Boot-to-snapshot design v0.2

Bryn M. Reeves <bmr@redhat.com>,
Marion Csontos <mcsontos@redhat.com>,
Zdenek Kabelac <zkabelac@redhat.com>.

Last modified: 01/06/17

Contents

I Background **3**

1 Problem Statement **3**
1.1 Booting snapshots . . . 4
1.1.1 Booting snapshots of LVM2 volumes . . . 5
1.1.2 Booting snapshots of BTRFS volumes . . . 5
1.2 Core Goals . . . 6
1.2.1 Goal: enable use of snapshot boot capability for advanced users . . . 6
1.2.2 Proposal: document kernel command line parameters for booting snapshots 6
1.2.3 Goal: Enable easy selection of bootable snapshots . . . 6
1.2.4 Proposal: Make snapshots accessible from the boot menu . . . 6
1.2.5 Goal: Automate the management of boot entries for snapshots . . . 7
1.2.6 Proposal: Develop an integrated snapshot management tool . . . 7
1.3 Additional Goals . . . 7
1.3.1 Goal: allow substitution of selected VFS mounts for snapshot versions . . 7
1.3.2 Proposal: introduce a mechanism for boot time file system substitution. . 7
1.3.3 Goal: Simplify creation of point-in-time snapshots of multiple file systems 8
1.3.4 Proposal: Create multiple snapshots simultaneously as a coherent set . . 8
1.4 Future extensions . . . 8
1.4.1 Goal: Simplify management of large numbers of snapshots . . . 8
1.4.2 Proposal: Provide policy controlled tools for snapshot life cycle management 8
1.4.3 Goal: Simplify file operations on inactive snapshot volumes . . . 9
1.4.4 Proposal: Implement familiar file operations for snapshots . . . 9

2 Linux boot process management **9**
2.1 Early user space interaction . . . 10
2.2 Boot loader entries . . . 11
2.3 Bootloader abstractions . . . 11
2.3.1 Grubby . . . 12
2.3.2 perl-Bootloader . . . 12
2.4 BootLoader Specification . . . 12
2.4.1 BLS configuration snippets . . . 13
2.4.2 Proposed BLS enhancements (i) . . . 14
2.4.3 Proposed BLS enhancements (ii) . . . 15
2.5 Alternate proposals . . . 15
2.5.1 Directly booting LVM2 volumes and supporting "/boot" on LVM2 devices 15
2.5.2 Adopting an existing bootloader abstraction . . . 16

3 Existing solutions **16**
3.1 Snapper . . . 17
3.2 Solaris ZFS and `beadm` . . . 17
3.3 Stratis Storage . . . 18

4 Requirements **18**
4.1 Boot Manager requirements . . . 18
4.1.1 Ability to create BLS compliant boot loader entries . . . 18
4.1.2 Ability to remove BLS boot loader entries . . . 18
4.1.3 Ability to list the available BLS boot loader entries and their properties . 18
4.1.4 Ability to display and modify the selected default boot entry . . . 19
4.2 File system substitution requirements . . . 19
4.2.1 Accept file system substitutions on the kernel command line . . . 19
4.2.2 Perform substitutions before general system initialisation . . . 19
4.2.3 Support clean shut-down of substituted mount points . . . 19
4.3 Snapshot Manager requirements . . . 19
4.3.1 Allow creation and removal of snapshots using `LVM2`, `BTRFS` or `Stratis`. 19
4.3.2 Automatically create boot configuration for snapshots of the root file system 19
4.3.3 Support expiry of snapshots by user specified policy . . . 19
4.3.4 Simultaneous creation of snapshots of multiple volumes . . . 20
4.3.5 Snapshot file operations . . . 20

II Solution overview **20**

5 Introduction **20**

6 Boot Manager **21**

7 File System Substitution **21**
7.1 Direct modification of `fstab` . . . 21
7.2 Substituting `fstab` via bind mounts . . . 22
7.3 Substituting individual mounts via bind mounts . . . 22

8 Snapshot Manager **23**

III Implementation **23**

9 Software Components **23**

10 Development languages **23**
10.1 Boot Manager . . . 24
10.2 File System Substitution . . . 24
10.3 Snapshot Manager . . . 24

11 Library Interfaces **24**

About this document

This document is written using LyX 2.2.2.

Please contact the development team using the dm-devel <dm-devel@redhat.com> mailing list with any questions, corrections or suggestions.

Executive Summary

Existing boot loader management tools are not well adapted to the problem of supporting multiple, bootable snapshots of the root file system, and have no support for managing snapshots of auxiliary file system mounts. We propose the creation of a new *Boot Manager* tool to address the problem of maintaining boot loader entries corresponding to snapshots of the system state, and a high-level *Snapshot Manager* to both automate and abstract the low-level detail of snapshot creation and boot entry management, and to provide additional value-added functionality such as snapshot meta data and life cycle management, and high-level operations on snapshot content.

Part I
Background

1 Problem Statement

Linux systems today have the ability to take a snapshot of file system state, including that of the root file system, using a number of different technologies:

- Classic device-mapper *copy-on-write* snapshots.
- Device-mapper thin-provisioning snapshots (*thin snapshots*).
- BTRFS *subvolume* snapshots.

A snapshot preserves the state of the file system at the time it is created. A *read-only* snapshot provides a persistent record of the file system state at the time of creation while *read-write* snapshots allow the history of the file systems to diverge over time, as different patterns of writes occur in the origin and snapshot volumes.

Currently the most widely used mechanism for creating snapshots using the Linux `device-mapper` is the `LVM2` suite of volume management tools. Although it is possible to create both classical and thinly provisioned `device-mapper` snapshots manually using the `dmsetup` program, this type of configuration is not widely used in practice and is not further considered here.

An alternative implementation of `device-mapper` snapshots and other volume and file system management techniques for Linux, STRATIS[1], has been proposed and may in time provide another means to implement volume snapshots.

Snapshots may be used to effect a number of useful administrative tasks, for example:

1. Creation of a snapshot (read-only or read-write) for consistency checking, and to determine if action is needed on the live data.
2. Point-in-time snapshots for backup and recovery.
   (a) Creation of a stable disk image to be backed up to other media.

   (b) Regular scheduled snapshots on a rolling or persistent basis for "live" backup, reference, and recovery.

