Snapper 2.0 - Roadmap

Rumen Mitov

June 20, 2025

1 DONE [#A] Setup Snapper Project

DEADLINE: *<2025-06-16 Mon>*

`Effort: 4`

- ☒ Create run file and Makefile
- ☒ Use Goa as a build tool
- ☒ Define data structures and `Snapper` object.

2 TODO [#C] Unit Tests

DEADLINE: *<2025-06-16 Mon>*

`Effort: 4`

The following unit tests are used to check if the functions are not throwing any unexpected errors. For testing with actual virtual pages in PhantomOS check out the Integration Tests section.

2.1 DONE Snapshot creation

1. Create an array of 1000 elements containing random integers.
2. Create a snapshot of every element in the array.
3. If test is successful, there should be no errors thrown.

2.2 TODO Snapshot successful recovery

2.2.1 Test #1

Note, this test requires an archive of a generation storing integers in the range from 1 to 1000 in increasing order.

1. Create an empty array that can hold 1000 integers.
2. Recover each file from the generation into an element in the array.
3. If test is successful, the array should store all numbers from 1 to 1000 in ascending order.

2.2.2 Test #2

Note, this test requires an archive containing two generations: one invalid, the second invalid. Both generations are snapshots of an array containing all integers from 1 to 1000 in increasing order. However, the older, invalid one contains some files that are needed by the valid generation.

1. Create an empty array that can hold 1000 integers.
2. Recover each file from the valid generation into an element in the array.
3. If test is successful, the array should store all numbers from 1 to 1000 in ascending order.

2.2.3 Test #3

Note, this test requires an archive containing a generation and an invalid snapshot created after the valid generation. The valid generation should store integers in the range from 1 to 1000 in increasing order.

1. Create an empty array with capacity for 1000 integers.
2. Recover each file from the generation into an element in the array.
3. If test is successful, the array should store all numbers from 1 to 1000 in ascending order and the incomplete generation's directory should have been removed.

2.3 DONE Snapshot unsuccessful recovery

Note, this test requires an archive of a generation whose archive file has an invalid CRC.

1. Try to recover the generation.
2. If test is successful, the recovery should not be possible and an error should be written stating that the archive file is invalid.

2.4 DONE Snapshot purge

Note, this test requires an archive of two valid generations. Both generations are snapshots of an array containing all integers from 1 to 1000 in increasing order. However, the older one contains some files that are needed by the valid generation.

1. Purge the older generation.
2. Create an empty array with capacity for 1000 integers.
3. Trying to recover from the older generation should be unsuccessful.
4. Recover each file from the non-purged generation into an element in the array.
5. If test is successful, the array should store all numbers from 1 to 1000 in ascending order.

3 TODO [#B] Snapshot Creation

DEADLINE: *<2025-06-16 Mon>*

`Effort: 10`

1. If the latest generation does not have a valid archive file, delete it (the generation is incomplete).
2. Initialize a new generation directory with an RTC timestamp as the name.
3. Within the generation directory create the archive file and the snapshot directory.

4. Check if there is a valid prior generation (based on the timestamps). If there is, load the archive file's data into the `Snapper::Archiver`.

5. Let h_i := `Snapper::Archiver[i]`. If `Snapper::Archiver[i]` contains backlinks, use the *first backlink* (i.e. the earliest backlink).

6. For each $p_i \in P$ where the CRC of the file h_i does not match the CRC of p_i (or h_i does not exist):

   (a) Create new file, h_j, and save the binary contents of p_i into this new file.

   (b) Initialize the snapshot file with the new CRC of the data, a reference count of 1, and the binary data of p_i.

   (c) Update `Snapper::Archiver[i]` $\leftarrow$ *path(* h_j *)*, there *path()* is the path relative to <snapper-root>.

7. For each $p_i \in P$ where CRC of the file h_i matches the CRC of p_i:

   (a) If the file h_i has a reference count greater than or equal to **SNAPPER_REDUND**:

      i. Create a new file h_j as outlined in Step 6.

      ii. Increment the reference count for all files in `Snapper::Archiver[i]`.

      iii. Enqueue *path(* h_j *)* to `Snapper::Archiver[i]`.

   (b) If the file h_i has a reference count lower than **SNAPPER_REDUND**, increment the reference count of it and all other redundant files in `Snapper::Archiver[i]`.

8. Save `Snapper::Archiver` into the archive file and calculate its CRC.

4 TODO #B Snapshot Recovery

DEADLINE: *<2025-06-23 Mon>*

`Effort: 10`

1. Choose a generation to boot from (by default the latest one).
2. Check if the generation is valid (i.e. has an archive file with a valid CRC). If not, recovery is not possible.
3. Load the archive file of the latest valid generation into `Snapper::Archiver`.
4. For each h ∈ `Snapper::Archiver` and for each backlink, $h_i \in h$:
   (a) Check the CRC with the stored data.
   (b) If h_i does not exist or there is a mismatch with the CRC, try the next backlink.
   (c) If there are no more backlinks to check, respond according to the configured policy.
   (d) If the CRC matches h_i, load the data of h_i into the corresponding page p_i.

5 TODO #C Snapshot Purge

DEADLINE: *<2025-06-23 Mon>*

`Effort: 10`

Note, that when a file's reference count is decremented to 0, the file is removed. If a directory becomes empty as a result, it is removed.

1. Make sure the generation is valid (i.e. it has an archive file with a valid CRC).
2. If the archive file has an invalid CRC:
   (a) If **SNAPPER_INTEGR** is set to true, crash the system and ask the system administrator to replace the generation's corrupted archive file with a backup copy.

Note, that if no backup copy exists it is highly recommended to manually remove the current generation as well as all subsequent generations. Snapper can continue to function without the removal, but the broken generation and its files will never be removed. Alternatively, the administrator could manually remove the broken generation and set **SNAPPER_INTEGR** to false. That way any snapshots that relied on the broken generation will only output warnings but will not crash the system if they are unable to recover a file.

(b) Otherwise, log an error message and boot the system into a clean state.

3. If the archive file has a valid CRC:

   (a) Load the archive file into `Snapper::Archiver`.

   (b) For each entry $h \in$ `Snapper::Archiver` and for each file $h_i \in h$: decrement the file h_i's reference count.

   (c) Delete the archive file.

6 TODO #C XML Configuration Support

DEADLINE: *<2025-06-30 Mon>*

`Effort: 5`

- ☐ SNAPPER_ROOT
- ☐ SNAPPER_THRESH
- ☐ SNAPPER_INEGR
- ☐ SNAPPER_REDUND
- ☐ Retention::MAX_SNAPS
- ☐ Retention::EXPIRATION

7 TODO #C Integration Into PhantomOS

DEADLINE: *<2025-06-30 Mon>*

`Effort: 10`

8 TODO [#C] Integration Tests

DEADLINE: *<2025-07-07 Mon>*

```
Custom_id: integration-tests
Effort: 5
```

The following tests will be conducted within PhantomOS.

- ☐ Snapshot creation
- ☐ Snapshot recovery
- ☐ Snapshot purge

9 TODO [#C] Demo Application to Demonstrate Snapper

DEADLINE: *<2025-06-14 Sat>*

```
Effort: 15
```

Create a graphical application to demonstrate Snapper's capabilities. Perhaps a weather app that graphs real-world data? The application state should be taken a snapshot of which will be restored after a system reboot.