snapintime

Release 2.1.1

Eric Mesa

Feb 20, 2023

CONTENTS:

snapintime is meant to manage the creation, culling, and send to a remote location of btrfs snapshots.

At this point in time it creates snapshots, culls local snapshots three days old, and can btrfs send/receive to a remote btrfs subvol.

CHAPTER

ONE

USAGE

Grab config.json from the Github repo (https://github.com/djotaku/Snap-in-Time), edit it, and place it in $HOME/.config/snapintime (or /root/.config/snapintime/ if you're going to run as root)

1.1 Creating Local Snapshots

If running from a git clone:

```
pip -r requirements.txt
cd snapintime
python create_local_snapshots.py
```

If running from PyPi, run: python -m snapintime.create_local_snapshots

If you want to run it from cron in a virtual environment, you can adapt the following shell script to your situation:

```
#!/bin/bash
cd "/home/ermesa/Programming Projects/python/cronpip"
source ./bin/activate
python -m snapintime.create_local_snapshots
```

Make it executable and have cron run that script as often as you like.

For a more involved script, useful for logging, see *Putting it All Together*.

1.2 Backing Up to Remote Location

This code makes the assumption that you have setup ssh keys to allow you to ssh to the remote machine without inputting a password. It is recommended to run the remote backup code BEFORE the culling code to increase the chances that the last snapshot on the remote system is still on the local system. (This will minimize the amount of data that has to be transferred to the remote system.

```
pip -r requirements.txt
cd snapintime
python remote_backup.py
```

If running from PyPi, run: python -m snapintime.remote_backup

1.3 Culling Local Snapshots

The culling follows the following specification:

- Three days ago: Leave at most 4 snapshots behind - closest snapshots to 0000, 0600, 1200, and 1800. (implemented)
- Seven days ago: Leave at most 1 snapshot behind - the last one that day. In a perfect situation, it would be the one taken at 1800. (implemented)
- 90 days ago: Go from that date up another 90 days and leave at most 1 snapshot per week. (implemented)
- 365 days ago: Go form that date up another 365 days and leave at most 1 snapshot per quarter (implemented)

(Not going to care about leap years, eventually it'll fix itself if this is run regularly)

I recommend running culling submodule AFTER remote backup (if you're doing the remote backups). This is to prevent the removal of the subvol you'd use for the btrfs send/receive. If your computer is constantly on without interruption, it shouldn't be an issue if you're doing your remote backups daily. And why wouldn't you? The smaller the diff betwen the last backup and this one, the less data you have to send over the network. So it's more of a precaution in case you turn it off for a while on vacation or the computer breaks for a while and can't do the backups.

```
pip -r requirements.txt
cd snapintime
python culling.py
```

If running from PyPi, run: python -m snapintime.culling

1.4 Putting it All Together

Here is my crontab output:

```
0 * * * * /root/bin/snapshots.sh
@daily /root/bin/remote_snapshots.sh
0 4 * * * /root/bin/snapshot_culling.sh
```

remote_snapshots.sh:

```
#!/bin/bash

cd "/home/ermesa/Programming Projects/python/cronpip"
source ./bin/activate
echo "######################" >> snapintime_remote.log
echo "Starting remote backups" >> snapintime_remote.log
python -m snapintime.remote_backup >> snapintime_remote.log
echo "######################" >> snapintime_remote.log
#!/bin/bash
```

snapshot_culling.sh:

```
#!/bin/bash

cd "/home/ermesa/Programming Projects/python/cronpip"
source ./bin/activate
echo "######################" >> snapintime_culling.log
```

(continues on next page)

(continued from previous page)

```
echo "Starting culling" >> snapintime_culling.log
python -m snapintime.culling >> snapintime_culling.log
echo "######################" >> snapintime_culling.log
```

snapshots.sh:

```
#!/bin/bash

cd "/home/ermesa/Programming Projects/python/cronpip"
source ./bin/activate
echo "######################" >> snapintime.log
echo "Starting snapshots" >> snapintime.log
python -m snapintime.create_local_snapshots >> snapintime.log
echo "######################" >> snapintime.log
```

CHAPTER

TWO

CONFIG.JSON

An example of the config.json file:

```
{   "0":
{ "subvol": "/home",
"backuplocation": "/home/.snapshot",
"remote": "True",
"remote_location": "user@server",
"remote_subvol_dir": "/media/backups",
"remote_protected":["2022-02-08-0000", "2022-02-15-0000", "2022-04-26-0000", "2021-10-17-
↪0000", "2022-04-16-0000"]
},
"1":
{ "subvol": "/media/Photos",
"backuplocation": "/media/Photos/.Snapshots"
},
"2":
{ "subvol": "/media/Archive",
"backuplocation": "/media/NotHome/Snapshots/Archive"
}
}
```

- For the 0, 1, 2, 3, etc - there is currently (as of 0.8.1) not any inherent meaning to the fact that they are numbers. They just need to be distinct alpha-numberic sequences.
- subvol: this should be the subvolume you want to create a snapshot of.
- backuplocation: the subvolume that holds your backup subvolumes.
- remote: If set to True, an attempt will be made to backup to the remote location. Any other value or lack of this field means it will not try and backup to the remote location.
- remote_location: The username@theserver where the backup subvolumes will be sent to.
- remote_subvol_dir: Just like backuplocation, but on the remote machine.
- remote_protected: A list of snapshots you don't want to delete. For example, these may be the last snapshot you used to another remote location or maybe you're using it for an NFS share and don't want it to be culled.