3. Very frequent snapshots to catch brief events or quickly changing content.
4. Before-and-after snapshots of system state for testing and roll back.
5. Creation of temporary environments for testing updates and changes, to be either discarded or merged into the main volume at a later time.
6. Creation of stable, re-usable test configurations for quality assurance and continuous integration.
7. Preparation of standardised *base image*s for further customisation.

When used in this manner snapshots avoid the expense involved in creating, or retrieving, a backup of the entire file system state or selected parts of it.

For example, the administrator may create one snapshot prior to applying software updates, and a second once the updates have been successfully applied. If problems emerge during testing or later, the prior system state may be immediately inspected via the snapshot, and rolled back to if necessary. Alternately, a single snapshot may be taken before the update operation and rolled back to in the event of failure.

Additionally, it is possible to interrogate snapshots to perform operations such as retrieving a file's content at an earlier moment in time, performing a "`diff`" operation between different versions of a file, or even to perform actions within a snapshot image, such as running an arbitrary command and adding or removing files or directories.

1.1 Booting snapshots

In many situations is is useful to be able to boot the system into the state represented by one or more snapshots of the root file system and other system volumes.

This allows inspection of the state of the system at the time the snapshot was created, as well as isolated testing of potentially destructive software updates, configuration changes, and other administrative actions:

1. Booting into a snapshot to inspect prior system state
   (a) To reference an earlier configuration, or software update level while troubleshooting
   (b) To retrieve data files, or information from past system state[1]
2. Boot into a snapshot to test updates and other changes
   (a) For distributions that reboot to apply updates
   (b) To test updates before accepting (merging) into the main image
   (c) Booting a system to safely test configuration changes, scripts, and other potentially destructive modifications on a snapshot of live data.
3. To recover from failed updates and other system changes
4. To create a new virtual or physical machine image from an existing image or template
5. Repeatable software testing of boot-time components from a consistent, reproducible image

[1] For example data stored within encrypted volumes in a nested storage configuration, or information from a database that cannot be retrieved from the files offline.

It has been technically possible for a Linux system to boot into a previously created snapshot for a number of years, but currently it is not possible for users of common distributions to boot into an arbitrary snapshot image of the system without modifying system managed files, and providing specific kernel command line parameters to select and enable the snapshot volumes.

Although the options themselves are relatively straightforward for one-off use cases, they require knowledge of the snapshot name or identifier in advance, as well as correlation of the root device specification and snapshot specification, and no tooling currently exists to either detect suitable snapshots of the system, and create appropriate boot entries for them, or to manage those entries at the time of snapshot creation or removal.

The syntax required to boot a system using different snapshot implementations (`BTRFS`, `LVM2`, `Stratis`) necessarily differs due to differences in the underlying snapshot mechanisms: for example, all snapshots in LVM2 have a unique name used to identify them (qualified by the volume group where required), while snapshots in a BTRFS volume are identified by a unique numeric identifier

1.1.1 Booting snapshots of LVM2 volumes

To boot a system that uses the `dracut[3]` tool to generate `initramfs` images using LVM2 snapshots, the user must specify the root device to be used via the `root=` kernel command line parameter, and the snapshot logical volume to be activated via `the rd.lvm.lv=. dracut` parameter.

For example:

```
root=/dev/mapper/vg00-lvol0 rd.lvm.lv=vg00/lvol0
```

Note that current syntax requires that the device be specified twice: firstly as a device node or symbolic link path in "`/dev`" to set the system root device, and then a second time as a direction to early user space to activate the appropriate device (this time using LVM2 "`VG/LV`" notation).

A bug currently exists in `dracut` (dracut-044) that prevents this syntax from activating LVM2 thin snapshots: this is because these snapshots have the *activation skip*[2] flag set by default. A pull request has been submitted to make this behaviour consistent with classic LVM2 snapshots when the `rd.lvm.lv` syntax is used (since this requests activation of a specific, named logical volume)[4].

1.1.2 Booting snapshots of BTRFS volumes

Similarly for BTRFS, the root device is specified as before, and the snapshot's `subvolumeid` is given using the `rootflags=` command line parameter:

```
root=/dev/sda2 rootflags=subvolid=262
```

The identifier passed for `subvolid` is the `BTRFS` subvolume identifier: an opaque, numeric identifier assigned by the file system that uniquely identifies a particular subvolume within a larger BTRFS volume.

BTRFS snapshot volumes can also be specified using the `subvol` path, a path in the volume namespace that resolves to a snapshot subvolume:

```
root=/dev/sda2 rootflags=subvol=/snapshots/20170528-1
```

When either flag is present the value overrides the superblock *default subvolume* setting, forcing the specified subvolume to be mounted instead.

[2] The ability to skip activation of certain logical volumes (marked with the *activation skip* flag) was introduced in LVM2.02.99.

1.2 Core Goals

A number of goals and concrete proposals for the current work are suggested. Core aims are labelled separately from other features that are of additional benefit, but do not form a necessary part of the basic snapshot booting proposal.

It is expected that the core and additional goals can be delivered as a number of discrete milestones over a time-scale of one to two years. This is an aggressive and ambitious schedule and progress towards milestones will need to be carefully tracked to ensure delivery of core functionality on time.

1.2.1 Goal: enable use of snapshot boot capability for advanced users

The capability to boot into a selected snapshot of the root file system already exists but is poorly documented and not widely known. In the shortest possible term this can be improved by taking simple steps to raise awareness of this mechanism and to assist users in successfully creating manual configurations.

1.2.2 Proposal: document kernel command line parameters for booting snapshots

By explicitly documenting the ability to boot into a snapshot of the root file system, in product content, tool documentation, and elsewhere, the functionality can be made available to a wider audience with minimal effort. This will also create additional interest in the type of higher-level management tools we hope to introduce as part of this work.

Although this is not a configuration method that we wish to promote, or to advise users to deploy, not documenting it at all seems incomplete, and inconsistent with other aspects of system documentation (for instance, the "`kexec`" facility, which is fully documented although we only support, and encourage users to configure it, via the "`kdump`" tools, or the "`iptables`" firewall, which we strongly encourage users to configure via higher-level mechanisms such as "`system-config-firewall`" and "`firewalld`").

1.2.3 Goal: Enable easy selection of bootable snapshots

Although it is possible to boot a Linux system into various snapshot configurations, few administrators are aware of, or make use of this functionality since it is poorly documented and relies on a detailed knowledge of kernel command line parameters, system device, and VFS configuration.

With appropriate tools these powerful features would be accessible to a much wider audience of system administrators, users and developers, as well as to teams developing projects using Linux as a base operating system.

