структур данных, интерактивную работу с логами и графическое сравнение дифференциалов (diff) в системе контроля версий Git, что замедляет процессы командного ревью кода и отладки крупномасштабных скриптов.

На этом фоне среда Visual Studio Code демонстрирует неоспоримые преимущества, выступая в качестве оптимального сбалансированного решения для автоматизатора. Скорость запуска VS Code близка к скорости запуска обычных текстовых редакторов, а потребление оперативной памяти в базовой конфигурации строго ограничено, что позволяет держать среду постоянно открытой даже на маломощных административных терминалах. Благодаря своей расширяемой модульной архитектуре, базирующейся на протоколах LSP и DAP, VS Code обеспечивает абсолютную полиглотность. В рамках единого рабочего пространства (Workspace) инженер может одновременно редактировать Java-классы автоматизации бизнес-логики, Bash-скрипты развертывания и конфигурации Ansible, получая для каждого файла полноценную контекстную подсветку, автодополнение и валидацию синтаксиса без переключения между различными специализированными программами.

Таблица 2 – Сравнение сред разработки

Среда разработки	Требовательность	Универсальность	Автодополнение и рефакторинг	Инструментарий	Удалённая разработка
IntelliJ IDEA	Средняя	Java и Kotlin, с плагинами доступны другие языки	Имеет данный функционал, нацеленный в основном на язык Java	Совместима с многими инструментами разработчиков в таких как CVS, Subversion, Apache Ant, Maven	Имеется только в платной версии
Vim	Низкая	Способен редактировать большинство типов текстовых файлов	Имеется для текста и кода	Совместима с некоторыми инструментами разработчиков	Имеется с помощью сторонних плагинов
Visual Studio Code	Низкая, чуть выше текстовых редакторов	Способен редактировать большинство типов файлов с кодом	Имеет данный функционал для множества языков	Совместима с многими инструментами разработчиков	Имеется с помощью сторонних плагинов

По итогу из вышеописанных сред разработки был выбран Visual Studio Code, который имеет функционал на уровне монолитных сред, при этом нагружая компьютер чуть сильнее простых консольных редакторов. При этом он обладает более удобным интерфейсом и большим набором различных плагинов для облегчения разработки.

1.5 ПО для клонирования системы

По рекомендации преподавателя для клонирования эталонной системы на новые компьютеры будет использоваться программа под названием Clonezilla. Это специальное программное обеспечение, предназначенное для клонирования данных с запоминающих устройств, а также создания образов системы, которые можно использовать для восстановления, переноса или клонирования системы компьютера вместе со всеми настройками, файлами и данными. Имеет несколько версий, отличающиеся возможностью клонировать на несколько устройств. В ней можно работать как через консольные команды, так и через простой графический интерфейс.

Интерфейс программы представлен на рисунке 2.

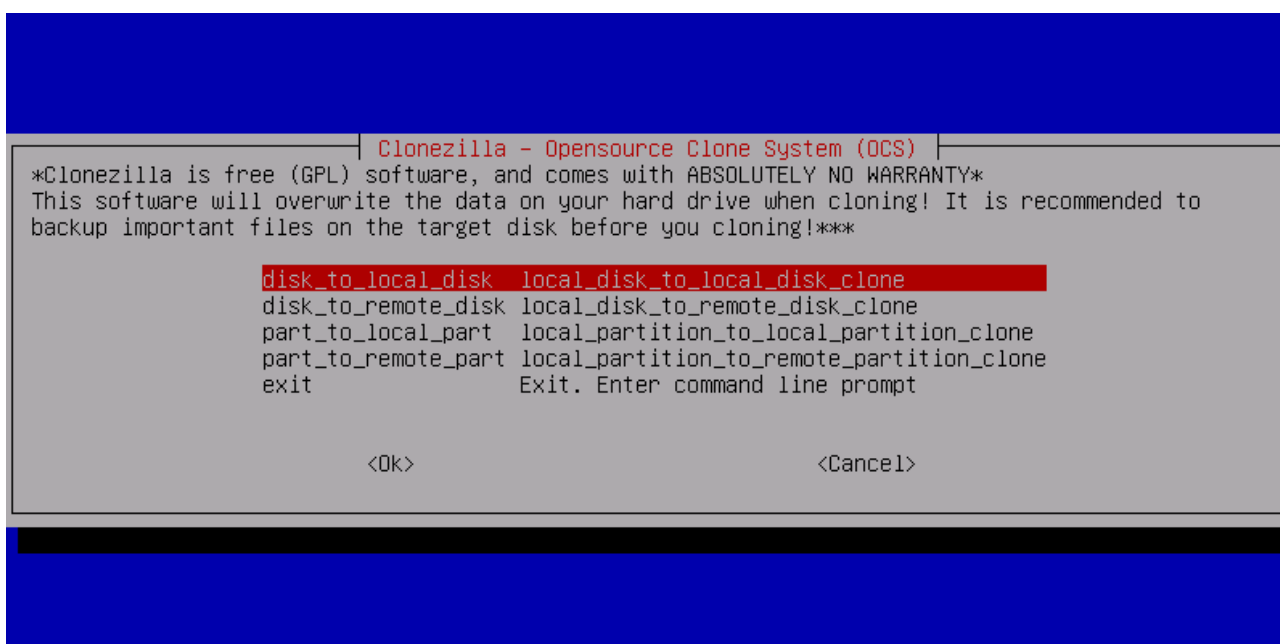


Рисунок 2 – Окно выбора способа копировать систему

Clonezilla live подходит для резервного копирования и восстановления на одном компьютере. Clonezilla lite или SE предназначен для массового развертывания, он может клонировать множество (более 40) компьютеров одновременно. Clonezilla сохраняет и восстанавливает только используемые блоки на жестком диске. Это повышает эффективность клонирования. На некоторых высокопроизводительных устройствах в кластере из 42 узлов было зафиксировано восстановление по многоадресной рассылке со скоростью 8 ГБ/мин. [11].

