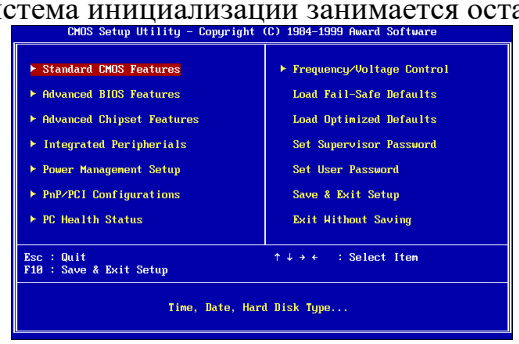
Система инициализации (SystemD)

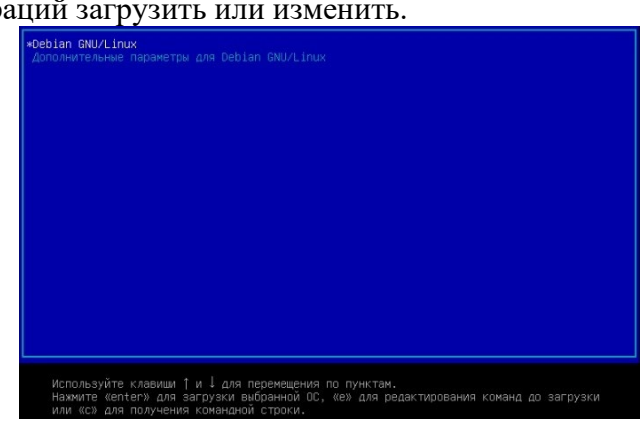
Система инициализации — это система в Linux, которая подготавливает к работе операционную систему. Система инициализации запускается ядром как первый процесс в операционной системе. Затем, этот первый процесс, запускает все остальные процессы. Также при выключении система инициализации занимается остановкой всех процессов.



Всем пользователям персональных компьютеров известно о программной среде БИОС. Это тот компонент системы, который загружается в первую очередь, соответственно, перед запуском операционной системы. БИОС неизменно присутствует на каждом компьютере, так как без нее невозможна целесообразная работоспособность всех элементов вашего компьютера в единой сети. Алгоритм загрузки ОС Linux включает в себя ряд этапов и всё начинается с БИОС, рассмотрим их подробнее.

1. Когда компьютер только включается, то выполняется код из микросхемы **BIOS**[1]**.**
   1.1. Вначале этот код выполняет тестирование системы **POST (power-on self test)**, этим тестированием проверяется аппаратная часть системы.
   1.2. Затем из всех дисков определяется загрузочный. BIOS умеет работать только с дисками размеченными MBR способом. На первом секторе такого диска находится так называемый MBR (master boot record). Сектор диска имеет размер 512 байт. В эти 512 байт помещается маленький загрузчик и таблица разделов. Кстати, если диск не загрузочный, то вместо загрузчика там может находиться код, который сообщает, что это не загрузочный диск (При попытке загрузиться с такого диска мы видим надпись «No bootable device»). BIOS передает управление маленькому загрузчику из MBR на загрузочном диске. Код загрузчика из MBR настолько мал, что способен лишь найти и запустить следующий загрузчик.

Взаимодействуя с **GRUB 2** при старте системы вы можете выбрать, какую из возможных конфигураций загрузить или изменить.



[1] **BIOS** расшифровывается как **Basic Input/Output System**, что в переводе – **Базовая Система Ввода/Вывода**. Это программа записана на специальной микросхеме, которая находится на материнской плате компьютера.

2. Следующий загрузчик уже больше и умнее. Бывают разные загрузчики для Linux, но самым популярным является **GRUB 2**[2]. Он позволяет пользователю не только выбрать операционную систему для загрузки, но и умеет загружать операционные системы с логических разделов. **GRUB 2** расположен в каталоге **/boot** и довольно часто этот каталог выносят на отдельный логический раздел жесткого диска. Для просмотра содержимого этого каталога можно воспользоваться командой: **ls -l /boot**
   В данном каталоге содержатся: **grub** - подкаталог с загрузчиком; файлы **vmlinuz -** ядра разных версий; файлы **initrd.img -** архивы с временной файловой системой и утилитами необходимыми для загрузки ядра.
3. После того, как **GRUB 2** получил управление, он загружает Kernel (в переводе с английского на русский – «ядро») Linux (файл **vmlinuz**). При этом **GRUB 2** загружает в память файловую систему из файла **initrd.img**. А затем запускает ядро, и при запуске, передаёт ему некие параметры.

Зачем нужен образ корневой файловой системы с утилитами **initrd.img**?

В момент времени, когда **GRUB 2** начинает загружать систему **корневая файловая система** еще **не смонтирована**. Поэтому **GRUB 2** использует **initrd** как временную корневую файловую систему, которая загружается из файла в оперативную память.

Initrd – представляет собой файл, содержащий загружаемые модули ядра и минимальный набор утилит для загрузки этих модулей. Он сжимается и загружается в оперативную память с помощью загрузчика. Ядро получает доступ к нему, как если бы была установлена файловая система. Можно воспользоваться командой **lsinitramfs**, чтобы увидеть содержимое файла **initrd.img** (имя файла, зависит от версии ядра, установленного на конкретной системе, которую мы используем):

lsinitramfs -l /boot/initrd.img-9.18.0-13-amd64 | head -n 20

Таким образом, при запуске ядро использует временную файловую систему из файла **initrd.img** для того, чтобы загрузиться полностью.

4. Далее ядро запускает процесс инициализации операционной системы. Процесс инициализации является первым процессом в системе, так как ядро запускалось не как процесс. Каким именно будет процесс инициализации зависит от используемой системы инициализации. Существуют разные системы инициализации, но в **Debian** как и в **Ubuntu** используется система инициализации под названием **SystemD**.

То есть, первым процессом при запуске этих двух систем является **systemd**. Чтобы это проверить выполним следующие две команды:

ps -p 1 позволит увидеть, какой процесс в системе работает под номером 1; из результата выполнения видно, что это **systemd**.

ls -l /sbin/init позволит узнать на что ссылается файл, который традиционно считается файлом системы инициализации; он ссылается на **/lib/systemd/systemd**

Систем инициализации много, первой такой системой была **SysV**, потом для **Ubuntu** написали **Upstart**. Сейчас система инициализации SystemD – является системой инициализации по умолчанию во многих популярных дистрибутивах Linux, в том числе и в **Ubuntu** и в **Debian**.