1.2.4 Proposal: Make snapshots accessible from the boot menu

The bootloader menu provides a familiar mechanism for selecting among the available bootable environments on the system. By adding the ability to automatically create and manage boot entries for snapshot volumes, without the need for very detailed knowledge of either boot loader, kernel command line, or system configuration, the feature is made accessible to the widest audience of system administrators and users.

Using well-known technology such as the `grub` menu maximises the administrator's familiarity with the new feature and enables the use of existing skills to further customise the configuration (for example, interactive edits to the bootloader entry).

By creating tooling to exploit existing features of the operating system to enable easy generation of bootloader configuration for snapshots created using any technology (`LVM2`, `Stratis` or `BTRFS`) the need to create additional, LVM2-specific, tooling is avoided and the maintenance burden is spread over a greater number of components.

1.2.5 Goal: Automate the management of boot entries for snapshots

Once tooling has been established to allow bootloader entries for snapshots to be created and managed, it is possible to further simplify the administrator's task by automating the process of adding and removing the required bootloader configuration when snapshots are created or removed.

This may be done either by integrating the boot manager with the various tools responsible for snapshot management (`btrfs(8)`, `lvcreate(8)`, `lvremove(8)` etc.), or by creating a new tool that will perform the snapshot operation itself, using the appropriate tool for the type of volume being operated upon, before calling into the boot manager to update bootloader configuration

1.2.6 Proposal: Develop an integrated snapshot management tool

Since there are multiple snapshot mechanisms (LVM2, BTRFS and potentially Stratis), and since adding support for updating boot manager configuration would require changes to the tools for each of these mechanisms, it is proposed that a new, higher-level snapshot management tool be created to perform the task of creating and removing snapshots of the required type, and updating the boot configuration as appropriate.

As well as avoiding the need to modify the tools of every snapshot implementation this creates a consistent and uniform experience for users, independent of the particular snapshot technology in use. This benefits users migrating from one technology to another (for example, from a distribution using BTRFS for the root file system to one using LVM2, or a move from classic LVM2 snapshots to LVM2 using thin provisioning), allows the creation of consolidated documentation and other learning materials, and simplifies the extension of the feature to additional back-end mechanisms (for example, STRATIS).

This addition of a new component for snapshot management creates an integration point for further enhancements, for example adding policies to control various aspects of snapshot maintenance, and interfaces to provide monitoring and statistical information on snapshot usage.

1.3 Additional Goals

Additional goals, beyond the core functionality required to boot into a snapshot of the root file system, are presented in this section.

These items are not essential to the aim of allowing a user to boot into a selected snapshot of the root file system but provide considerable added value, allowing the user to more precisely recreate system state by including auxiliary file systems in the snapshot boot configuration.

1.3.1 Goal: allow substitution of selected VFS mounts for snapshot versions

In order to assemble a coherent view of the system corresponding to an earlier point in time, it is necessary to be able to substitute other VFS mounts with their snapshot equivalents during boot.

Addressing this goal exceeds the capabilities of Snapper and brings the solution closer to feature parity with ORACLE Solaris/ZFS in terms of the ability to quickly recover an earlier system state, including auxiliary file system content.

1.3.2 Proposal: introduce a mechanism for boot time file system substitution.

A mechanism will be created to allow a list of file system mount points to be given for substitution with specified snapshot versions, at the time the system is booted. The syntax must extend to LVM2, BTRFS and Stratis snapshot volumes, and should allow VFS mount options to be inherited from the existing configuration for the mount point under substitution, or specified at the time of substitution.

1.3.3 Goal: Simplify creation of point-in-time snapshots of multiple file systems

Currently no mechanism exists to allow an administrator to simultaneously snapshot multiple devices or file system mount points, and to combine these snapshots together into a coherent view of prior system state.

It should be possible for the administrator to specify mount points to be included or excluded from the configuration (for example to allow sharing of data between snapshot instances, or to allow logs to be written to a common location).

This is necessary in order to be able to boot the system into an actual prior state, as recorded in the snapshots. This is the case because as well as application state, modifications to the root file system may necessitate changes to data hosted on other volumes that together comprise the overall operating environment of the system (for example: an update to a database package installed to the root file system, that then requires changes to the format of files stored in other volumes, or an update of a system to some new internal baseline image, with corresponding changes to data deployed in other system volumes).

1.3.4 Proposal: Create multiple snapshots simultaneously as a coherent set

The Snapshot Manager tool will allow the user to specify multiple volumes for which snapshots will be created simultaneously, and will create a corresponding boot entry that includes the necessary file system substitution if the root file system is included.

1.4 Future extensions

Once the snapshot boot mechanism and snapshot management tools are established it becomes possible to deliver additional snapshot driven features beyond the basic capacity to boot the system into a prior state: although unrelated to the boot capability, these represent potentially significant added functionality and value.

1.4.1 Goal: Simplify management of large numbers of snapshots

When regular snapshots of the system are taken, or when snapshots are preserved over a long period of time, large numbers of snapshot volumes may accumulate on the system. This is a problem for several reasons: large numbers of entries clutter system menus and tool output, making selections more difficult to navigate while increasing the risk of error, as well as lengthening the time taken for the user to make a correct selection,

These effects are especially acute in restricted terminal environments used for system recovery, such as `systemd`'s emergency mode or the shell spawned by `dracut` when an early user space error occurs.

Additionally, in the case of classic `device-mapper` snapshots, I/O performance degrades when many snapshots of the same origin exist due to the non-shared copy-on-write store design.

By not addressing these issues in generic features, the cost is pushed onto users of the snapshot facility, who will need to implement appropriate policies themselves, either manually or via automation.

1.4.2 Proposal: Provide policy controlled tools for snapshot life cycle management

Once a tool exists to create and remove system snapshots and their corresponding boot configuration, it becomes a relatively simple step to apply policies to automate some aspects of that management, for example creating snapshots to a particular schedule, around certain events, or expiring snapshots by age, count or some other applicable mechanism.

By providing hooks that the administrator, or other applications can use to modify policy actions the flexibility of these mechanisms can be increased.

1.4.3 Goal: Simplify file operations on inactive snapshot volumes

In addition to booting into snapshots to perform administrative tasks, it is useful in some instances to be able to inspect these volumes from the running system, and even to perform actions within them (running a command, accessing file content).

Normally this would involve the administrator activating the volume, determining a path where it can be temporarily mounted, and issuing a command to mount the device before carrying whatever task is needed, and then reversing these steps once the volume is no longer required.