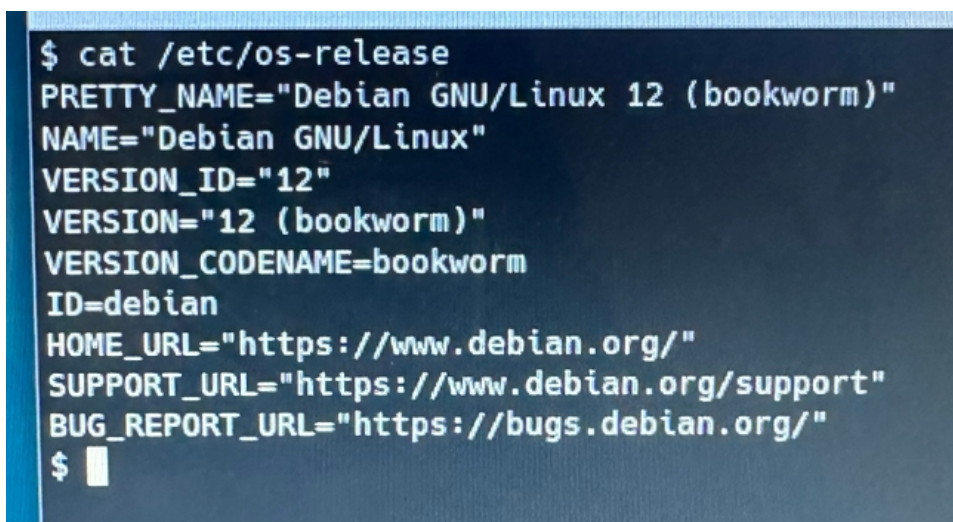
В учебной практике будет использована версия Clonezilla lite Server, чтобы производить клонирование одновременно на несколько машин. Для использования данной программы нам потребуется два переносных запоминающих устройства: один будет использоваться для записи образа программы, а второй для хранения образа системы эталонного компьютера. Сам процесс клонирования будет описан в разделе 2 «Практическая часть».

2 Практическая часть

2.1 Подготовка эталонного компьютера

Эталонный компьютер – машина, с которой будет вестись копирование эталонной системы. Для обеспечения возможности администрирования и выполнения операций с системой на всех компьютерах аудитории был создан учётная запись *praktika* с административными привилегиями. Наличие прав *sudo* является обязательным условием для установки программного обеспечения, модификации системных файлов и выполнения операций по переносу данных.

Первоначальным этапом стала проверка готовности операционной системы на эталонном компьютере. На рисунке 3 представлены сведения о версии ядра и дистрибутиве, установленном на компьютере S-02, который в дальнейшем выступал в роли одного из целевых устройств для клонирования.



```
$ cat /etc/os-release
PRETTY_NAME="Debian GNU/Linux 12 (bookworm)"
NAME="Debian GNU/Linux"
VERSION_ID="12"
VERSION="12 (bookworm)"
VERSION_CODENAME=bookworm
ID=debian
HOME_URL="https://www.debian.org/"
SUPPORT_URL="https://www.debian.org/support"
BUG_REPORT_URL="https://bugs.debian.org/"
$
```

Рисунок 3 – Информация об операционной системе на компьютере S-02

На данном этапе был выполнен вход в систему под учётной записью *root*. Результаты, отображённые на рисунке, указывают на использование операционной системы Debian GNU/Linux 12 (bookworm) с ядром версии 6.1.0-37-amd64. Данная конфигурация является стабильной и широко распространена в образовательных учреждениях. Дополнительно система предоставила сведения об аппаратном обеспечении, включая материнскую плату ASUS Maximus Extreme и версию прошивки 0803.

Для корректного создания пользователя была применена команда «*useradd -m praktika*». Как показано на рисунке 4, выполнение команды прошло без ошибок, что подтверждает успешное создание учётной записи с домашней директорией. Приглашение командной строки вернулось в активное состояние, что свидетельствует о готовности системы к дальнейшим

действиям.

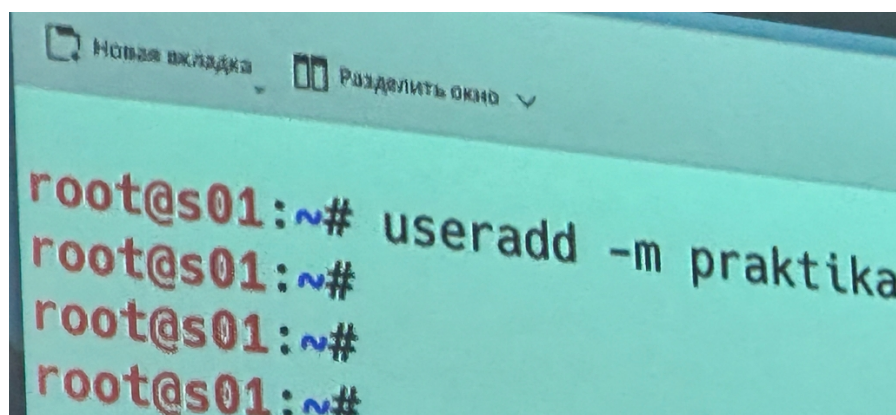


Рисунок 4 – Создание пользователя praktika с помощью useradd

Следующим шагом стала установка пароля для созданного пользователя. На рисунке 5 показано выполнение команды «passwd praktika», которая инициировала интерактивный запрос нового пароля и его подтверждение. В целях безопасности вводимые символы не отображались на экране. Успешное завершение команды подтверждает, что учётная запись полностью готова к использованию.

