Файловые системы в контейнерах

1. Введение

Контейнеры — это не просто процессы с изолированным окружением, но и **иерархия файлов**, построенная на основе образов. Файловая система играет ключевую роль в том, как контейнер работает, как быстро запускается и насколько эффективно расходует ресурсы.

🎁 Как устроено хранение в контейнерах

- Каждый контейнер создаётся на базе **слоёв образа** (read-only).
- Изменения вносятся в **верхний слой**, который записывает всё новое и удалённое.
- Такой подход требует **Copy-on-Write (CoW)** для экономии пространства и скорости запуска.

Слои контейнера: read-only + writable верхний слой

🔍 Зачем использовать специфические файловые системы

- ФС с поддержкой CoW позволяет эффективно работать с многослойными образами.
- Некоторые драйверы хранилища требуют специфических возможностей от ФС (snapshots, subvolumes).
- На выбор ФС влияет производительность, надёжность и гибкость монтирования слоёв.

💡 **Copy-on-Write (CoW)** — это механизм, при котором изменения записываются только при необходимости, не копируя все данные. Это позволяет экономить место и ускоряет запуск контейнеров.

🗂️ Файловые системы с поддержкой CoW

- **overlay2** (на базе OverlayFS) — встроена в ядро Linux, дефолт в Docker/Podman.
- **Btrfs** — CoW, снапшоты, субтомы, сжатие, подходит для контейнерных хостов.
- **ZFS** — расширенные возможности хранения и защиты, особенно для серверов.
- **AUFS** — устаревшая, но ранее популярная в Docker (до overlay2).

⚙️ Файловая система контейнерного хоста влияет на:
— производительность I/O;
— скорость запуска контейнеров;
— поддержку снапшотов и откатов;
— надёжность хранения данных.

2. Слои образов и Copy-on-Write

Контейнерная файловая система построена на слоистом принципе. Это позволяет экономить место, ускорять развёртывание и использовать общий кэш между разными контейнерами.

Базовая структура слоёв

- **Base Image (базовый слой)** — например, `ubuntu:22.04`. Он содержит стандартную файловую систему.
- **Intermediate layers** — каждый шаг в `Dockerfile` добавляет read-only слой (например, `RUN apt install`).
- **Writable layer (верхний слой)** — создаётся при запуске контейнера. Все изменения идут сюда.

Модель CoW: read-only слои + writable верхний слой

📄 Как работает Copy-on-Write

При изменении файла в контейнере:

1. Файл остаётся в read-only слое.
2. Когда приложение пишет в него — создаётся **копия в верхнем слое**.
3. Дальнейшие изменения затрагивают только эту копию.

Это позволяет десяткам контейнеров использовать один и тот же образ, не занимая место на диске.

💡 Пример: запуск контейнера и изменение файла

```
# Создаём и запускаем контейнер
docker run -it ubuntu bash

# Внутри контейнера создаём файл
echo "Hello" > /hello.txt
```

Этот файл появится **только в верхнем слое** текущего контейнера. При удалении контейнера он будет потерян, но base image останется неизменным.

⚠ CoW не означает автоматическое создание снапшотов — это лишь механизм хранения изменений поверх базовых слоёв.
💡 Некоторые ФС, такие как **Btrfs** и **ZFS**, реализуют CoW на уровне всей файловой системы, позволяя эффективно работать с контейнерами без дополнительного драйвера.

3. OverlayFS / overlay2

OverlayFS — это модуль ядра Linux, реализующий простую, но эффективную **CoW-файловую систему** путём объединения нескольких директорий в одну "сложенную" структуру. Версия `overlay2` стала **дефолтным драйвером хранения** в Docker и Podman начиная с Linux 4.x.