Providing automated mechanisms to allow these operations avoids the need to disrupt the system by rebooting for simple tasks, and simplifies the execution of these tasks on the running system by automating snapshot activation and mount operations.

1.4.4 Proposal: Implement familiar file operations for snapshots

A set of operations that can be applied to snapshot volumes is proposed. These operations are based on familiar UNIX command idioms and provide a means to access and inspect snapshot content from the system.

df report snapshot disk space usage.

du estimate snapshot file space usage.

cp Copy a file between the snapshot and the file system.

cat Concatenate snapshot file content and print to standard output.

diff compare file content between the snapshot and file system line-by-line.

exec Execute the specified command in the snapshot name space.

2 Linux boot process management

A Linux operating system instance is loaded in multiple phases:

Firmware Platform-specific initialisation carried out by embedded software.

Bootloader Selection, loading, and initialisation of a kernel image and auxiliary images.

Kernel Kernel and driver initialisation, VFS initialisation, process initialisation.

Early Userspace Selection and mounting of the root file system.

Init Control is passed to `init` daemon: auxiliary file systems mounted, services started.

Aside from the first phase, which is implemented by hardware specific embedded software, these phases are carried out by components provided by the operating system (in environments which permit booting more than one operating system the administrator may choose to create separate bootloader environments for each, or to manage a system wide, shared instance: no single convention for managing this situation has yet been adopted by the community).

The earliest phases of the Linux boot process (from the end of firmware initialisation to the point that control is handed to the kernel) have long lacked standardisation of both implementation and configuration.

Each distribution tends to carry its own tweaks, hacks and management scripts[12][13] to augment the bootloaders provided by upstream projects, to the point that in some cases a distribution's bootloader may be considered an independent fork of the upstream project.

Since today there is very little cooperation between distributions on shared management of a single, system-wide "`/boot`", administrators who choose to configure the system in this way

are left with the task of ensuring that the various installations do not conflict, and that there is a single, coherent bootloader installation visible to firmware at boot time.

Bootloaders in use today include:

elilo Bootloader for IA-64 based Intel systems (legacy)

lilo Bootloader for x86/x86_64 based PC systems (legacy)

grub1 Bootloader for x86/x86_64 based PC systems (legacy)

grub2 Cross-platform bootloader (x86/x86_64, IA-64, ARM/AARCH64, PowerPC)

yaboot Bootloader for PowerPC systems (legacy).

zipl Bootloader for IBM z Systems

Current RHEL and Fedora releases (Red Hat Enterprise Linux 7, and Fedora 16 onwards) use the grub2 boot loader on all supported architectures except for IBM System z. The current releases of Ubuntu (Ubuntu 16.04) and OpenSuSE (the rolling Tumbleweed release and OpenSuSE Leap 42.2) both use the grub2 boot loader by default (although do not support the *BootLoader Specification* extensions discussed in 2.4).

2.1 Early user space interaction

The Plymouth[14] graphical boot system provides a mechanism to present a graphical or text mode progress indicator to the user during system boot. Plymouth has been used as the graphical boot agent in Red Hat distributions since Fedora 10 and Red Hat Enterprise Linux 6, and is currently used in Ubuntu (16.04) and OpenSuSE (Tumbleweed / Leap 42.2)

Plymouth consists of a daemon, plymouthd, which manages the display and handles requests from clients, and a command line and library interface used to notify the daemon of state changes, and to request user interaction via the console.

In addition to the progress indication and logging services provided by plymouth, limited interaction with the user is permitted in order to obtain answers to simple question/response prompts, or to provide specific information (typically authentication or decryption passwords).

The interaction verbs supported by current versions (0.8.9) of plymouth are[15]:

ask-for-password Ask user for password

ask-question Ask user a question

display-message Display a message

hide-message Hide a message

watch-keystroke Become sensitive to a keystroke

ignore-keystroke Remove sensitivity to a keystroke

These verbs accept the following options:

```
Options for ask-for-password command:
  --command=<string> Command to send password to via standard input
  --prompt=<string> Message to display when asking for password
  --number-of-tries=<integer> Number of times to ask before giving up (requires --command)
  --dont-pause-progress Don't pause boot progress bar while asking

Options for ask-question command:
  --command=<string> Command to send the answer to via standard input
  --prompt=<string> Message to display when asking the question
```

```
  --dont-pause-progress Don't pause boot progress bar while asking

Options for display-message command:
  --text=<string> The message text

Options for hide-message command:
  --text=<string> The message text

Options for watch-keystroke command:
  --command=<string> Command to send keystroke to via standard input
  --keys=<string> Keys to become sensitive to

Options for ignore-keystroke command:
  --keys=<string> Keys to remove sensitivity to
```

While it is technically possible to envisage using these primitives to implement a snapshot selection mechanism during early user space, the available primitives are limited and would make construction of menus having numerous entries awkward and clumsy, and enforces a question-response type of interaction (without the ability to, for example, scroll through the available entries).

Discussions with the original authors of Plymouth indicate that there is no interest in extending the interaction capabilities of the tool at this time: avoiding complexities such as internationalisation and font configuration during this phase of boot form part of the design goals of Plymouth and there is active hostility towards enhancing this mechanism to support more complex interaction.

2.2 Boot loader entries

Although each bootloader has its own configuration mechanism and syntax, generally a *boot loader entry* may be considered a tuple of (*kernel, initramfs, arguments*, [device tree])[3], or (*kernel version, arguments*), with the precise image file names being formed from a template and the *kernel version* value. For architectures that support the use of a *device tree* (notably `ARM/AARCH64`), there is an additional tuple member to specify this parameter.

Boot loaders typically present the available boot entries in menu form, allowing the user to select from the available entries or to accept the default. This is a well known mechanism for controlling the system boot process although the details and precise capabilities vary among the existing bootloader implementations.

2.3 Bootloader abstractions

Several attempts have been made to create higher level abstractions over the multiple bootloader configuration formats, with varying degrees of success. Typically each distribution, or family of distributions, has a favoured tool which is used by the operating system installer, update tools, and package manager installation scripts to add, remove, or modify bootloader configuration.

These tools are generally designed to accommodate automated installation and update use-cases, for example: adding a boot configuration for a new kernel package, or removing all boot configurations corresponding to a specified kernel package.

Two examples are considered in detail here: the RED HAT `grubby` package, and SUSE's `perl-Bootloader` modules.