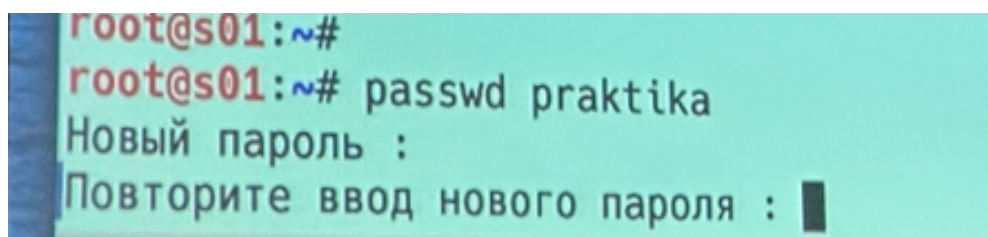


Рисунок 5 – Установка пароля для пользователя praktika

Для автоматизации процесса создания учётной записи на всех компьютерах аудитории был разработан скрипт, представленный на рисунке 6. Первая команда «./pssh -h all "useradd -m -G sudo praktika"» обеспечивает массовое создание пользователя с домашней директорией и добавлением его в группу sudo. Вторая команда устанавливает единый пароль на всех машинах, что значительно упрощает последующее администрирование.

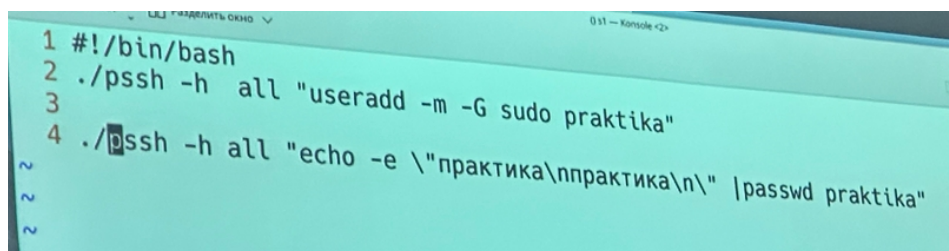


Рисунок 6 – Скрипт для массового создания пользователя praktika

Результаты выполнения скрипта продемонстрированы на рисунке 7.

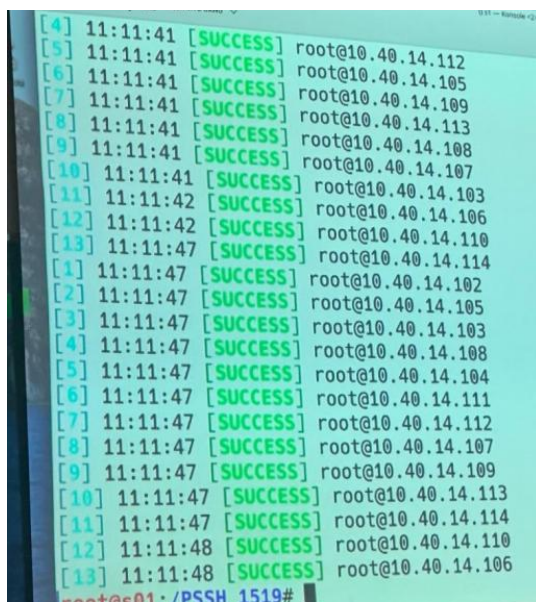


Рисунок 7 – Результат выполнения скрипта на всех компьютерах

Утилита pssh успешно подключилась ко всем компьютерам в диапазоне IP-адресов 10.40.14.102–10.40.14.114 и выполнила команды создания пользователя и установки пароля. Метка [SUCCESS] для каждого хоста подтверждает корректное выполнение операций. Данный подход позволяет эффективно управлять конфигурацией всей аудитории без необходимости ручного вмешательства на каждом устройстве.

2.2 Удаление неактивных пользователей и перенос на жесткий диск

В рамках учебной практики необходимо копировать на этот компьютер пользователей со старых компьютеров и установить выбранное в первом разделе программное обеспечение.

Первой задачей является копирование папок пользователей со всех компьютеров на отдельный переносной жёсткий диск, предоставленный преподавателем. При этом не все папки файлы необходимо копировать. Перед процедурой нужно удалить:

- папки пользователей, которые не проявляли активность позже 1 сентября 2024 года;
- полностью пустые папки пользователей;
- в самих папках удалить файлы расширений .iso (образы дисков), .avi (крупные медиафайлы), .r (файл кода на языке R), файлы .conku и .cache (системные файлы, содержащие данные по действиям пользователя на конкретной машине, не нужны ввиду переноса) и папки с файлами, содержащие слово «linux» (поскольку они скорее всего будут связаны с образами Linux, занимающие много памяти).

Чтобы не осуществлять данные операции вручную, был создан bash-скрипт, который удаляет из папок home и ACM все папки и файлы из списка.

