Backups

DevOps: "Development and Operations". The idea is that you have a single staff that does both development and the logistic stuff.

The philosophy of DevOps is that the same person should be responsible of the development and the operations of the system. Historically, the communication between the two are often not that good. If the same person knows both, they'll know when and how best to optimize.

Anyway, a simple type of version control is the concept of **backups**.

Failure Modes

Backups provide ways to recover from **failure modes**:

- **Drive failure**: the physical hardware malfunctions.
  - Total failure: the entire drive is dead.
  - Partial failure: a few blocks go bad.
  - Corrupted data with no indication (a silent and very serious error!).
- **Operational failure**: someone responsible for maintaining the software makes a mistake, like erasing data.
- **External attackers**: hackers that break into your system, encrypt all your data, and hold it ransom.
- **Data corruption**: due to power outage
- **Cosmic rays**: can cause a bit flip in the RAM or even flash on an open computer.
- etc.

> **ASIDE:** You can protect against bit flips by using **ECC RAM**. For every amount of bits, you have several **check bits** that you can use to test if a bit was flipped and correct it. All SEASnet servers and high-end work computers in general have ECC RAM. Of course, this increases the cost due to more testing required to ensure reliability.

Something to keep in mind for failure modes is not simply *what* can go wrong, but *how often* it can go wrong. This will dictate how often you should perform backups and what strategy to use.

One way to check for this is the **annualized failure rate (AFR)**. An AFR of 1% means the drive-maker thinks that if you use the drive in "the usual way" (some number of read/write operations), then on average 1% of the drives you use will fail in a year. These numbers are *estimates* based on similar drives in the past.

> **ASIDE: S.M.A.R.T.** is an interface you can use to interrogate a drive to determine things like how much it's been used and lifetime estimates.

Performing Backups

IMPORTANT: Note that you every backup procedure should have a recovery procedure. You must also TEST your recovery procedures, or you might not be able to even restore your backup!!

The "Simple Version"

Backup procedure

Periodically, make a copy of your whole *state*. Of course, this is inefficient and *doesn't scale*.

Recovery procedure

Simply go to the most recent copy you made and copy it *into* your state.

You could also hold onto more than one copy at a time and go to copies even earlier than the most recent one when recovering. For example, if an attacker broke in several days ago, you should probably back up to a copy *before* that and be wary of any changes after the break-in event.

What to Back Up?

RAM + registers vs. persistent storage? Ideally you want to recover both because that's the entire *state* of your system.

In reality, RAM + registers are hard to backup because they rapidly change and because there often aren't good software interfaces for doing this. Thus, usually we focus more on persistent storage.

Backup Levels

You can perform backups at multiple "levels", with the higher ones being more *efficient* and the lower ones being simpler and more *general*.

Application-level Backup

Each application knows how to back up its own data and can tailor it to its own needs.

File-level Backup

You save:

- Contents of the files
- Metadata of the files
- Partitioning (file system layout)

Block-level Backup

The above is built on top of a lower-level system that's **block** based. You want to save all the blocks representing the above.

All files are typically stored as arrays of blocks, with each blocks representing things like directory listing, actual file contents, etc. You don't have to worry about what each block represents - simply back up all the blocks and you will have backed up the entire system!

Practically, most systems use a combination of these levels.

ASIDE: Why split a drive into multiple areas?

Example 1-drive system:

```
+---+------+-------+------
| / | /usr | /home | etc.
+---+------+-------+------
```

Maybe `/` has stuff needed to boot the system, `/usr` has non-essential system files, and then `/home` has the user's documents. This way, even if there were a problem with the `/usr` segment, the system would still be able to boot, and you can then recover `/usr` from there.

The blocks are physically partitioned on the drive, with each block corresponding to a specific part of the file system.

Within the file system, there'll be a low-level representation of the file contents like metadata about who owns it, etc.

When to Back Up?

Do you bring the system down/idle to do backups?

- If **YES**, this greatly simplifies your backup strategy. This ensures that no one's changing the blocks *while* you're doing the backup.
- If **NO**, your backup might not match any actual state because it'll be a *mixture* of some blocks from the old state and some blocks from the new state. This is sometimes called **live backups**, and this can be a real issue.

Suppose you did something like:

```
cd /home
tar -cf /backup/file1 *
# sometime later...
tar -cf /backup/file2 *
```

This creates a huge file `/backup/file1` containing every file under the `/home` directory. But if you do work *while* tar is working, the image that `/backup/file1` represents would be a mixture.