В первой системе инициализации **SysV** программа инициализации называлась **init**, для совместимости в **SystemD** оставили это название, но **init** это всего лишь символьная ссылка на **systemd**.

[2] GRand Unified Bootloader 2 - это новая версия GRUB, сильно отличающаяся от предыдущей.

UFEI

BIOS существует уже давно и эволюционировал мало. Даже у компьютеров с ОС MS-DOS, выпущенных в 1980-х, был BIOS. Естественно, со временем BIOS всё-таки менялся и улучшался, также были разработаны его расширения, в частности, ACPI, Advanced Configuration and Power Interface (усовершенствованный интерфейс управления конфигурацией и питанием). Это позволяло BIOS проще настраивать устройства и более продвинуто управлять питанием, например, уходить в спящий режим. Но BIOS развился вовсе не так сильно, как другие компьютерные технологии со времён DOS.

У традиционного BIOS до сих пор есть серьёзные ограничения. Он может загружаться только с жёстких дисков объёмом не более 2,1 Тб. Сейчас уже повсеместно встречаются диски большего объема, и с них компьютер с BIOS не загрузится (Это ограничение связано с BIOS MBR).

BIOS должен работать в 16-битном режиме процессора и ему доступен всего 1 Мб памяти. У него проблемы с одновременной инициализацией нескольких устройств, что ведёт к замедлению процесса загрузки, во время которого инициализируются все аппаратные интерфейсы и устройства.

BIOS давно пора было заменить. Intel начала работу над Extensible Firmware Interface (EFI) ещё в 1998 году. Apple выбрала EFI, перейдя на архитектуру Intel на своих Маках в 2006-м, но другие производители не пошли за ней.

Однако, в 2007 Intel, AMD, Microsoft и производители персональных компьютеров договорились о новой спецификации, которая получила название Unified Extensible Firmware Interface (UEFI), унифицированный интерфейс расширяемой прошивки. Это индустриальный стандарт, обслуживаемый форумом UEFI (http://www.uefi.org/) и он зависит не только от Intel. Поддержка UEFI в ОС Windows появилась с выходом Windows Vista Service Pack 1 и Windows 7. Таким образом, почти все компьютеры, которые вы можете купить сегодня, используют UEFI вместо BIOS.

И BIOS, и UEFI - примеры программного обеспечения низкого уровня, которые выполняются (запускаются) при старте компьютера перед тем, как загрузится операционная система.

UEFI – более новое решение, он поддерживает жёсткие диски большего объёма, быстрее грузится, более безопасен – и, что очень удобно, обладает графическим интерфейсом и поддерживает мышь.

UEFI заменяет традиционный BIOS на PC. На существующем PC никак нельзя поменять BIOS на UEFI. Нужно покупать аппаратное обеспечение, поддерживающее UEFI. Большинство версий UEFI поддерживают эмуляцию BIOS, чтобы вы могли установить и работать с устаревшей ОС, ожидающей наличия BIOS вместо UEFI (факт наличия обратной совместимости).

Измерения в байтах								
ГОСТ 8.417—2002			Приставки СИ		Приставки МЭК			
Название	Обозначение	Степень	Название	Степень	Название	Обозначение		Степень
байт	Б	10^0	—	10^0	байт	B	Б	2^0
килобайт	Кбайт	10^3	кило-	10^3	кибибайт	KiB	КиБ	2^{10}
мегабайт	Мбайт	10^6	мега-	10^6	мебибайт	MiB	МиБ	2^{20}
гигабайт	Гбайт	10^9	гига-	10^9	гибибайт	GiB	ГиБ	2^{30}
терабайт	Тбайт	10^{12}	тера-	10^{12}	тебибайт	TiB	ТиБ	2^{40}
петабайт	Пбайт	10^{15}	пета-	10^{15}	пебибайт	PiB	ПиБ	2^{50}
эксабайт	Эбайт	10^{18}	экса-	10^{18}	эксбибайт	EiB	ЭиБ	2^{60}
зеттабайт	Збайт	10^{21}	зетта-	10^{21}	зебибайт	ZiB	ЗиБ	2^{70}
йоттабайт	Ибайт	10^{24}	йотта-	10^{24}	йобибайт	YiB	ЙиБ	2^{80}
роннабайт	Рбайт	10^{27}	ронна-	10^{27}	робибайт	RB	РиБ	2^{90}
кветтабайт	Квбайт	10^{30}	кветта-	10^{30}	квебибайт	QB	КвиБ	2^{100}

Новый стандарт обходит ограничения BIOS. Прошивка UEFI может грузиться с дисков, теоретический предел объема для которых составляет 9,4 зеттабайт. Это примерно в три раза больше всех данных, доступных сегодня в Интернет. UEFI поддерживает такие объёмы из-за использования разбивки на разделы GPT3 (Guid Partition Table – стандарт формата размещения таблиц разделов на физическом жестком диске, часть EFI стандарта) вместо MBR. Также у неё стандартизирован процесс загрузки, и она запускает исполняемые программы EFI вместо кода, расположенного в MBR.

UEFI может работать в 32-битном или 64-битном режимах и её адресное пространство больше, чем у BIOS – а значит, быстрее загрузка. Также это значит, что экраны настройки UEFI можно сделать красивее, чем у BIOS, включить туда графику и поддержку мыши. Хотя, многие компьютеры до сих пор работают с UEFI с текстовым режимом, которые выглядят и работают так же, как старые экраны BIOS.

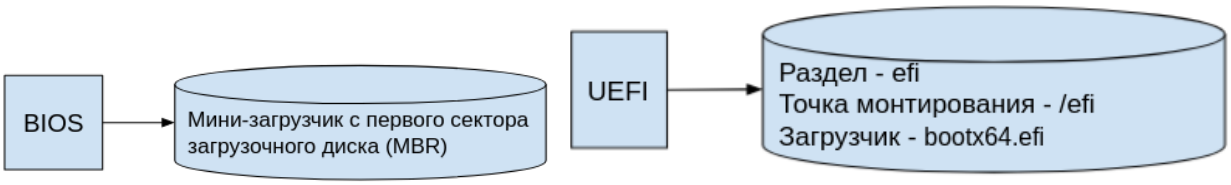
В UEFI встроено множество других функций. Она поддерживает безопасный запуск Secure Boot, в котором можно проверить, что загрузку ОС не изменила никакая вредоносная программа. Она может поддерживать работу по сети, что позволяет проводить удалённую настройку и отладку. В случае с традиционным BIOS для настройки компьютера необходимо было сидеть прямо перед ним.