Код скрипта представлен на рисунке 8.

```
#!/bin/bash
sudo find /home -type d -maxdepth 1 ! -newermt "2024-09-01" -o -empty -exec rm -r "{}" \;
sudo find /ACM -type d -maxdepth 1 ! -newermt "2024-09-01" -o -empty -exec rm -r "{}" \;
sudo find /home \( -name '*.iso' -o -name '*.avi' -o -name '.r' -o -name '.conky' -o -name '.cache' -o -name '*linux*' \) -exec rm -r "{}" \;
sudo find /ACM \( -name '*.iso' -o -name '*.avi' -o -name '.r' -o -name '.conky' -o -name '.cache' -o -name '*linux*' \) -exec rm -r "{}" \;
```

Рисунок 8 – Bash-скрипт для удаления лишних пользователей

Данный скрипт использует команду **find**, позволяющая найти файлы и папки по множеству различных условий (например, название, тип, расширение и т.д.) и по-разному взаимодействовать с ними. В качестве каталогов, где будет производится поиск указаны папки **home** и **ACM**. Первые две строчки содержат условие поиска **-type d** (искать только папки) **-maxdepth 1** (искать лишь внутри папки не углубляясь) **! -newermt "2024-09-01"** (изменения были внесены не позднее 1 сентября 2024 года) **-empty** (только пустые папки). **-o** служит логическим или для условий, чтобы удалить и пустые и старые папки со всем содержимым. Через логическое или также прописано условие для удаления файлов определённого расширения и имени через условие **-name ' '** . При обнаружении подходящих файлов они удаляются с помощью условия **-exec rm -r "{}" \;** (выполнить команду удаления папки/файла). После запуска скрипта остаётся лишь копировать оставшихся пользователей на жёсткий диск. Для этого используем файловый менеджер Midnight Commander, активируемый с помощью команды **mc**. В нём необходимо выделить оставшиеся папки пользователей и копировать на диск.

2.3 Установка языков для написания скриптов

Для написания и запуска программ на языке программирования Java необходимо установить Java Development Kit (JDK). Это полный набор инструментов для разработки, компиляции, отладки и запуска программ на языке Java. Он включает все необходимые компоненты для полноценной работы с Java-кодом:

- 1) компилятор (javac). Преобразует исходный код Java в байт-код, который выполняется на виртуальной машине Java (JVM);
- 2) отладчик (jdb). Помогает находить ошибки в коде и анализировать производительность приложений;
- 3) генератор документации (javadoc). Создаёт документацию на основе исходного кода.
- 4) утилита jar. Используется для работы с JAR-архивами;
- 5) интерактивная консоль jshell для экспериментов с кодом;
- 6) стандартные библиотеки Java – готовые классы и API для работы с коллекциями, строками, вводом-выводом, сетью и другими задачами;
- 7) среда выполнения java-программ (JRE) с виртуальной машиной (JVM).

Чтобы установить последнюю версию JDK достаточно ввести команду «`sudo apt install default-jdk`» и дождаться установки всех компонентов на компьютер. После этого осуществляем проверку установки с помощью команды «`java --version`», которая выводит текущую версию JDK.

Установка Visual Studio Code производится с помощью ввода команд, указанных в официальной инструкции [12]. Первым шагом является импорт на компьютер GPG-ключа подписи с помощью команд:

```
sudo apt install wget gpg
wget -qO- https://packages.microsoft.com/keys/microsoft.asc | sudo gpg --dearmor -o
/usr/share/keyrings/microsoft.gpg.
```

После успешного импорта ключа-подписи необходимо создать контекстный файл `vscode.sources` по адресу `/etc/apt/sources.list.d/` для работы программы. Файл можно создать как с помощью команд, так и файлового менеджера Midnight Commander. В этот файл через любой текстовый редактор необходимо вписать контекст, а именно следующие строки:

```
Types: deb
URIs: https://packages.microsoft.com/repos/code
Suites: stable
Components: main
Architectures: amd64,arm64,armhf
Signed-By: /usr/share/keyrings/microsoft.gpg.
```

Завершив все приготовления, остаётся запустить команду «`sudo apt install code`», которая установит VS Code на компьютер.

Выполнив все эти шаги, на эталонный компьютер были установлены средство для работы с языком программирования Java и среда разработки Visual Studio Code.

2.4 Клонирование системы с помощью Clonezilla

Установив всё необходимое программное обеспечение, можно приступить к клонированию эталонной системы на новые компьютеры. Первым шагом является создание образа системы: вставляем диск с Clonezilla и запускаем программу, в меню мы выбираем пункт `savedisk`, чтобы создать образ диска (Рисунок 9).