[3]Some environments, for example, the Xen paravirtualised hypervisor, which implements the multiboot standard[22], require additional boot tuple values to specify auxiliary images to be loaded by the bootloader.

2.3.1 Grubby

The grubby[16] project has been used in RED HAT distributions since the early 2000s. It consists of a C program, grubby[17], with a command line interface, and a shell script, new-kernel-pkg[12], intended to be called from packaging scripts.

The grubby program understands configuration formats for a number of bootloaders (grub1, grub2, lilo, elilo, yaboot, and zipl), and is able to add, remove, modify, and display information for bootloader entries including the currently selected default entry.

Given a kernel, initramfs image, command line arguments, and title, the command will create a new entry, optionally substituting missing elements from a template generated from existing entries.

The tool is also able to adjust the default entry, and to probe for the presence of some supported bootloaders.

The bootloader abstractions implemented by grubby attempt to hide details of the individual formats (for instance, different keywords used to specify the kernel or initial ramfs image), as well as distribution-specific details such as the precise path of the bootloader configuration file in the "/boot" file system.

The project TODO[18] file records an intent to create a library interface but no progress is apparent at the time of writing.

The new-kernel-pkg script provides an additional layer over the grubby command which adds the ability to create or remove an initramfs image and to run RPM post-transaction hooks stored in "/etc/kernel/postinst.d".

2.3.2 perl-Bootloader

The SUSE perl-Bootloader packages and update-bootloader script provide very similar functionality to the RED HAT grubby tool: a set of perl modules are provided that can read and write the configuration formats of various bootloaders (lilo, elilo, grub1, grub2, grub2efi), as well as a script to perform the task of adding, removing and updating entries for a particular kernel.

Unlike grubby, the perl-Bootloader modules treat grub2 used in EFI configurations as a separate configuration family but this is a relatively minor implementation detail.

Like new-kernel-pkg and grubby, update-bootloader supports templating of new entries based on values used for previous configurations. Unlike the Red Hat package update script, the update-bootloader tool does not support generating a new initial ramfs image and requires the path to a previously generated image to be provided.

2.4 BootLoader Specification

The *BootLoader Specification[19](BLS)* is an attempt by the FREEDESKTOP.ORG[20] project to define a common standard for a shared, system-wide "/boot" file system.

The specification discusses the motivation for better standardisation of boot entry configuration, and the limitations of both existing open solutions and the EFI standards.

The specification documents requirements for the "/boot" file system and its directory structure, as well as a format for drop-in configuration *snippets* that each correspond to a single bootable instance of an operating system.

Compliant implementations create and delete files in the drop-in directory in order to add and remove bootable menu entries from the configuration. This avoids the need to edit entries embedded in a larger configuration file and the complexities and uncertainties that this introduces (a considerable part of the complexity of both grubby and perl-Bootloader is the need to accommodate both manual and machine modifications to the configuration as well as ambiguous or variable aspects of each boot loader's native configuration syntax).

The current version of the BLS has the following requirements:

- Boot file system located at $BOOT

  - $BOOT place holder resolved at installation time to a location meeting the requirements of the specification. Specific requirements depend on the type of firmware (BIOS vs. EFI) and disk label (MBR vs. GPT) in use, and whether or not a usable EFI System Partition (ESP) already exists.

- $BOOT/loader
  - Root directory for all BLS configuration.
- $BOOT/loader/entries
  - Drop-in directory for BLS configuration snippets.

The BLS further specifies that the $BOOT file system must be a VFAT 16 or 32 file system. This requirement appears to be an attempt to harmonise with aspects of the (U)EFI specification, particularly the requirements of the ESP. At this time there are no known implementations of BLS that enforce this requirement.

Currently, BLS also requires that the partition containing the $BOOT file system have partition type code 0xEA on systems using the MBR partition scheme. This is problematic since this value is not reserved for this use and is currently in use by at least one other partitioning tool[21] for a conflicting purpose. Again: no known implementation of BLS enforces this requirement.

An implementation of a subset of the published BootLoader Specification is present in the grub2 packages shipped in recent Red Hat Enterprise Linux and Fedora releases. The implementation does not extend to the full set of restrictions on the $BOOT volume as described in the standard (which would need support from the installer and other components), but does provide a usable mechanism that may be harnessed to create additional boot entries on these systems.

Although the specification was written some years ago, and is implemented in several Red Hat distributions, development appears to have stalled. Private conversations with one of the contributors to the specification indicate that although there was general agreement over the direction the standard should take that insufficient motivation existed at the time to further its development and adoption.

The standard is appealing for the current work in part because it directly aims to solve the problem of a shared, system-wide "/boot", and to solve the difficulties in configuration and file conflicts that arise.

These problems are a direct superset of the problems facing a snapshot boot implementation: where the operating system instances considered by the BLS are independent distributions (or even operating systems), the instances considered in the case of snapshot boot are simply multiple, divergent copies of one operating system instance.

It is this convergence of problem and requirement that makes the BLS solution readily adaptable to snapshot boot needs: an early prototype of the boot manager data store was developed prior to the authors adopting the BLS and considerable agreement existed between the two, in terms of the information present, data layout and directory structure.

For this reason, it is proposed that the immediate priority for the Boot Manager be to generate BLS compliant boot configuration on systems where a compliant boot loader is available, and that the formats used for this information by the Boot Manager and BLS be treated as the authoritative store of this data. For platforms where no bootloader that implements BLS is available (for example, IBM z systems), an extension may be created to render Boot Manager or BLS data into the native configuration syntax.

The imgbased project[25] currently generates BLS boot entries for managed snapshot volumes.

2.4.1 BLS configuration snippets

A configuration snippet is a UNIX-formatted plain text file in UTF-8 encoding containing key/value pairs that define the values necessary to boot the entry. The following keys are currently defined:

title A human-readable title string for this entry ("Fedora 24 (Workstation Edition)").

version A human-readable version string for this entry ("4.10.8-100.fc24.x86_64").

machine-id contains the machine ID of the installation ("611f38f...").

linux the bootable kernel image to be loaded ("vmlinuz-4.10.8-100.fc24.x86_64").

initrd the initial ram file system image to be loaded ("initramfs-4.10.8-100.fc24.x86_64.img").

options the kernel command line arguments to be passed ("root=/dev/mapper/vg00-lvol0 rd.lvm.lv=vg00/lvol0").

efi an arbitrary EFI program to be loaded (a path relative to $BOOT) ("/System/Library/CoreServices/boot.efi")

devicetree a binary device tree archive to be loaded on platforms that use DTB files ("tegra20-paz00.dtb").