И это не просто замена BIOS. UEFI – это небольшая операционная система, работающая над прошивкой PC, поэтому она способна на гораздо большее, чем BIOS. Её можно хранить в флэш-памяти на материнской плате или загружать с жёсткого диска или с сети.

У разных компьютеров бывает разный интерфейс и свойства UEFI. Всё зависит от производителя компьютера, но основные возможности одинаковы у всех.

Наличие обратной совместимости, о чем говорилось ранее, приводит к тому, что жесткий диск, разбитый на разделы с помощью таблицы разделов **GPT,** в первом секторе продолжает хранить **MBR** запись. Это делается для того, чтобы старые системы с **BIOS** могли использовать такие диски.

UEFI использует загрузчик, который называется **ESP** или **EFI**. Это специальный загрузчик от **UEFI**. Обычно это файл **/efi/boot/bootx64.efi**.



UEFI имеет специальный режим, который называется **Secure Boot**. Если этот режим включен, то загрузчик без специальной подписи не будет работать. Windows и некоторые системы Linux имеют такие подписи.

[3] EFI использует GPT там, где BIOS использует *Master Boot Record*, MBR

Дальнейшая загрузка происходит также как и в системах с BIOS. Загрузчик запускает GRUB 2, хотя может сразу запустить и ядро Linux, но тогда мы теряем гибкость GRUB 2. GRUB 2 в свою очередь запускает ядро Linux. Дальше ядро запускает систему инициализации, которая запускает все остальные процессы в системе.

SystemD

Основная цель **systemd** это ускорение загрузки операционной системы за счет **распараллеливания** запуска процессов и **отложенного запуска**.

Распараллеливание достигается одновременным запуском не связанных служб. То есть, если одна служба не зависит от второй и у обоих есть достаточно ресурсов для запуска, то они запускаются одновременно.

Отложенный старт достигается за счет подготовки всего необходимого к запуску службы, но сама служба запускается позже, по требованию. Например, создаются все сокеты для работы службы, но служба запустится только когда к сокету обратятся.

Все задачи, которые выполняет **SystemD** описываются в специальных файлах, которые называются юниты (unit). Юниты бывают разных типов, например:
Service – описывает сервис (службу) или скрипт, т.е. всё то, что можно запустить;
Target – группирует юниты, то есть мы можем объединить две службы и запускать их как одну;
Timer – определяет таймер (аналог cron). То есть службы могут запускаться по определённому расписанию, или с задержкой.

Вот примеры юнитов разных типов:

Команда	Описание
ls -l /lib/systemd/system/ssh.service	**ssh.service** – служба ssh сервера, с помощью которого мы удалённо управляем linux системой
ls -l /lib/systemd/system/default.target	**default.target** – таргет который запускается при включении системы по умолчанию, при этом запускаются службы у которых настроен автозапуск для этого таргета
ls -l /lib/systemd/system/logrotate.timer	logrotate.timer – таймер, который запускает службу logrotate.service для ротации логов.

Список всех активных юнитов в системе можно посмотреть с помощью следующей команды:

systemctl list-units

Так как юнитов слишком много, то обычно выводят только юниты определённого типа, например:

systemctl list-units -t service
systemctl list-units -t timer
systemctl list-units -t target

Использование опции -all позволяет получить список не только активных юнитов:

systemctl list-units -t target --all

Сами юниты можно найти в каталогах, список которых можно посмотреть, введя команду: **ls -ld /*/systemd/system**

Результатом выполнения будет получение следующей информация:

/etc/systemd/system	юниты создаваемые вручную, имеют наивысший приоритет.
/lib/systemd/system	системные юниты, устанавливаются обычно вместе с приложениями. Они имеют самый низкий приоритет.
/run/systemd/system	динамически создаваемые юниты. Имеют средний приоритет.

Юниты типа service

В SystemD все сервисы, которые можно запускать или останавливать описываются в специальных юнитах – service, следовательно в них описывается вся информация, которая поможет нам запустить сервис (службу).

Юниты – это обычные файлы. Рассматриваемый тип юнитов описывает способ запуска исполняемых файлов (бинарных или скриптов).

В SystemD есть специальная утилита, которая позволяет управлять службами – systemctl. Рассмотрим некоторые возможные варианты её использования:

systemctl start <unit.service>
Запуск службы, например, systemctl start <имя>. При этом происходит запуск одного или нескольких исполняемых файлов, причем при запуске исполняемых файлов в системе стартуют соответствующие процессы. Таким образом, после запуска службы в системе запустится один или несколько процессов, которые будут работать в одной службе.
systemctl stop <unit.service>
Остановка службы, следствием чего будет и остановка всех связанных со службой процессов. Заметим, что процессы не обязательно должны завершиться, это зависит от реализации самого юнита (что написано в нём).
systemctl status <unit.service>
Проверка статуса службы, т.е. можно узнать запущена ли она или нет, её процессы и их иерархию, также можно посмотреть последние логи.
systemctl reload <unit.service>
Перечитывание конфигурации. Данная команда заставит работающие процессы перечитать свои конфиги, реально, будет выполнено то, что написано в юните.
systemctl restart <unit.service>
Перезапуск службы, то есть остановка и последующее включение службы вновь.
systemctl enable <unit.service>
Включение автозапуска. Данная команда включает автозапуск для службы, но момент срабатывания автозапуска зависит от написанного в юните кода.
systemctl disable <unit.service>
Выключение автозапуска службы.
systemctl cat <unit.service>
вывод содержимого файла юнита. Результат аналогичен выполнению команды cat /путь/unit.service.