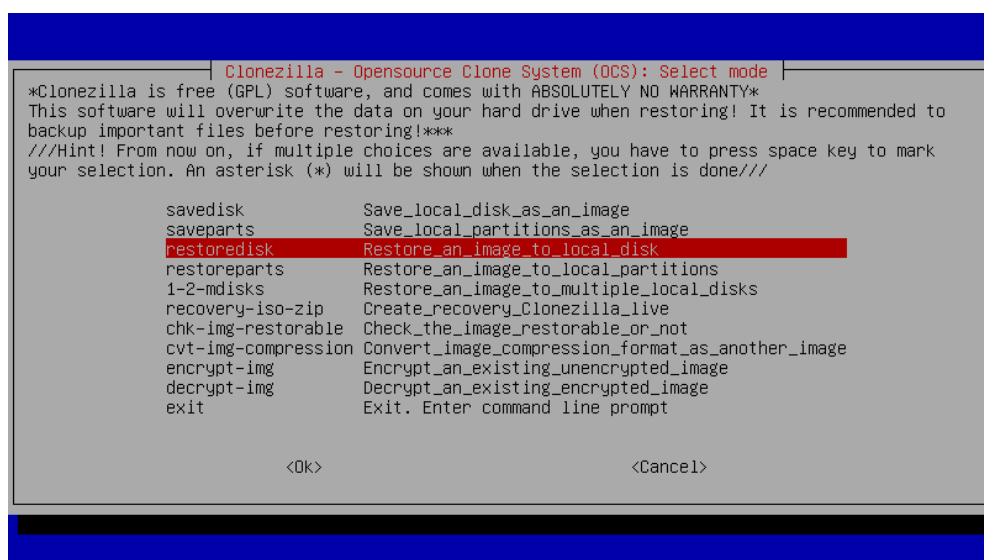


Рисунок 9 – Меню Clonezilla

Для ускорения процесса клонирования можно воспользоваться функцией Clonezilla lite Server для одновременного клонирования системы на несколько компьютеров. Перед этим также необходимо создать локальную сеть из эталонного и новых компьютеров. Чтобы подключить все компьютеры, преподаватель предоставил коммутатор на пять разъемов и пять кабелей для их подключения. После подключения и прописи фиксированных IP-адресов локальная сеть была готова. Чтобы произвести одновременное клонирование системы нужно:

- запустить Clonezilla на эталонном компьютере в режиме сервера;
- поочередно воткнуть диск с Clonezilla и запустить программу на новых компьютерах в режиме клиента;
- проверить целостность образа диска и подключение всех компьютеров;
- запустить процесс копирования.

В итоге все подключённые к сети компьютеры будут осуществлять клонирование системы с эталонной машины (Рисунок 10).

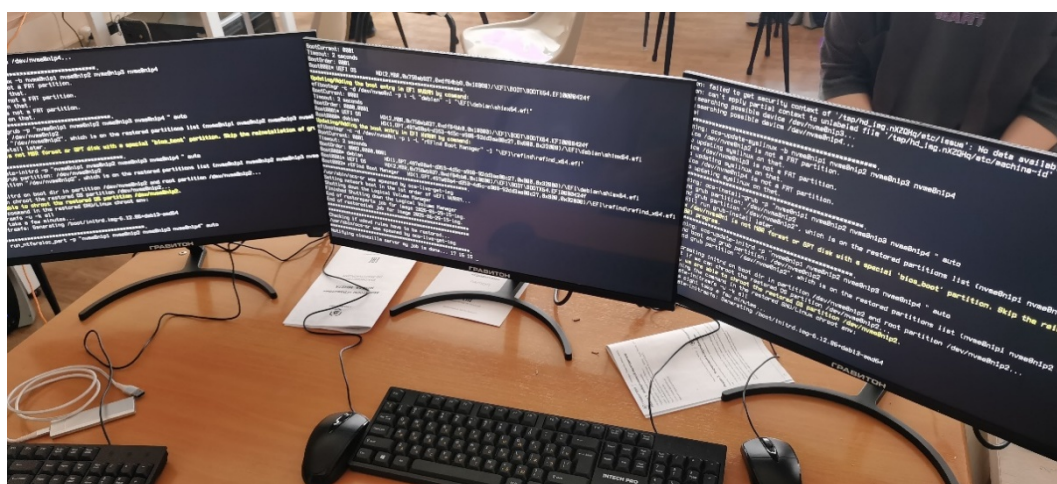


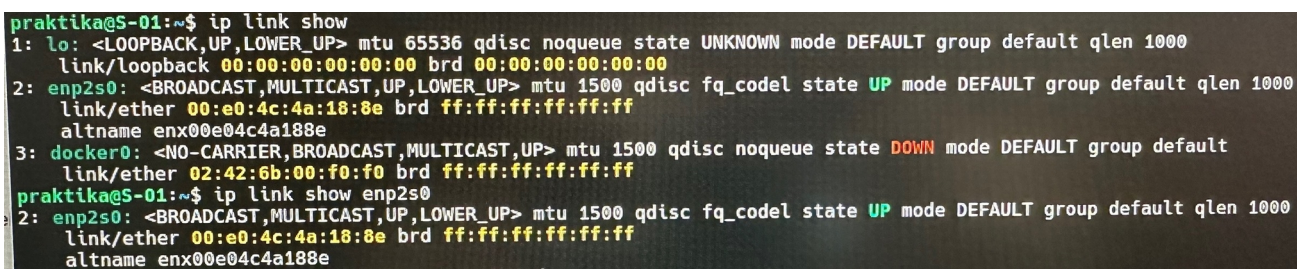
Рисунок 10 – Одновременное клонирование системы на три компьютера

В случае, если возникли проблемы при клонировании через сервер, его также можно осуществить по одной машине. Для этого необходимо подключить к компьютеру диски с Clonezilla и образом эталонной системы и в меню выбрать пункт `restoredisk` в качестве источника указав образ.