More considerations: How often should you reclaim storage from backups? How many backup files do you retain?

Obviously at least one. If the system crashes while backing up, then you just lost your only backup. Thus, you usually strive for at least two copies.

How to Back Up Cheaply

- Do them less often.
- Do them to a cheaper device, like a hard drive or optical device.
- Do them to a remote service (this would involve some *trust* in the backup service).
- Instead of making an entire copy of your data, you do **incremental backups**.

Incremental Backups

Backup procedure

Only backing up the *changes* between the old and new changes.

- The first backup is everything
- Subsequent backups are only the differences

A block-oriented backup would only need to back up the changed blocks.

Recovery procedure

- 1st copy: recover the full backup.
- Subsequent copies: apply the differences.

A limitation is that applying differences may take a while, so typically you would backup differences and periodically backup whole copies.

You could also *update* the backup with the differences, but this is like changing history, which introduces more concerns.

How do you keep track of which blocks are changes?

One approach is that with each block, you record the timestamp of when each block was changed. You'd have to record these timestamps somewhere, which means your data would be more complicated than a simple array of blocks. Some software does this anyway, where it sets aside some space for some metadata, checksums, etc.

Another option is to use the output of `diff` instead of the contents of the entire block. This is typically done at a higher level than the block level though.

Backup Optimization

Deduplication: Copying on Write

Suppose you execute:

```
cp bigfile bigcopy
```

At the block level:

```
    bigfile         bigcopy
    v               v
+---+---+---+---+---+---+---+---+
|   | b | f |   |   | b | c |   |
+---+---+---+---+---+---+---+---+
```

You could cheat by just making `bigcopy` point to the same file object as `bigfile`:

```
    bigcopy
    bigfile
    v
+---+---+---+---+---+---+---+---+
|   | b | f |   |   | b | c |   |
+---+---+---+---+---+---+---+---+
```

From the user's POV, it *looks* like a copy even if you don't have one in reality.

When you modify `bigcopy`, your `bigcopy` just "remembers" (in its underlying data structure) which additional parts are part of the copy.

```
    bigcopy --+
    bigfile   |
    v         v
+---+---+---+---+---+---+---+---+---+
|   | b | f | m |   | b | c |   |   |
+---+---+---+---+---+---+---+---+---+
```

This has recently become the *default* behavior of the GNU `cp` command.

This is known as **lazy copying**. Eventually, when you modify every block in your copy, *then* you will have made a copy of every block. Until then, you do it **lazily**, that is, *only as needed*.

This strategy has a name: **copy on write (CoW)**.

- ➕ Speed.
- ➕ Less space.
- ➖ Not good for *backups* because if a block gets corrupted, it affects both the original and copy.
- ➕ But this also means less underlying data when performing an actual backup.

`git clone` uses this strategy! An example of a backup at the *application level* too.

Other Techniques

Compression

Often done at the block level or the file level.

You can check this. While the `-l` switch includes how many *bytes* used, the `-s` switch includes how many *blocks* are used to store the underlying file:

```
ls -ls file
```

Multiplexing

Back up many different systems to a single backup device.

Staging

Fast devices back up to slow devices:

```
flash --> disk --> optical tape
(fast)    (slow)   (even slower)
```

Data Grooming

Remove old data that you don't need anymore before backing up.

It's tedious to determine which data isn't needed anymore, so a lot of this is now done automatically. Of course, this has the downside where programs might decide to remove data that users still want.

Encryption

Guard against problems like attackers from the inside stealing and/or selling data to other agents.

Checksumming: What if your backup software is flakey?

You make checksums of your data and then back up those checksums somewhere else. When you recover, you can check if the recovered data matches the checksums.

File Systems with Backups Built-in

Where old versions of files are always available.

Versioning Approach

The file system keeps track of the old version of every file you create. It's as if:

```
$ echo x > file
$ ls -l
---------------- file;1
$ echo y >> file
$ ls -1
---------------- file;1
---------------- file;2
```

When to automatically create a new version?

- Every time the file is opened for writing, differences made, and then closed
- Maybe there's some `newversion()` system call
- In the end, the *applications* decide when to version.

Still used in some software like OpenVMS, etc.

Snapshot Approach

Every now and then, the underlying system takes a consistent snapshot of the entire file system.

It doesn't actually copy all the blocks because that would be too expensive. Instead, it uses CoW.

This is used in software like ZFS, btrfs, WAFL (SEASnet).