Рассмотрим ssh.service

Вначале выполним команду **systemctl status ssh**, чтобы посмотреть статус этой службы.

```
● ssh.service - OpenBSD Secure Shell server
     Loaded: loaded (/lib/systemd/system/ssh.service; enabled; vendor preset: enabled)
     Active: active (running) since Thu 2023-02-21 13:59:59 MSK; 5min ago
       Docs: man:sshd(8)
             man:sshd_config(5)
    Process: 10614 ExecStartPre=/usr/sbin/sshd -t (code=exited, status=0/SUCCESS)
   Main PID: 10615 (sshd)
      Tasks: 1 (limit: 2340)
     Memory: 1.1M
        CPU: 76ms
     CGroup: /system.slice/ssh.service
             └─10615 sshd: /usr/sbin/sshd -D [listener] 0 of 10-100 startups
```

В результате можно увидеть расположение файла юнита: /lib/systemd/system/ssh.service. Также видно, что включена автозагрузка службы (статус) – enabled. И фраза vendor preset: enabled, которая означает что служба *поставляется* с включенной автозагрузкой. То есть, при установке пакета ssh, сразу включится автозапуск этой службы.

Увидеть содержимое юнита можно воспользовавшись одной из команд (результат аналогичен):

cat /lib/systemd/system/ssh.service или **systemctl cat ssh.service**

```
# /lib/systemd/system/ssh.service
[Unit]
Description=OpenBSD Secure Shell server
Documentation=man:sshd(8) man:sshd_config(5)
After=network.target auditd.service
ConditionPathExists=!/etc/ssh/sshd_not_to_be_run

[Service]
EnvironmentFile=-/etc/default/ssh
ExecStartPre=/usr/sbin/sshd -t
ExecStart=/usr/sbin/sshd -D $SSHD_OPTS
ExecReload=/usr/sbin/sshd -t
ExecReload=/bin/kill -HUP $MAINPID
KillMode=process
Restart=on-failure
RestartPreventExitStatus=255
Type=notify
RuntimeDirectory=sshd
RuntimeDirectoryMode=0755

[Install]
WantedBy=multi-user.target
Alias=sshd.service
```

Как видно, данный юнит состоит из трех блоков (разделов): [Unit], [Service], [Install]. В первом блоке описывается сам юнит и указываются возможные ограничения на его запуск; второй блок содержит информацию для запуска, работы и остановки службы; последний блок содержит описание дополнительных условий для запуска или автозапуска службы.

Возможные параметры блока [Unit]

Description	Описание юнита.
Documentation	Источники документации.
After	запускать юнит после запуска перечисленных юнитов, при этом не требуется чтобы эти юниты были запущены, этот параметр влияет только на очерёдность.
Requires	Необязательная опция, которая требует, чтобы перечисленные в ней юниты уже работали в системе, иначе рассматриваемый юнит не запустится.
ConditionPathExists	эта опция проверяет наличия указанного файла. Восклицательный знак перед именем файла означает, что юнит запустится только в том случае, если этого файла в системе нет. Получается, чтобы запретить запуск службы **ssh** достаточно создать файл **/etc/ssh/sshd_not_to_be_run**.

Возможные параметры блока [Service]

EnvironmentFile	Переменные окружения для службы и её процессов будут взяты из указанного файла. Минус перед файлом (как в нашем случае -/etc/default/ssh) означает, что файл может отсутствовать и это не приведёт к ошибке при запуске службы. Можно использовать в качестве файла с дополнительными настройками.
ExecStartPre	Указывается дополнительная команда, которая выполняется перед запуском службы. Можно использовать для подготовки окружения перед запуском.
ExecStart	Запуск службы, в этом параметре нужно указать главный исполняемый файл (утилиту или скрипт), ради которого служба создана.
ExecReload	Содержимое будет выполнится при **systemctl reload <unit.service>**. Если утилита не умеет перечитывать свою конфигурацию без перезагрузки, то можно опустить этот параметр. Параметров ExecStartPre, ExecStart, ExecReload может быть несколько, если нужно выполнить несколько команд.
KillMode	Указывает, как процессы службы должны завершаться. Существует несколько вариантов: control-group – сигнал остановки будет послан всем процессам службы; mixed – первый сигнал остановки будет послан главному процессу, а второй всем остальным процессам; process – сигнал остановки будет послан только главному процессу, все остальные процессы (если они есть) останутся работать; none – никому не будет отправлен сигнал остановки, служба будет выключена, но все процессы продолжат работать (возможность реализована, но не рекомендуется).
Restart	Здесь можно указать, в каких случаях нужно перезагружать службу: on-failure – при ошибке; on-success – при успехе (при корректном завершении службы); no – никогда (по умолчанию); always – всегда, то есть если ваша служба по

	каким-то причинам останавливается корректно, то этот параметр её перезапустит. Но это не сработает, если выполнить команду systemctl stop <unit.service>.
RestartPreventExitStatus	Если код аварийного завершения будет соответствовать указанному (в нашем случае 255), то служба не перезапускается, даже если установлен **Restart=on-failure**.
Type	Тип юнита – важный параметр, который отвечает за способ запуска службы и её процессов. Возможны следующие типы: simple или exec – используются для запуска программы как службы (Данный тип стоит использовать, когда запускаемая программа не умеет сама запускаться в фоновом режиме, т.е. запускается на переднем плане. Прервать выполнение такой программы можно с помощью комбинации клавиш Ctrl+C. При этом simple отрабатывает быстрее, но не следит за запуском исполняемого файла, указанного в ExecStart. А exec дожидается запуска исполнимого файла, и только после этого служба считается запущенной. Рекомендуется использовать exec вместо simple, так как в simple служба может оказаться успешно запущенной, даже если исполнимый файл не смог запустится.); forking – этот тип следует использовать если запускаемое приложение умеет само запускаться в фоновом режиме (Это можно использовать, если с помощью одного скрипта запускается несколько процессов, которые умеют запускаться в фоне. То есть в ExecStart указан скрипт, а в скрипте запускаются фоновые процессы. В этом случае сам скрипт выполнится и завершится, а служба будет управлять запущенными фоновыми процессами.); oneshot – если подразумевается разовый запуск утилиты или скрипта, то подойдет этот тип (Служба такого типа никогда не будет запущенной, т.е. после запуска службы происходит выполнение скрипта, далее служба останавливается.); dbus – если службе необходимо использование соединения D-Bus, то можно использовать этот тип службы (Когда служба запустится, то получит имя на шине D-Bus. Если это имя освободится, то служба будет считаться неисправной и все процессы службы начнут завершаться.); notify – поведение похоже на exec, но дополнительно ожидается, что служба отправит сигнал готовности после своего запуска; idle – система отложит выполнение двоичного файла службы до окончания запуска остальных (более срочных) задач, максимум на 5 секунд. В остальном поведение аналогично simple.
RuntimeDirectory	После запуска службы будет создана указанная директория. В нашем случае директория **/run/sshd/**
RuntimeDirectoryMode	Директория будет создана с указанными правами. В нашем случае – **755 (rwx r-x r-x)**.