2.5 Смена MAC-адресов

После завершения клонирования системы на все компьютеры в аудитории возникла необходимость изменить MAC-адреса сетевых интерфейсов. Это связано с тем, что новые компьютеры имеют иные MAC-адреса сетевой карты, отличающие от уже записанных в таблице маршрутизатора. Это приводит к конфликтам в локальной сети – сетевое оборудование не может корректно идентифицировать устройства с одинаковыми аппаратными адресами. Для решения данной проблемы была выполнена смена MAC-адресов на каждом клонированном компьютере.

Первым шагом необходимо определить текущий MAC-адрес сетевого интерфейса. На рисунке 11 представлен процесс просмотра сетевых интерфейсов и их MAC-адресов.

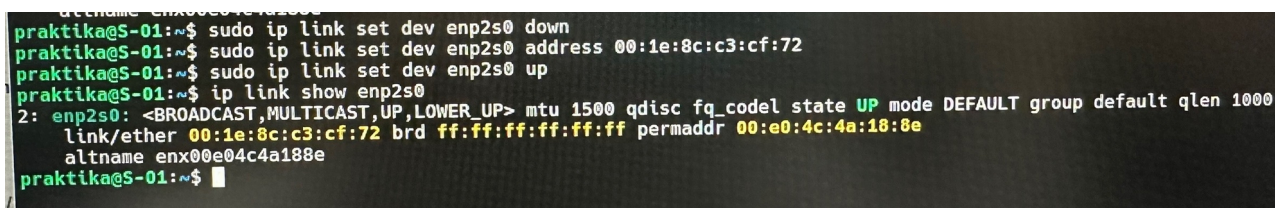


```
praktika@S-01:~$ ip link show
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN mode DEFAULT group default qlen 1000
   link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: enp2s0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UP mode DEFAULT group default qlen 1000
   link/ether 00:e0:4c:4a:18:8e brd ff:ff:ff:ff:ff:ff
   altname enx00e04c4a188e
3: docker0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue state DOWN mode DEFAULT group default
   link/ether 02:42:6b:00:f0:f0 brd ff:ff:ff:ff:ff:ff
praktika@S-01:~$ ip link show enp2s0
2: enp2s0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UP mode DEFAULT group default qlen 1000
   link/ether 00:e0:4c:4a:18:8e brd ff:ff:ff:ff:ff:ff
   altname enx00e04c4a188e
```

Рисунок 11 – Просмотр сетевых интерфейсов и MAC-адресов

Для просмотра информации о сетевых интерфейсах используется команда «`ip link show`». На рисунке видно, что в системе присутствуют три интерфейса: `lo` (локальная обратная связь), `enp2s0` (основной сетевой интерфейс) и `docker0` (интерфейс Docker). Для интерфейса `enp2s0` отображается текущий MAC-адрес `00:e0:4c:4a:18:8e`. Также видна попытка выполнения команды «`sudo link/ether`», которая была введена с ошибкой и не выполнена.

Далее необходимо изменить MAC-адрес на новый. Процесс смены MAC-адреса выполняется в несколько этапов и представлен на рисунке 12.



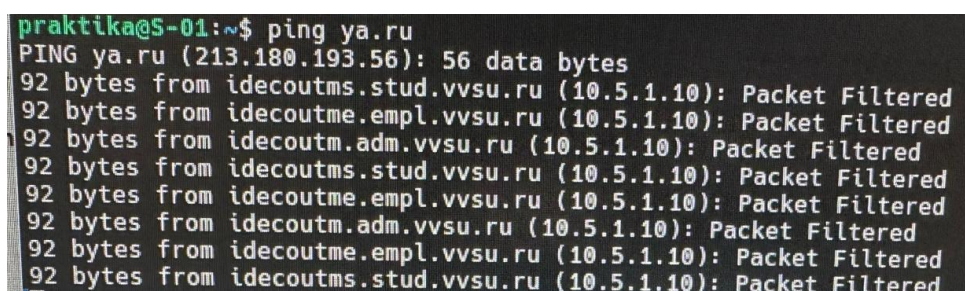
```
praktika@S-01:~$ sudo ip link set dev enp2s0 down
praktika@S-01:~$ sudo ip link set dev enp2s0 address 00:1e:8c:c3:cf:72
praktika@S-01:~$ sudo ip link set dev enp2s0 up
praktika@S-01:~$ ip link show enp2s0
2: enp2s0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UP mode DEFAULT group default qlen 1000
   link/ether 00:1e:8c:c3:cf:72 brd ff:ff:ff:ff:ff:ff permaddr 00:e0:4c:4a:18:8e
   altname enx00e04c4a188e
praktika@S-01:~$
```

Рисунок 12 – Изменение MAC-адреса сетевого интерфейса

Для изменения MAC-адреса необходимо последовательно выполнить следующие

команды. Сначала интерфейс отключается командой «sudo ip link set dev enp2s0 down». Затем устанавливается новый MAC-адрес с помощью команды «sudo ip link set dev enp2s0 address 00:1e:8c:c3:cf:72». После этого интерфейс снова включается командой «sudo ip link set dev enp2s0 up». На рисунке видно, что все команды выполнены успешно. После включения интерфейса выполняется проверка нового MAC-адреса командой «ip link show enp2s0». В результате отображается, что новый MAC-адрес 00:1e:8c:c3:cf:72 успешно установлен, при этом старый адрес сохраняется как permaddr 00:e0:4c:4a:18:8e – это физический адрес, зашитый в сетевую карту производителем.