🔧 Как работает OverlayFS

OverlayFS объединяет несколько слоёв:

- `lowerdir` — один или несколько read-only слоёв (например, базовый образ);
- `upperdir` — записываемый слой, куда идут все изменения при работе контейнера;
- `workdir` — техническая директория для внутреннего состояния CoW;
- `merged` — результирующий mount, видимый как единая ФС контейнера.

```
# Пример ручного монтирования overlay:
sudo mount -t overlay overlay \
 -o lowerdir=/image/layer1:/image/layer0,upperdir=/container/changes,workdir=/container/work \
 /container/merged
```

💡 **Docker** и **Podman** используют `overlay2` как драйвер хранения по умолчанию на большинстве современных систем (Ubuntu, CentOS, Fedora, Arch и др.).

✔ Преимущества overlay2

- 📦 **Встроено в ядро** — не требует внешних модулей или патчей.
- ⚙️ **Простота и стабильность** — поддерживается upstream, работает «из коробки».
- **Не требует отдельного раздела** или специальной ФС (работает поверх ext4, xfs и т.д.).
- 🚀 **Быстрый запуск контейнеров** за счёт быстрой сборки слоя "merged".

⚠️ Недостатки overlay2

- 🔕 **inotify и overlay:** не всегда корректно работает слежение за файлами (например, при hot-reload).
- **Кэш inode не сохраняется** между слоями, что влияет на производительность больших директорий.
- 🧱 **Нет встроенного сжатия или дедупликации** (в отличие от Btrfs/ZFS).

Вывод: `overlay2` — оптимальный выбор по умолчанию: лёгкий, быстрый, без сложных зависимостей. Но в сценариях с интенсивной файловой активностью или inotify-heavy приложениями стоит рассмотреть Btrfs/ZFS.

4. AUFS (устаревший, но встречается)

AUFS (Another Union File System) — файловая система с поддержкой объединения слоёв, использовавшаяся в Docker до 2016 года как основной механизм хранения. Она позволяла эффективно собирать контейнеры из нескольких read-only и writable слоёв.

⚠️ AUFS не входит в основное ядро Linux (*mainline*) и требует ручного включения/патча ядра.

🔧 Историческая роль AUFS

- Docker использовал AUFS по умолчанию до появления `overlay2`.
- Работал на системах с **старым ядром (2.6.x–3.x)**, где OverlayFS ещё не был доступен или стабилен.
- Поддерживал объединение до **128 слоёв** в один namespace.

✔ Преимущества AUFS

- **Гибкая работа с множеством слоёв** — полезно для сложных образов.
- 📄 **Поддержка whiteout-файлов** — механизм удаления файлов из нижележащих слоёв.
- 🔀 Возможность объединения произвольных директорий в одну виртуальную.

✖ Почему AUFS больше не используется

- **Сложность сопровождения:** требует патча ядра и ручной сборки модулей.
- ⊘ **Не включён в ядро Linux** — значит, не сопровождается основными разработчиками.
- 🎁 **Заменён на OverlayFS**, который включён в ядро и проще в использовании.

💡 Некоторые старые образы Docker или специализированные системы (например, `Docker-in-Docker` на старом ядре) могут до сих пор использовать AUFS. В новых системах его применять не рекомендуется.

5. Btrfs как backend для контейнеров

Btrfs (B-tree Filesystem) — это современная CoW-файловая система, поддерживающая сжатие, снапшоты, клонирование, контроль квот и многое другое. Она может использоваться в Docker и Podman напрямую как `btrfs`-драйвер хранения.

Особенности Btrfs

- Полная **Copy-on-Write** модель на уровне всей файловой системы.
- Поддержка **субтомов** (subvolumes) — логических разделов внутри ФС, которые можно монтировать и снапшотить независимо.
- **Снапшоты** и откаты — мгновенные, ненагрузочные.
- Поддержка **встроенного сжатия** (zstd, lzo) и **qgroups** — контроля квот на подтома.