Дополнительные опции блока [Service], которые можно использовать для создания своих служб:

User	Используя данный параметр можно указать пользователя (username или uid), от чьего имени будут запущены процессы службы. То есть этот пользователь будет запускать исполняемый файл, указанный в **ExecStart**.
ExecStop	Команда, которую нужно выполнить при остановке службы (аналогично systemctl stop <unit.service>).

KillSignal	Позволяет указать сигнал остановки процессов. А если этот параметр не указать, то будет использован сигнал **SIGTERM**.
LimitNOFILE	Позволяет ограничить максимальное число открытых файлов, которые смогут открыть процессы службы.
LimitCPU	Позволяет ограничить максимальное количество процессорного времени в секундах, после исчерпания которого служба завершится с ошибкой.
LimitRSS	Позволяет ограничить лимит потребления памяти (в байтах). При достижении лимита служба завершится с ошибкой.
LimitNPROC	Позволяет задать максимальное число процессов в службе.
Nice	Позволяет задать уровень любезности запускаемых процессов службой. Любезность (Nice) – это параметр обратный приоритету. Чем выше Nice, тем ниже приоритет у процесса, и тем меньше он нагружает процессор. Может быть от -20 (самый приоритетный) до 19 (наименьший приоритет).

Возможные параметры блока [Install]

WantedBy	Если мы включим автозагрузку этой службы (с помощью команды **systemctl enable <имя службы>**), то она должна запуститься при загрузке мультипользовательского режима (**multi-user.target**).
Alias	Псевдоним службы. Указание псевдонима приведёт к тому, что к службе можно будет обратиться ещё и по имени псевдонима. Например, вместо **ssh.service** можно использовать **sshd.service** (**systemctl status sshd.service**).

Дополнительно можно посмотреть документацию по настройке юнитов:
https://www.freedesktop.org/software/systemd/man/systemd.unit.html
https://www.freedesktop.org/software/systemd/man/systemd.exec.html

Юниты типа target

Система инициализации **SystemD** состоит из юнитов. Иногда требуется объединить различные юниты в группу, например, создать группу, которая будет запускать сразу несколько служб. Для достижения подобных целей существует специальный тип юнитов – target.

В отличие от юнитов типа Service, этот тип юнитов имеют только 2 блока – [Unit] и [Install], т.е. отдельного блока [target] не существует.

Пример объединения служб **ssh** и **apache2**:

sudo nano /etc/systemd/system/ssh-apache.target

```
[Unit]
Description=ssh apache target
Requires=ssh.service
Requires=apache2.service

[Install]
WantedBy=multi-user.target
```

На рисунке показано создание **ssh-apache.target.** Нами были использованы следующие параметры:

Description	Описание юнита
Requires	Указывается юнит, который должен запуститься при запуске таргета. В случае необходимости запуска нескольких служб можно использовать несколько таких параметров.
WantedBy	Указывается загрузочный таргет для запуска юнита.

Для проверки работоспособности созданного таргета:

1. Выключим обе службы:
   sudo systemctl stop apache2.service
   sudo systemctl stop ssh.service
2. Убедимся, что они выключены:
   systemctl status apache2.service
   systemctl status ssh.service
3. После создания нового юнита обязательно нужно выполнить команду заставляющую systemd перечитать все файлы:
   systemctl daemon-reload:
4. Запустим созданный ssh-apache.target:
   sudo systemctl start ssh-apache.target
5. Проверим статус таргета:
   systemctl status ssh-apache.target
6. Проверим статус связанных служб:
   systemctl status apache2.service
   systemctl status ssh.service

Таргеты можно не только запускать и останавливать, но и включать их в автозагрузку, используя команду:

sudo systemctl enable ssh-apache.target

Загрузочные таргеты. В Linux есть так называемые режимы запуска, в первой системе инициализации они назывались “**Уровни запуска**” (**runlevel**). В **SystemD** они называются загрузочными таргетами, или **boot targets**.

В операционной системе Windows есть режим безопасной загрузки и обычной загрузки. В Linux это работает примерно также. На каждом уровне – запускаются или останавливаются определённые службы. В отличие от ОС Windows, в ОС Linux можно переходить из одного уровня в другой не перезагружая компьютер. Стоит отметить, что **перезагрузка** – это один из уровней запуска, также как и **выключение** компьютера.

Можно настроить определенный уровень запуска, как уровень по умолчанию. Именно в этом режиме будет загружаться ваша система.

Чтобы посмотреть список существующих загрузочных таргетов можно использовать следующую команду:

ls -l /lib/systemd/system/runlevel*.target

```
/lib/systemd/system/runlevel0.target -> poweroff.target
/lib/systemd/system/runlevel1.target -> rescue.target
/lib/systemd/system/runlevel2.target -> multi-user.target
/lib/systemd/system/runlevel3.target -> multi-user.target
/lib/systemd/system/runlevel4.target -> multi-user.target
/lib/systemd/system/runlevel5.target -> graphical.target
/lib/systemd/system/runlevel6.target -> reboot.target
```

Как видно на рисунке видна связь загрузочных таргетов SystemD и уровней запуска (runlevel). То-есть каждый уровень запуска – это символьная ссылка на определённый таргет.

А для перехода в другой режим нужно выполнить специальную команду с указанием названия соответствующего таргета, например:

sudo systemctl isolate rescue.target

sudo systemctl isolate multi-user.target

Таргет по умолчанию или **default.target,** является символьной ссылкой на один из загрузочных таргетов:

ls -l /lib/systemd/system/default.target

```
/lib/systemd/system/default.target -> graphical.target
```

Как видно даже серверная **Ubuntu**, также как и **Debian**, по умолчанию загружается в графическом режиме (**graphical.target**). Хотя и не была установлена изначально(!).