После смены MAC-адреса необходимо убедиться, что сеть работает корректно и компьютер имеет доступ к интернету. На рисунке 13 представлена проверка работоспособности сети после смены MAC-адреса.



```
praktika@S-01:~$ ping ya.ru
PING ya.ru (213.180.193.56): 56 data bytes
92 bytes from idecoutms.stud.vvsu.ru (10.5.1.10): Packet Filtered
92 bytes from idecoutme.empl.vvsu.ru (10.5.1.10): Packet Filtered
92 bytes from idecoutm.adm.vvsu.ru (10.5.1.10): Packet Filtered
92 bytes from idecoutms.stud.vvsu.ru (10.5.1.10): Packet Filtered
92 bytes from idecoutme.empl.vvsu.ru (10.5.1.10): Packet Filtered
92 bytes from idecoutm.adm.vvsu.ru (10.5.1.10): Packet Filtered
92 bytes from idecoutme.empl.vvsu.ru (10.5.1.10): Packet Filtered
92 bytes from idecoutms.stud.vvsu.ru (10.5.1.10): Packet Filtered
```

Рисунок 13 – Проверка сетевого подключения

На рисунке видно, что новый MAC-адрес 00:1e:8c:c3:cf:72 успешно применён к интерфейсу enp2s0. Затем выполняется проверка доступности внешних ресурсов с помощью команды «ping ya.ru». В ответ на запросы приходят сообщения от различных внутренних серверов (idecoutms.stud.vvsu.ru, idecoutme.empl.vvsu.ru, idecoutm.adm.vvsu.ru), что подтверждает успешное подключение к сети и работоспособность интернет-соединения.

Таким образом, на каждом клонированном компьютере была выполнена смена MAC-адреса, что позволило избежать конфликтов в локальной сети и обеспечить корректную работу всех устройств. В качестве новых адресов использовались уникальные значения, что гарантирует отсутствие дублирования аппаратных адресов в сети аудитории.

2.6 Развертывание кластера

Последним заданием перед написанием отчёта является создание кластера компьютеров в аудитории. Кластер – это группа компьютеров, имеющая возможность выполнять распараллеленные программы. Создание кластера необходимо для надёжной работы крупной системы, чтобы её работа не зависела от одного компьютера. Также кластеризация позволяет равномернее распределять нагрузку при вычислениях и работе программ.

Для развёртывания кластера необходимо будет связать компьютеры не только через локальную сеть, но и через специальное ПО, благодаря которому можно выполнять параллельное программирование. Этим ПО является Local Area Multicomputer (LAM) – среда для создания соединения между компьютерами, а также среда параллельного программирования.

Для связи компьютеров между собой сперва создаётся специальный пользователь с логином cluster и таким же паролем. Чтобы создать данного пользователя на всех компьютерах, лучше всего прописать консольные команды:

```
./pssh -h .all/ "useradd -m cluster"
./pssh -h .all/ "echo -e `\"cluster\\ncluster\\n\\`\" | passwd cluster".
```

Благодаря модулю pssh можно одновременно выполнять многие консольные команды (однако оно не работает при необходимости ввода с клавиатуры).

Дальше необходимо в каждом компьютере зайти через созданного пользователя и сгенерировать специальные ключи, которые необходимы для беспрепятственного доступа к тому же пользователю на других компьютерах. Это необходимо, чтобы программы при параллельном выполнении не сталкивались с запросом пароля. Генерируются ключи с помощью команды «ssh-keygen», после которой необходимо несколько раз нажать enter, пропуская дополнительные настройки. Пример сгенерированного ключа показан на рисунке 14.

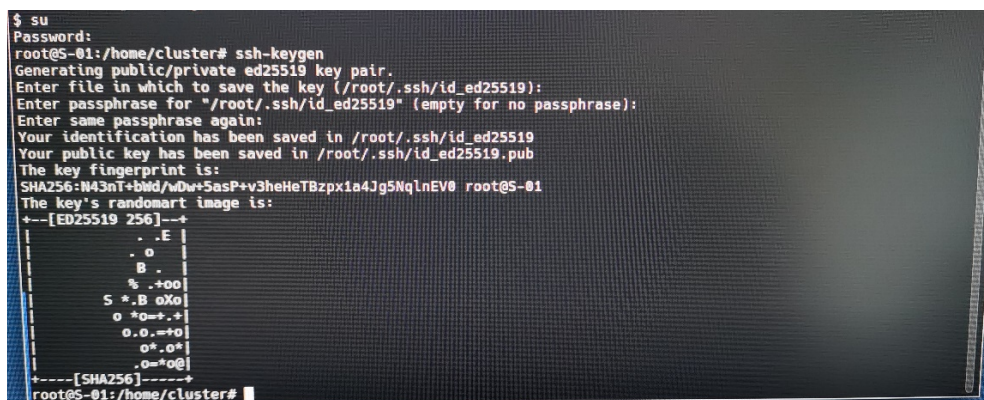


Рисунок 14 – Сгенерированные ключи

Чтобы другие компьютеры могли подключаться, необходимо данный ключ передать с помощью команды «ssh-copy-id сетевое_имя_компьютера».

Последним шагом является установка LAM на все компьютеры. Преподаватель заранее скачал пакет программы в формате .deb на эталонный компьютер, который нужно установить. Для этого выполняем команду «pssh -h /all “cd /opt/lam; dpkg -i *.deb»». Данная команда совершает на всех компьютерах переход в папку lam, где хранится пакет программы и запускает её установку.