⚙️ Использование в Docker / Podman

Чтобы использовать Btrfs как драйвер хранения, необходимо:

1. Создать Btrfs-раздел или том (`mkfs.btrfs /dev/sdX`);
2. Указать `--storage-driver=btrfs` при запуске Docker/Podman, либо настроить `/etc/containers/storage.conf`;
3. Каждый контейнер будет размещаться в отдельном subvolume внутри корневого

тома Btrfs.

💡 `docker info` покажет текущий драйвер хранения. Для Btrfs это будет `Storage Driver: btrfs`.

✔ Преимущества Btrfs

- 🚀 **Мгновенное создание контейнеров** через subvolume clone.
- 🌀 **Поддержка снапшотов и откатов** без остановки контейнера.
- 🗜️ **Встроенное сжатие**: `compress=zstd`, экономия диска.
- 📊 **qgroups** позволяют контролировать объём хранения на каждый контейнер.

⚠️ Недостатки

- **Нестабильность на старых ядрах** (< 5.4) — возможны баги, особенно с кешем и снапшотами.
- **Требует отдельного раздела или тома** — не рекомендуется запускать поверх ext4.
- **× Нет поддержки в overlay2-драйвере**, несовместим с ним напрямую.

🔍 **Итог:** Btrfs — мощный и гибкий выбор для контейнерных хостов, особенно если важны снапшоты, откаты и оптимизация места. Однако требует осторожности в продакшене и достаточно свежего ядра.

6. ZFS для контейнеров

ZFS — это мощная файловая система и менеджер томов в одном. Она поддерживает проверку целостности данных, снапшоты, дедупликацию, клонирование и сжатие, что делает её отличным выбором для контейнерных хостов с высокими требованиями к надёжности и хранению.

🔧 Использование ZFS в контейнерах

- Контейнеры Docker/Podman могут использовать `zfs` как storage driver.
- Каждый контейнер размещается в отдельном **ZFS dataset** или **volume**.
- При запуске контейнера создаётся **клон снапшота**, обеспечивая быстрый запуск без копирования.

```
# Пример ручного создания пула и dataset
sudo zpool create tank /dev/sdX
sudo zfs create tank/containers
```

```
# Использование в Docker
dockerd --storage-driver=zfs
```

💡 В Podman и Docker также можно задать драйвер хранения через `/etc/containers/storage.conf` или флаг `--storage-driver=zfs`.

✔ Преимущества ZFS

- 🛡️ **Проверка целостности** (checksumming) — защита от silent corruption.
- 🗜️ **Сжатие на лету** (например, `lz4`, `zstd`), экономия места.
- 📄 **Дедупликация** (в проде — осторожно, требует много RAM).
- 🌀 **Снапшоты и клоны** — мгновенные, без копирования данных.
- 📊 **Управление квотами** на уровне datasets.

⚠️ Недостатки

- ⚙️ **Не встроен в ядро Linux** (вне mainline) — доступен через `zfs-dkms` или пакеты дистрибутива.
- 💽 **Требует выделенного пула** (zpool) — не рекомендуется использовать на обычных ext4/xfs разделах.
- **Более высокие накладные расходы** на RAM и CPU, особенно при дедупликации.
- 📋 **Сложнее в конфигурации** и обслуживании по сравнению с overlay2 или Btrfs.

💡 **ZFS** идеально подходит для серверов, CI/CD-систем, архивов и инфраструктуры, где важна целостность и контроль данных. Однако для десктопов или простых хостов может быть избыточным.

7. Сравнение драйверов хранения в Docker / Podman

Ниже представлена таблица сравнения популярных драйверов хранения для контейнерных систем. Каждый из них имеет свои особенности, ограничения и требования к файловой системе хоста.

ФС / Драйвер	CoW	Snapshot	Требует отдельного раздела	Сжатие	Поддержка в ядре
overlay2	✔	✖	✖	✖	✔

ФС / Драйвер	CoW	Snapshot	Требует отдельного раздела	Сжатие	Поддержка в ядре
aufs	✔	✖	✖	✖	✖ (патч ядра)
btrfs	✔	✔	✔ (желательно)	✔	✔
zfs	✔	✔	✔	✔	✖ (вне mainline)

💡 **Рекомендации:**

 `overlay2` — оптимален для большинства случаев (по умолчанию).
 `btrfs` — подходит для CI/CD, dev-сред и систем с необходимостью откатов.
 `zfs` — выбор для продакшн-серверов с высокой надёжностью.
 `aufs` — устарел, использовать не рекомендуется.