Рассмотрим юнит, используемый по умолчанию:

cat /lib/systemd/system/default.target

```
[Unit]
Description=Graphical Interface
Documentation=man:systemd.special(7)
Requires=multi-user.target
Wants=display-manager.service
Conflicts=rescue.service rescue.target
After=multi-user.target rescue.service rescue.target display-manager.service
AllowIsolate=yes
```

На рисунке определены следующие параметры:

Requires=multi-user.target	При запуске юнита, будет запущен юнит **multi-user.target**.
Wants=display-manager.service	Юнит ожидает запуска службы **display-manager.service**, но не требует её запуска. Это означает, что загрузку графического интерфейса мы подождем, но если он не установлен, то всё равно произойдет запуск.
Conflicts=rescue.service rescue.target	Данный параметр означает, что если указанный юнит выполняется, то данный юнит не будет запущен. А режим **rescue.target** – это режим восстановления, чем-то похож на *Безопасный режим* в Windows. Следовательно, если запущен режим восстановления (**rescue.target**), то нельзя запускать графический режим (**graphical.target**).
After	По очерёдности юнит должен запускаться после указанных юнитов, но не требует их запуска.
AllowIsolate	Если этот юнит можно использовать в команде **systemctl isolate <название таргета>**, то здесь нужно поставить **yes**. То есть, с помощью этого параметра, мы указываем что это именно **boot target**.

Второй способ выяснения **default target**‘а, т.е. используемого по умолчанию, заключается в использовании команды:

systemctl get-default

Более подробную информацию можно прочесть перейдя по адресу:
https://www.freedesktop.org/software/systemd/man/systemd.target.html

Юниты типа timer

Вначале следует отметить, что данные юниты сейчас постепенно вытисняют, а со временем возможно и заменят планировщик заданий – cron. Данный тип юнитов позволяет по расписанию запускать службы.

Чтобы понять, как это работает выполним следующие действия:

1. Создадим простой скрипт:
   nano timer.sh

```
#!/bin/bash
date >> ~/date.log
```

   Разрешим запускать этот скрипт пользователю:
   chown u+x timer.sh
2. Создадим сервис, который будет запускать созданный нами скрипт:
   sudo nano /etc/systemd/system/test-timer.service

```
[Unit]
Description=test timer

[Service]
Type=oneshot
ExecStart=/home/test/timer.sh
User=test
```

3. Создадим таймер, который будет запускать сервис по определённому расписанию:
   sudo nano /etc/systemd/system/test-timer.timer

```
[Unit]
Description=test timer

[Timer]
OnUnitActiveSec=30s
Unit=test-timer.service
AccuracySec=1s

[Install]
WantedBy=timers.target
```

В рассматриваемом примере:

OnUnitActiveSec=30s	Параметр определяет отсчёт времени от момента запуска юнита, который был активирован таймером. Другими словами отсчет ведётся с момента запуска test-timer.service, и когда он доходит до нуля, таймер срабатывает вновь. Допускается использование других параметров: OnActiveSec – отсчёт относительно момента активации самого таймера.

	OnBootSec – отсчёт ведётся с момента запуска компьютера, например через 30 секунд, после запуска компьютера. OnUnitInactiveSec – отсчёт начинается с момента деактивации юнита, который запускается таймером. Этот параметр противоположен OnUnitActiveSec. OnCalendar – определяет таймер реального времени. Этот параметр может быть указан несколько раз, если нужно. Пример: Thu,Fri 2023-*-1,15 12:34:56 – означает, что юнит будет запущен в 12:34:56, первого или пятнадцатого дня любого месяца 2023 года, но только если этот день приходится на четверг или пятницу. Все эти параметры можно включать в один таймер. При этом, запускаемый юнит будет выполнен, когда любой из отсчётов дойдёт до нуля. Время можно указывать не только в секундах, например можно указать следующее значение: OnBootSec=8h 10min.
Unit=test-timer.service	Указываем какой юнит следует запускать.
AccuracySec=1s	Точность таймера равна 1 секунде. По умолчанию точность таймера равно 1 минуте. Поэтому для заданий, которые выполняются чаще 1 минуты, нужно использовать этот параметр.

После создания юнита выполним команду:

sudo systemctl daemon-reload

Запустим службу для проверки, что скрипт работает как надо и проверим это посмотрев лог-файл:

sudo systemctl start test-timer.service

cat date.log

Запустим таймер:

sudo systemctl start test-timer.timer

Проверим лог-файл снова:

cat date.log

Подробнее: https://www.freedesktop.org/software/systemd/man/systemd.timer.html

Команды SystemD

☞ Команда **systemctl** – управляет юнитами, запускает их, останавливает, показывает конфигурацию, основные её параметры мы уже рассматривали, а дополнительно можно прочитать перейдя по ссылке: https://man7.org/linux/man-pages/man1/systemctl.1.html

☞ Команда **systemd-analyze** показывает необходимое для загрузки время:

systemd-analyze

```
Startup finished in 3.033s (kernel) + 15.281s (userspace) = 18.314s
graphical.target reached after 14.855s in userspace
```

С её помощью можно проверить время запуска отдельных служб:

systemd-analyze blame

В результате её выполнения мы сможем увидеть временные метки и последовательность запуска служб **SystemD**.

☞ Команда **timedatectl** позволяет увидеть и изменить настройки системных часов. Её можно использовать для изменения текущей даты, времени и часового пояса, включения автоматической синхронизации системных часов с NTP-сервером.

Утилита показывает время в разных ракурсах: Local time (локальное время в операционной системе); Universal time (всемирное координированное время (UTC), стандарт по которому общество регулирует время); RTC time (часы реального времени, представляют из себя элемент материнской платы).

Часовой пояс в системе определяется файлом **/etc/localtime**, который является символьной ссылкой, его расположение можно увидеть воспользовавшись командой:

ls -l /etc/localtime

Можно вручную отредактировать эту ссылку и указать нужный часовой пояс:

sudo ln -sf /usr/share/zoneinfo/Europe/Moscow /etc/localtime

Или воспользоваться утилитой **timedatectl**:

sudo timedatectl set-timezone «Europe/Moscow»

Если не устраивает выбранный часовой пояс, то можно посмотреть список поддерживаемых часовых поясов так:

timedatectl list-timezones

Можно установить время (**sudo timedatectl set-time 11:00:00**), дату (**sudo timedatectl set-time 2023-02-22**) и время и дату единовременно (**sudo timedatectl set-time '2023-02-22 11:00:00'**), но это возможно только при отключенной синхронизации (**sudo set-ntp off**).