Заключение

В ходе выполнения учебной ознакомительной практики были выполнены все поставленные задачи, направленные на изучение языков для написания скриптов, автоматизацию управления пользователями и освоение процессов установки и клонирования программного обеспечения.

В теоретической части работы проведён анализ пяти языков для написания скриптов: `bash`, `tcl`, `perl`, `python` и `java`. По итогу сравнения для установки на эталонный компьютер был выбран язык `Java`. Данный язык обладает большей производительностью, чем остальные представленные в исследовании, он имеет большое количество библиотек, что позволяет писать программы под решения большинства задач. А благодаря кроссплатформенности, написанные программы можно использовать и в других системах с установленным `Java Development Kit`. Особенностью данного языка по сравнению с остальными является строгая типизация, где необходимо чётко отслеживать взаимодействия различных объектов между собой. Это позволяет заранее выявлять ошибки семантических и синтаксических ошибок, несоответствий типов и нарушений интерфейсов ещё на стадии разработки. Поэтому данный язык подходит для создания разнообразного программного обеспечения в учебных и рабочих целях.

В практической части работы был создан пользователь `praktika` с административными правами на всех компьютерах аудитории с помощью утилиты `pssh`. Проведён анализ домашних каталогов, выявлены неактивные пользователи (не заходившие в систему с 2023 года). Разработаны `bash`-скрипты для удаления неактивных пользователей и очистки временных файлов. Выполнен перенос базы данных `User` на внешний жёсткий диск с проверкой его производительности. Также на эталонный компьютер установлен `java` как язык для написания скриптов. С помощью `Clonezilla` создан образ диска и выполнено клонирование на все компьютеры аудитории. Проведена постобработка: удалены старые идентификаторы машин, выполнена смена имён хостов для уникальности в сети.

Учебная ознакомительная практика позволила закрепить и углубить знания и умения в области алгоритмизации, программирования и системного администрирования, полученных в ходе аудиторных (лекционных и лабораторно-практических) занятий.

Список используемых источников

- 1 Требования к оформлению текстовой части письменных работ с изменениями – Текст: электронный // ВВГУ – URL: <https://izdo.vvsu.ru/files-signed/9843E3CA-9D2B-4F63-B5BA-322166D95906.pdf> (дата обращения: 15.06.2026).
- 2 Пирогов В. Ю. Основы программирования на языке Bash: учебное пособие / В. Ю. Пирогов. – Москва: ФЛИНТА, 2024. – 94 с. – ISBN 978-5-9765-5653-9. – Текст: электронный. – URL: <https://znanium.ru/catalog/product/2179290>.
- 3 Ткаченко К.С. Использование программ на высокоуровневых языках OCAML и PERL для аналитического моделирования системы массового обслуживания с буферированием / К.С. Ткаченко // Системный Администратор. – 2021. – №6. – С. 92-95.
- 4 Мусихин А. Г. Проектирование устройств и систем вычислительной техники: учебное пособие / А. Г. Мусихин, Е. С. Данилович. – Москва: РТУ МИРЭА, 2023. – 289 с. – ISBN 978-5-7339-1763-4. – Текст: электронный // Лань: электронно-библиотечная система. – URL: <https://e.lanbook.com/book/369191>.
- 5 Северанс Ч. Р. Python для всех / Ч. Р. Северанс; перевод с английского А. В. Снастина. – Москва: ДМК Пресс, 2021. – 262 с. – ISBN 978-5-93700-104-7. – Текст: электронный // Лань: электронно-библиотечная система. – URL: <https://e.lanbook.com/book/241115>.
- 6 Зорина Н. В. Программирование на языке Джава: учебное пособие / Н. В. Зорина. – Москва: РТУ МИРЭА, 2021 – Часть 1 – 2021. – 148 с. – Текст: электронный // Лань: электронно-библиотечная система. – URL: <https://e.lanbook.com/book/218423>.
- 7 Артём Опарин Скриптовые языки программирования: основы, применение и тренды – Текст: электронный // vc.ru. – 2024. – URL: <https://vc.ru/marketing/1231737-skriptovye-yazyki-programmirovaniya-osnovy-primenenie-i-trendy>.
- 8 Гегечкори Е. Т. Программирование на языке Python: учебное пособие / Е. Т. Гегечкори. – Омск: ОмГТУ, 2023. – 172 с. – ISBN 978-5-8149-3617-2. – Текст: электронный // Лань: электронно-библиотечная система. – URL: <https://e.lanbook.com/book/421673>.
- 9 Ван Х. Б. PyCharm Профессиональная работа на Python: руководство / Х. Б. Ван, К. Нгуен; перевод с английского И. Л. Люско. – Москва: ДМК Пресс, 2024. – 618 с. – ISBN 978-5-93700-274-7. – Текст: электронный // Лань: электронно-библиотечная система. – URL: <https://e.lanbook.com/book/456716>.
- 10 Пантелеев Е. Р. Основы разработки приложений семантического Web на платформе SWI-Prolog: учебное пособие для вузов / Е. Р. Пантелеев. – Санкт-Петербург: Лань, 2025. – 120 с. – ISBN 978-5-507-52206-4. – Текст: электронный // Лань: электронно-библиотечная система. – URL: <https://e.lanbook.com/book/482999>.

11 Clonezilla – About // Clonezilla – URL: <https://clonezilla.org/> (дата обращения: 29.06.2026).

12 Installing Visual Studio Code on Linux // Visual Studio Code – URL: <https://code.visualstudio.com/docs/setup/linux> (дата обращения: 04.07.2026).