Each configuration snippet must contain at least one linux key or one efi key, and must be stored in a separate file in "$BOOT/loader/entries", named according to the scheme "<machine-id>-<uname -r>.conf".

For example:

```
6a9857a393724b7a981ebb5b8495b9ea-3.8.0-2.fc19.x86_64.conf
064d6dfabdea4552b3483779f63b656e-4.8.6-300.fc25.x86_64.conf
064d6dfabdea4552b3483779f63b656e-4.10.11-200.fc25.x86_64.conf
```

A BLS snippet for a Fedora 25 installation using kernel 4.10.11-200.fc25.x86_64 may contain:

```
title Fedora 25 (Workstation Edition)
version 4.10.11-200.fc25.x86_64
machine-id 064d6dfabdea4552b3483779f63b656e
options ro root=/dev/mapper/vg_f25-root rd.lvm.lv=vg_f25/root LANG=en_GB.UTF-8
linux /064d6dfabdea4552b3483779f63b656e/4.10.11-200.fc25.x86_64/vmlinuz
initrd /064d6dfabdea4552b3483779f63b656e/4.10.11-200.fc25.x86_64/initramfs
```

2.4.2 Proposed BLS enhancements (i)

Matthew Garrett has proposed a number of enhancements intended to address shortcomings in the BLS[26]. These include the following changes:

- Introduction of a formal name space in $BOOT ("$BOOT/org/freedesktop/.../")
- Support for chain loading entries via chainload key.
- Support for the Multiboot[22] standard via multiboot and module keys.
- Changes required MBR partition ID to 0xEF (EFI FAT-12/16/32).
- No requirement that the partition be VFAT.
- Support for kernel and ramfs images on other partitions.
- Remove the requirement that kernels be valid UEFI executables on UEFI systems.

These proposals have not yet been incorporated into the BLS. With the exception of the name space change (which although superficially reasonable fails to fully address the problem it seeks to solve) these are all improvements to the specification in terms of its suitability for use in booting snapshot images.

2.4.3 Proposed BLS enhancements (ii)

We propose two minor extensions to the BootLoader Specification that will enhance the facility for the use of a snapshot boot implementation:

1. Addition of a simple, free text tagging mechanism to support grouping, filtering and sorting of entries.
2. Amendment of the suggested configuration file naming recommendation to include a free text discriminator field to allow snapshot entries to be given a unique name.

These changes have not been formally proposed at this time. It is not clear whether the best route to having these changes included is to amend the published BootLoader Specification, or to first attempt to implement these features in the FEDORA grub2 package.

2.5 Alternate proposals

2.5.1 Directly booting LVM2 volumes and supporting "/boot" on LVM2 devices

It has been suggested by the LVM development team that the problem of booting directly to snapshot volumes (as well as other complex volume types) be addressed by enhancing the native grub2 support for booting LVM2 volumes and allowing both the target root file system, and the "/boot" file system to be hosted on LVM2 devices (or obviating the need for a separate "/boot" entirely, and allowing it to become a directory within the root file system).

Boot entries for snapshot volumes would then be identified either via the existing "grub2-mkconfig" discovery framework[4], or by enhancing grub2 to directly detect and enumerate operating system instances located in LVM2 volumes.

Enhancing grub2 to directly comprehend LVM2 metadata, and to be able to place the "/boot" directory on LVM2 devices are existing long-term goals. Previous efforts in this area (by the grub2 developers) have been lacking in terms of safety guarantees and support for a comprehensive set of LVM2 features:

- Grub2 LVM2 runtime is developed independently of upstream LVM2
  - Duplicates internal constants (e.g. "GRUB_LVM_LABEL_SCAN_SECTORS"), assumptions, and implementation, requiring manual synchronisation
  - Independently implements and enforces metadata and device checks (e.g. whether a pvmove is in progress).
- Limited set of LVM2 features and targets supported.
  - No support for thinly provisioned volumes.
  - No support for cache volumes.
  - LVM2 RAID support may be out-of-sync with upstream due to separate development.
- Bugs in Grub2 LVM2 support may cause boot failures, or the bootloader, or kernel to crash, and in some cases could result in data corruption of LVM2 volumes or the devices that contain them.

The LVM development team has proposed the creation of a module, using native LVM2 code for discovery and metadata handling as a solution to these problems and to address the problem of grub2 and LVM2 device support becoming out-of-sync.

This is a viable long-term strategy but significant obstacles exist to realising it in the short term:

1. LVM2 user space run time must be ported to the grub2 run time

[4] Using the DEBIAN "os-prober"[23] tool that is currently used to automatically detect Linux operating system instances installed to system devices

(a) Requires creation of a compatibility layer to allow LVM2 library code to be compiled and linked into both the usual ELF DSO used by applications on the running system, as well as and appropriate module for the `grub2` run time.

(b) The `grub2` environment is necessarily heavily restricted:

i. No standard I/O

ii. Limited memory management model

iii. No process management, no `ioctl` services

iv. Non-standard terminal interaction

2. It is unclear how the build for the combined solution would work. Currently the modules linked into grub2 (either into the core image, or as separate loadable modules), are built from a common source tree. Since the proposed module would be developed and maintained in the LVM2 source tree from the existing user space code, a mechanism would need to be found to either generate source code for import to the `grub2` tree, or to make the necessary build artefacts available from an LVM2 build to be used in order to build `grub2`.

These challenges can be addressed but involve considerable time-consuming development work and coordination with the maintainers of the `Grub2` environment.

There is currently limited experience in developing and debugging software for the `grub2` early boot environment in the LVM development team and many routine debugging aids (such as `valgrind`[24]) are not available in this context.

2.5.2 Adopting an existing bootloader abstraction

An alternative to using the *BootLoader specification* is to adopt one of the existing bootloader abstractions discussed in 2.3 on page 11.

In a similar manner as is proposed for the *Boot Manager* using BLS, these tools allow the creation, removal, and configuration of bootloader entries for a given tuple of boot values.

Although this would allow building on existing solutions, rather than developing a new *Boot Manager* component from scratch, this option was not further considered at this time for the following reasons:

- Chosing an abstraction commits the solution to a single distribution, or family of distributions
  - Otherwise, an additional abstraction layer would need to be developed to allow the solution to target multiple different bootloader management back-ends.
- The bootloader abstractions today represent the historical state of boot loader support for Linux: multiple divergent implementations and formats with some common aspects. Persisting with this model when virtually all supported architectures have now moved to a single, common bootloader implementation ties the project to unhelpful and unnecessary historical detail.