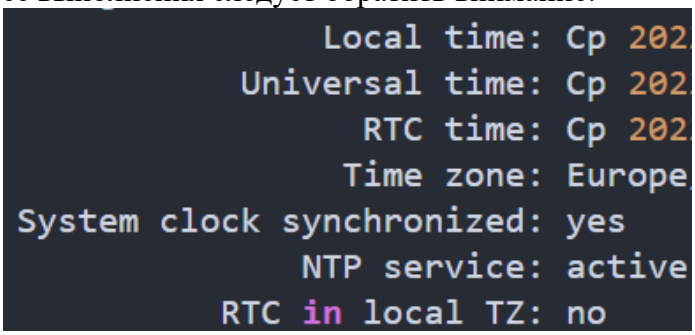
В системах с **SystemD** есть механизм синхронизации времени, который называется **systemd-timesyncd**. Этот механизм является частью **SystemD** и не требует установки. Для просмотра текущего времени на сервере linux:

Date

Для просмотра более подробной информации:

timedatectl

В результате её выполнения следует обратить внимание:



- System clock synchronized — синхронизация времени включена;
- NTP service — служба NTP активна;
- RTC in local TZ — аппаратные часы работают не в локальном часовом поясе.

Добавлять сервера синхронизации времени можно в конфигурационном файле /etc/systemd/timesyncd.conf, а редактировать этот файл может только root. Также после редактирования нужно перезагрузить службу:

sudo nano /etc/systemd/timesyncd.conf

```
NTP=0.ru.pool.ntp.org 1.ru.pool.ntp.org 2.ru.pool.ntp.org 3.ru.pool.ntp.org
FallbackNTP=0.debian.pool.ntp.org 1.debian.pool.ntp.org 2.debian.pool.ntp.org 3.debian.pool.ntp.org
RootDistanceMaxSec=5
PollIntervalMinSec=32
PollIntervalMaxSec=2048
```

- NTP — пул серверов времени для синхронизации;
- FallbackNTP — запасной пул серверов;
- RootDistanceMaxSec — попытка связаться с основным пулом в течении этого количества секунд;
- PollIntervalMinSec и PollIntervalMaxSec — если не получилось связаться с пулами серверов в течении RootDistanceMaxSec, то повторять попытки через этот интервал (указывается минимальный и максимальный интервал).

Перезагружаем службу:

sudo systemctl restart systemd-timesyncd

☞ Команда **hostnamectl** используется для получения и изменения имени компьютера.

Данный инструмент различает три разных имени одного компьютера:

Pretty hostname (красивое имя) – может включать любые специальные символы и храниться по адресу: **/etc/machine-info.** Пример установки красивого имени компьютера:

sudo hostnamectl set-hostname "fast rabbit" –pretty

В результате «красивое имя», введенное вами будет записано в файл */etc/machine-info* config file. При отсутствии данного файла можно его создать (**touch /etc/machine-info**) и вписать в кавычках значение соответствующего параметра (**PRETTY_HOSTNAME="fast rabbit "**).

Static hostname (статическое имя) – используется для инициализации компьютера в сети, поэтому длина ограничена 64 символами, его можно увидеть по адресу: **/etc/hostname**.

Если статическое имя установлено и отличное от localhost, то временное имя не используется, так как оно является запасным.

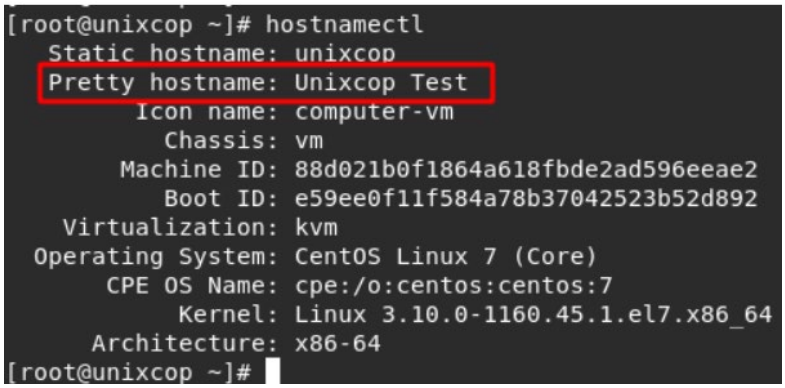
Посмотреть статическое имя можно также воспользовавшись командой:

cat /etc/hostname

Или в более подробном формате, для этого выполним:

hostnamectl

В результате выполнения данной команды можно увидеть признак работы в виртуальной среде: «Virtualization: kvm»



☞ Команда **localectl** используется для получения и изменения региональных настроек (локали) в системе.

Вывод текущих настроек:

localectl

Список доступных для выбора (сгенерированных) локалей

localectl list-locales

Изменение локали по умолчанию:

sudo localectl set-locale LANG="ru_RU.UTF-8"

☞ Команда **loginctl** позволяет просматривать активные сеансы пользователей и завершать их.

Просмотр активных сеансов работы:

loginctl list-sessions

Просмотр списка подключенных пользователей:

loginctl list-users

Заметим, что пользователи переключившие себя на других пользователей с помощью команды su, не будут показаны. Например если пользователь подключился по ssh под именем test, а затем используя su переключился на root, то здесь будет только первый логин test.

Убить сессию по имени пользователя (сможет только root):

sudo loginctl kill-user <имя пользователя>

Завершить сессию по имени пользователя (сможет только root):

sudo loginctl terminate-user <имя пользователя>

В качестве имени можно использовать и номер сессии, полученный в результате выполнения **loginctl**.

☞ Команда **journalctl** используется для просмотра содержимого журнала **SystemD**. Системный журнал хранит записи ядра, служб и приложений в бинарном виде.

Логирование настраивается в конфигурационном файле: **/etc/systemd/journald.conf**.