CHAPTER

THREE

ORIGINS OF MY CULLING ALGORITHM

I'm basing it on a conversation I had in the btrfs mailing list. Here's how the guy who inspired me, Duncan, explained it to me:

"However, best snapshot management practice does progressive snapshot thinning, so you never have more than a few hundred snapshots to manage at once. Think of it this way. If you realize you deleted something you needed yesterday, you might well remember about when you deleted it and can thus pick the correct snapshot to mount and copy it back from. But if you don't realize you need it until a year later, say when you're doing your taxes, how likely are you to remember the specific hour, or even the specific day, you deleted it? A year later, getting a copy from the correct week, or perhaps the correct month, will probably suffice, and even if you DID still have every single hour's snapshots a year later, how would you ever know which one to pick? So while a day out, hourly snapshots are nice, a year out, they're just noise.

As a result, a typical automated snapshot thinning script, working with snapshots each hour to begin with, might look like this:

Keep two days of hourly snapshots: 48 hourly snapshots

After two days, delete five of six snapshots, leaving a snapshot every 6 hours, four snapshots a day, for another 5 days: 4*5=20 6-hourly, 20 +48=68 total.

After a week, delete three of the four 6-hour snapshots, leaving daily snapshots, for 12 weeks (plus the week of more frequent snapshots above, 13 weeks total): 7*12=84 daily snaps, 68+84=152 total.

After a quarter (13 weeks), delete six of seven daily snapshots, leaving weekly snapshots, for 3 more quarters plus the one above of more frequent snapshots, totaling a year: 3*13=39 weekly snaps, 152+39=191 total.

After a year, delete 12 of the 13 weekly snapshots, leaving one a quarter. At 191 for the latest year plus one a quarter you can have several years worth of snapshots (well beyond the normal life of the storage media) and still be in the low 200s snapshots total, while keeping them reasonably easy to manage. =:^)"

CHAPTER

FOUR

CREATE_LOCAL_SNAPSHOTS

Read in configuration file and create local snapshots.

snapintime.create_local_snapshots.**create_snapshot**(*date_suffix: str*, *subvol: str*, *backup_location: str*)

Create a btrfs snapshot.

Parameters

- **date_suffix** – a datetime object formatted to be the name of the snapshot
- **subvol** – The subvolume to be snapshotted
- **backup_location** – The folder in which to create the snapshot

snapintime.create_local_snapshots.**get_date_time**() → str

Return the current time, uses system time zone.

snapintime.create_local_snapshots.**iterate_configs**(*date_time: str*, *config: dict*) → list

Iterate over all the subvolumes in the config file, then call create_snapshot.

Parameters

- **date_time** – The date time that will end up as the btrfs snapshot name
- **config** – The config file, parsed by import_config.

Returns

A list containing return values from create_snapshot

snapintime.create_local_snapshots.**main**()

CHAPTER

FIVE

CULLING

CHAPTER

SIX

REMOTE_BACKUP

CHAPTER

SEVEN

CONFIG

Load the config file.

snapintime.utils.config.**import_config**() → dict

Import config file.

Returns
A dictionary containing configs

Raises
FileNotFoundError

CHAPTER

EIGHT

DATE

Provide date and Time Operations needed by snapintime.

snapintime.utils.date.**many_dates**(*start_date: datetime*, *interval_start: int*, *interval_end: int*) → list

Provide a list of dates within a certain range.

Used by quarterly culling and yearly culling to determine date range to cull.

Parameters

- **start_date** – The reference point for the intervals
- **interval_start** – How many days ago you want to start getting dates from.
- **interval_end** – How many days ago you want to stop getting dates from.

snapintime.utils.date.**prior_date**(*start_date: datetime*, *day: int = 0*) → datetime

Provide a prior date offset by the variable given in day.

Unintuitively, positive numbers subtract days.

Parameters

- **start_date** – The date from which you want to count back or forward.
- **day** – The number of days you want to go back.

Returns

A datetime object day amount of days in the past.

snapintime.utils.date.**quarterly_weeks**(*start_date: datetime*) → list

Provide a list of 13 weekly date lists.

Parameters

start_date – Date from which to go back a quarter.

Returns

A list of lists containing datetime objects. Each sublist represents a week.

snapintime.utils.date.**yearly_quarters**(*start_date: datetime*) → list

Provide a list of 4 quarterly date lists.

Parameters

start_date – Date from which to go back a year.

Returns

A list of lists containing datetime objects. Each sublist represents a quarter.

CHAPTER

NINE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

S

INDEX