3 Existing solutions

Other operating systems and storage stacks (notably Solaris/`ZFS`, `Linux`/`BTRFS`, and Linux distributions shipping the `ZFS` file system) give the administrator the ability to boot a machine into a selected snapshot of the root file system (and in some cases to substitute other file system mounts with snapshot versions).

This section focuses on the implementations of most immediate relevance: Oracle Solaris/`ZFS`, SuSE's Snapper using the `BTRFS` driver, as well as the proposed Stratis Storage project.

3.1 Snapper

The SNAPPER[5] tool from SUSE includes both a command-line interface and a graphical application for creating, viewing and managing snapshots.

The tool also includes a mechanism to boot into a selected snapshot of a BTRFS file system. This feature relies on a BTRFS-specific superblock feature: the *default subvolumeid* field. By overwriting this value the tool controls the BTRFS subvolume that will be visible by default for any subsequent mount, including at system boot time.

This method avoids the need to modify the `fstab`, kernel command line parameters, or the `initramfs`, but has the disadvantage that recovery is made awkward in the case that the current default is for any reason unbootable. This behaviour is also considered objectionable by some users[6][7] since it assumes control of a file system feature normally available for the administrator's use.

There are also concerns with the fact that Snapper currently stores its snapshot meta data in a subdirectory within the volume that is being managed. This causes problems during recovery in some cases - a common workaround is to relocate the directory elsewhere and replace the original with a symbolic link[7].

- + Simple graphical user interface
- + Ability to 'diff' and recover specific files (BTRFS only)
- + Built-in snapshot expiration and clean-up (multiple policies)
- - Advanced features tied to btrfs
- - Requires administrator to give default subvolume control to Snapper
- - Limited ability to create varied menu options and to control boot process at boot time
- - Lacks ability to mount snapshots of non-root file systems at boot time.

3.2 Solaris ZFS and `beadm`

SOLARIS gained the ability to take snapshots of the root file system with the introduction of ZFS in SOLARIS 10. Since that time new management tools have been introduced to simplify the maintenance and use of multiple bootable configurations on a single system.

SOLARIS introduces the notion of a "*boot environment*": a bootable instance of the OS that encapsulates the file systems (or snapshots) to be made available as well as the kernel and ram disk images to be used.

An initial boot environment is created at installation time and further environments can be created by the administrator with the "`beadm`"[8] command. The command also provides the ability to list, modify and destroy environments and to change the active (default) boot environment.

The boot environment mechanism is strongly coupled to ZFS and uses ZFS pools, file system volumes, and snapshots to provide the file system mount points required by a boot environment.

An instance of a particular mountpoint (the origin volume, or a clone) is referred to as a *dataset*: each boot environment comprises exactly one *root dataset*, and one or more *additional datasets* (auxiliary volumes to be mounted below the root mountpoint). Datasets are associated with a given boot environment using the `beadm` tool. This feature allows a coherent set of snapshots to be combined into a single bootable environment.

The `beadm` command is also responsible for managing and updating bootloader menu entries for the managed boot environments, and for calling "`bootadm`"[9] to update a boot archive (equivalent to the use of an `initramfs` image for Linux systems). On x86 systems SOLARIS uses a modified build of the `Grub1` bootloader.

Menu entries are generated automatically using the assigned "`name@description`" labels accepted by "`beadm create`"; labels are a free-text field which may include time and date strings, descriptions etc. to uniquely and meaningfully identify a given instance.

SOLARIS distinguishes between a *"snapshot"* of a boot environment, and a "*clone*": a *snapshot* is a read-only point-in-time image of a volume that is not bootable. To create a *clone*, that supports writes, and that can be used to boot the system, a previously created snapshot is used to create a new boot environment[10].

ZFS licensing precludes its inclusion in most distributions[11].

- + Closely integrated with file system and overall system administration suite.
- + Capability to associate auxiliary data volumes with boot configuration.
- + Simple interface to create, destroy, list, and modify boot environments.
- - Reliant upon SOLARIS' `bootadm` and `Grub1` integration.
- - Directly tied to `ZFS` concepts and features.

3.3 Stratis Storage

The Stratis Storage project aims to provide `ZFS`/BTRFS-style volume management and file system features by integrating layers of existing technology: Linux's `device-mapper` subsystem, and the `XFS` file system. The project is relatively new but is under very active development and may represent a future snapshot management technology alongside the existing LVM2 and BTRFS stacks.

The Stratis Storage design document proposes the addition of boot capabilities in future versions[2] but this work is planned for later releases (Section 12.6 on page 24 proposes boot support for `Stratis` 4.0).

The Boot Manager may be extended to support `Stratis` file systems when it becomes possible to use these volumes as the system root file system.

4 Requirements

4.1 Boot Manager requirements

4.1.1 Ability to create BLS compliant boot loader entries

The tool must be able to accept a set of boot tuple values on the command line or via the library interface, and to create a corresponding BLS compliant boot loader entry ("BLS entry") capable of booting the system to that configuration.

On platforms that require the use of a *device-tree* the tool must accept and propagate this key in addition to the basic boot tuple values.

4.1.2 Ability to remove BLS boot loader entries

The tool must be able to remove a currently configured BLS entry when given appropriate selection criteria via the command line or library interface.

Possible criteria include the entry *Title*, *root*= value, or the unique per-entry identifier generated at the time an entry is created.

Where the given criteria correspond to more than one entry the tool will remove all matching entries (for example, to remove all entries associated with a specific kernel, or snapshot volume). Deletion of multiple entries simultaneously may require confirmation via a prompt or command line parameter.

4.1.3 Ability to list the available BLS boot loader entries and their properties

It must be possible to display all available BLS entries on the system, and to list relevant properties for them. It is desirable for the listing to also include a filtering mechanism, allowing entries to be selected using similar criteria to those used for deletion and other operations.

4.1.4 Ability to display and modify the selected default boot entry

In order to provide comprehensive management of the snapshot boot facility the tool should be able to both report the current default boot entry, and to modify the default stored by the bootloader (for bootloaders that support persistent configuration).

4.2 File system substitution requirements

4.2.1 Accept file system substitutions on the kernel command line

The file system substitution mechanism must accept a specification on the kernel command line.

This maximises the flexibility of the feature, and allows experienced users to create or alter this configuration on-the-fly using the boot loader's native editing facilities.