Среди параметров, наиболее интересны:

Storage	**volatile**	хранить журнал только в памяти, используется /run/log/journal
	persistent	хранить журнал на диске, используется /var/log/journal
	auto	если существует файл /var/log/journal то хранится на диске, если нет то в памяти
	none	не хранить журнал
Compress	**yes**	сжимать логи
	no	не сжимать логи

SplitMode	Определяет доступ к журналу пользователям:	
	uid	все пользователи имеют доступ
	login	каждый пользователь может читать только сообщения, относящиеся к его сеансу
	none	пользователи не имеют доступа
SyncIntervalSec	время в секундах, после которого происходит запись журнала на диск	
RateLimitIntervalSec и **RateLimitBurst**	настройки ограничения скорости генерации сообщений для каждой службы. По умолчанию 10000 сообщений за 30 секунд	
SystemMaxUse	максимальный размер журналов на диске	
SystemKeepFree	объём свободного места, которое должно оставаться на диске после сохранения логов	
RuntimeMaxUse	максимальный объём, который логи могут занимать в файловой системе /run	
RuntimeKeepFree	объём свободного места, которое должно оставаться в файловой системе /run после сохранения логов	

Команда **journalctl** без параметров покажет все содержимое журнала, поэтому уместно использовать дополнительные параметры, позволяющие отфильтровать вывод. Некоторые варианты использования:

Логи с момента загрузки системы, использующей SystemD:
journalctl -b
Список предыдущих сессий (при условии хранения журнала на диске):
journalctl --list-boots
Журнал конкретной сессии:
journalctl -b <номер сессии>
Вывод журнала с определенной даты:
journalctl --since "2023-02-22 09:00:00"
Вывод журнала за вчера и сегодня:
journalctl --since yesterday
journalctl --since today
Журнал для определенного сервиса по имени сервиса:
journalctl -u <имя сервиса>
Фильтруем журнал по номеру процесса:
journalctl _PID=<номер процесса>
журнал по UID пользователя:
journalctl _UID=<номер пользователя>
Только сообщения ядра ОС:
journalctl -k
Сообщения об ошибках:
journalctl -p err
Наблюдать за журналом в режиме реального времени:
journalctl -f
Посмотреть сколько занимает журнал места на диске:
journalctl --disk-usage

Задание к лабораторной работе №3:

Цель работы Получение практических навыков по работе с подсистемой инициализации и управления службами.

Задача_1: написать демон, представляющий собой программу для отдыха глаз. Демон будет показывать уведомление о начале отдыха раз в заданный промежуток времени (repetition_period) и уведомление об окончании отдыха через заданный промежуток времени (relax_time). Временные промежутки задаются случайным образом.

Вариант 1	Картинки «Красивые места»
Вариант 2	Стереокартинки Например: https://kartinki.pibig.info/30844-kartinki-dlja-otdyha-glaz.html
Вариант 3	Геометрические фигуры
Вариант 4	Герои мультфильмов
Вариант 5	Графические файлы из папки пользователя
Вариант 6	На выбор студента исходя из анализа предметной области

Задача 2. Корутина (Сопрограмма (англ. coroutine))

На диске лежат файлы, в них числа в строковом виде, в произвольном порядке, они разделены пробелами. Нужно отсортировать каждый файл и затем слить их в один большой. То есть выполнить сортировку слиянием.
Реализовать задачу с использованием корутины.

Правила:

- Каждый файл надо сортировать в отдельной корутине. Эта часть задания нацелена на понимание кооперативного планирования задач.

- Файлы имеют кодировку ASCII. То есть это обычный текст, не unicode или что-то такое.

- Необходимо замерить время выполнения операций, используя clock_gettime (CLOCK_MONOTONIC).

Ограничения:

- Глобальные переменные запрещены (кроме уже существующих).

- Для сортировки нельзя использовать встроенные функции типа qsort(), system("sort ...") и т.д.

- Сложность сортировки индивидуальных файлов должна быть < O(N^2) (например, нельзя использовать сортировку пузырьком).

- Суммарное время работы всей программы ограничено. (Для теста можно использовать 6 файлов, каждый с 40к чисел, время не должно занимать больше одной секунды).

- Работа с файлами должна быть

  - либо через числовые файловые дескрипторы и функции open() / read() / write() / close(),

  - либо через FILE* дескрипторы и функции fopen() / fscanf() / fprintf() / fclose(). Нельзя использовать std::iostream, std::ostream, std::istream и прочий STL.

- корутины должны переключаться. Делать так называемые 'yield', 'илды'.

Послабления:

- Числа помещаются в 'int'.
- Можно полагать, что все файлы помещаются целиком в память, даже все одновременно.
- Финальный шаг - само слияние сортированных массивов – можно делать прямо в main() снаружи от корутин.

Советы:
- Вы можете найти больше информации о разных новых функциях при помощи консольной команды 'man'. Наример, 'man read' (или 'man 2 read') напечатает документацию функции 'read()'.
  'man strdup' расскажет больше про функцию 'strdup()'. Таким же образом можно искать другие встроенные функции.

- Шаги, которым можно следовать, если не знаете, с чего начать:

  - Реализовать обычную сортировку одного файла. Без корутин, без множества файлов. Просто прочитать и отсортировать один файл.
    Протестируйте этот код.

  - Расширьте свой код, чтобы теперь он сортировал много файлов через сортировку слиянием. Без корутин. Проверьте свой код на реальных тестах из задания. Когда он заработает, вы сможете сконцентрироваться на добавлении корутин, и не тратить время на отладку одновременно и корутин, и сортировки, и работы с файлами.

  - Встройте в свой код корутины.

Предусмотреть в выводе: суммарное время работы программы, время работы и количество переключений каждой корутины. Учесть, что время работы корутины не включает ее время ожидания. То есть пока она спала во время coro_yield(). Вы должны остановить таймер работы корутины прямо перед coro_yield() и возобновить его сразу после.

Контрольные вопросы:
1) Что такое демон?
2) Для чего в операционной системе Linux применяется подсистема systemD?
3) Какие типы юнит-файлов вы знаете?
4) С помощью каких команд осуществляется управления демонами?
5) Чем корутина отличается от потоков?
6) Почему корутины должны быть реализованы как выделенная языковая возможность?

7) Какая польза от корутины?