8. Практические сценарии

Выбор файловой системы и драйвера хранения зависит от задач, типа хоста, а также требований к надёжности, скорости откатов и объёму I/O. Ниже приведены распространённые сценарии и рекомендации.

🎁 overlay2 на Ubuntu/Debian

- Используется по умолчанию в Docker/Podman на большинстве дистрибутивов (Ubuntu, Debian, Fedora).
- Не требует специальной подготовки: работает поверх ext4 или XFS.
- Подходит для типичных нагрузок с небольшим объёмом I/O.

⚙ btrfs для embedded и CI/CD

- Позволяет мгновенно создавать, клонировать и откатывать контейнеры через subvolumes и snapshots.
- Удобен для автоматических тестов, быстрой смены окружений, сборки образов.
- Хорош для встраиваемых систем, где важна экономия места (сжатие + контроль квот).

🗂 ZFS на надёжных серверных системах

- Используется в продуктивных средах, где важна **целостность данных** и

возможность отслеживания повреждений (checksumming).
- Актуален в OpenZFS-системах, NAS-хостах, CI-серверах с высокой I/O-нагрузкой.
- Позволяет делать дедупликацию, жёсткие квоты, атомарные снапшоты и репликацию.

⊘ Когда не стоит использовать overlay2

- 🧱 Приложения с интенсивным I/O (например, БД внутри контейнера).
- 👀 Программы, активно использующие `inotify` (например, hot-reload веб-фреймворки) — могут не видеть изменения.
- 📄 Контейнеры с большим количеством мелких файлов — падение производительности.

✅ **Рекомендация:** начинайте с `overlay2`, но для специализированных сценариев не бойтесь перейти на Btrfs или ZFS — это окупается гибкостью.

9. Советы по выбору

При проектировании контейнерной инфраструктуры важно выбрать файловую систему, которая соответствует **текущим и будущим нагрузкам**. Ниже — практичные рекомендации для принятия решения.

- ✅ **Начинайте с `overlay2`**, если не уверены. Это стандарт по умолчанию в Docker/Podman, работает без особой настройки.
- 🎁 **Храните постоянные данные вне контейнера** — используйте `volume` или `bind mount`. Не полагайтесь на writable layer.
- **Btrfs и ZFS — мощный выбор** для dev/CI, снапшотов, тестовых сред и контроля за данными.
- 🔐 **Проверьте совместимость с SELinux и AppArmor** — особенно важно при использовании `overlay2`, т.к. они накладывают ограничения на mount namespaces.
- ⚙️ **Избегайте AUFS** в новых проектах — он устарел, не поддерживается в mainline-ядре.

💡 **Совет:** Файловая система — не только про производительность, но и про удобство обновлений, резервного копирования и восстановления.

10. Заключение

Выбор файловой системы и драйвера хранения в контейнерной среде — это не

просто техническая деталь, а **ключевой фактор стабильности, гибкости и производительности**.

- **Файловые системы — не просто хранилище, а часть экосистемы контейнеров.**
- ⚙️ `overlay2` **— это стандарт и безопасная точка старта**, но он не решает всех задач и может ограничивать функциональность.
- 🌀 `Btrfs` **и** `ZFS` **— это выбор для тех, кто ценит снапшоты, откаты, контроль ресурсов и целостность данных.** Они требуют настройки и понимания, но предоставляют продвинутые возможности.

✅ **Вывод:** нет универсального решения — выбирайте драйвер хранения в зависимости от нагрузки, целей, требований к отказоустойчивости и удобству обслуживания.