4.2.2 Perform substitutions before general system initialisation

All file system substitutions must be fixed early on in the system boot process, in particular, before any services or other long-running processes that might otherwise use the substituted mount points have been started.

4.2.3 Support clean shut-down of substituted mount points

In addition to creating the required configuration during system boot, the file system substitution component must correctly reverse this configuration during shut-down, unmounting file systems and releasing the corresponding block devices at the appropriate points in the system shut-down sequence.

4.3 Snapshot Manager requirements

4.3.1 Allow creation and removal of snapshots using `LVM2`, `BTRFS` or `Stratis`.

The tool must allow the user to specify an existing device, logical volume name, or file system mount point to be used to create a new snapshot. The user should be able to specify a distinguishing label and other properties for the snapshot, and to override existing defaults and inherited settings where applicable.

The tool must allow an already existing snapshot to be removed by providing a unique reference to the snapshot (the snapshot name, or device name where applicable, or some other unique identifier either created by the underlying snapshot mechanism or generated by the Snapshot Manager).

4.3.2 Automatically create boot configuration for snapshots of the root file system

The Snapshot Manager must be able to automatically create an appropriate boot entry for new snapshots of the root file system; required values that are not specified by the user should be obtained from either system templates or existing configuration as appropriate.

When a snapshot is removed using the Snapshot Manager tool, the manager must remove any corresponding boot entries that were previously created.

4.3.3 Support expiry of snapshots by user specified policy

The Snapshot Manager should allow the configuration of a policy controlling snapshot expiry, either on a system wide basis, or a per-snapshot basis.

Possible expiry policies include:

- Snapshot count

  - Snapshots are expired when the number of snapshots exceeds a user-defined threshold. Snapshots are removed in least-recently-created order until the limit is reached.

- Snapshot age
  - Snapshots are expired when they reach a threshold age specified by the user. Snapshots are removed in least-recently-created order until the limit is reached.

- Preservation tags
  - Tags are applied to snapshots to indicate that they are to be preserved. The presence of a preservation flag exempts a snapshot from any other expiry policy.

- Snapshot space usage
  - Snapshots are expired when a threshold of space consumption in the snapshot pool or host file system is exceeded. Snapshot are removed in least-recently-created order until the limit is reached.

4.3.4 Simultaneous creation of snapshots of multiple volumes

It should be possible for a user to specify multiple volumes (one of which may be the root file system) for which snapshots will be created simultaneously.

When creating a boot configuration for a multi-volume snapshot, the tool must be able to automatically create an appropriate file system substitution configuration, and to add it to the boot configuration for the snapshot.

4.3.5 Snapshot file operations

It is desirable to be able to perform additional high level operations on snapshot volumes as proposed in 1.4.4 on page 9. This functionality requires that the Snapshot Manager be either able to mount snapshot volumes as needed, or otherwise be able to inspect content within the image contained in the snapshot (for example using the `libguestfs`[27] library). In the case that it is necessary to preserve superblock time stamps while performing mounts of snapshot volumes, the Snapshot Manager may choose to create new, temporary read-only or read-write snapshots in order to satisfy this requirement.

Part II
Solution overview

5 Introduction

Initial efforts will focus on the `x86_64` platform and current Red Hat Enterprise Linux 7 and Fedora releases, since these include support for BLS.

It is expected that additional architectures and distributions can be addressed with relatively little effort at a later time[5].

A number of steps can be taken to solve the problems discussed in this document at this time; these may be implemented in an iterative manner, building on the results of each successive unit of work (with work proceeding in parallel on independent components where possible).

1. Document use of `root=` and `rd.lvm.lv=` to boot from snapshot volumes

[5] In particular, the `grub2` bootloader is not currently used for IBM z Systems running Red Hat operating systems: it may be necessary to render Boot Manager generated BLS snippets into the native `zipl` configuration format.

   (a) Address dracut support for thin-provisioned snapshots[4].

2. Provide tooling to create and manage instances of bootable snapshots, and to integrate these with the existing bootloader menu system or boot-time display manager.

3. Enable additional auxiliary file system snapshots to be associated with a bootable environment, to be automatically mounted in place of the origin file systems at boot time.

4. Provide a higher level snapshot management tool that handles the low-level aspects of snapshot creation and removal, as well as the creation and removal of corresponding bootloader entries.

5. Provide features to enable the management of snapshot life cycles, expiring and removing snapshots over time according to administrator controlled policy.

6. Allow snapshots of multiple volumes to be taken simultaneously, permitting a coherent snapshot of system state to be created.

7. Allow familiar operations on files and file systems contained within snapshots to be carried out from the running system.

6 Boot Manager

The Boot Manager will be responsible for creating and deleting BLS snippets corresponding to bootable snapshot instances. It is the first new software deliverable for the proposed project, and forms the basis of much of the functionality to be delivered in the near term.

The tool is envisaged as a command line interface and library bindings to allow an administrator, or another software component to inspect and manage the set of BLS boot loader entries present on the system.

7 File System Substitution

The component responsible for file system substitution must accept a value on the kernel command line and perform the requested mount point substitutions during the boot process.

There are two points at which the opportunity to make these changes exists: (i) during early user space processing (initial ramfs), after the selection and mount of the root device, but before the pivot of the system root, and (ii) after the pivot, but before `systemd` reaches the `local-fs` target.

To substitute file system mounts during boot it is necessary to either modify the visible copy of "`/etc/fstab`", or to modify the system mount state after local file systems have been mounted. Modifying the `fstab` has the advantage that tools such as `mount(8)` which operate on this file will behave as expected.

7.1 Direct modification of `fstab`

It is possible to control the snapshot used by the system by directly modifying the system `fstab` before non-root file systems are mounted. While this is the simplest approach it has a number of disadvantages that make it unsuitable for the proposed work:

- Modifying file system content modifies the snapshot being booted
- The substitution needs to be manually torn down at system shut-down time.
  - A backup must be taken of the original content and restored during shut-down.

  - A crash, power failure, or other unexpected reboot would leave the system in an inconsistent state: booting into some default root file system with the snapshot auxiliary volumes from the previous boot. This has the potential to lead to confusion, data loss or corruption, and considerable user frustration.

- User changes made while inside the snapshot environment must also be backed up and restored or they will be lost when the environment is next initialised and the `fstab` rewritten.

7.2 Substituting `fstab` via bind mounts

One method for modifying the `fstab` (without needing to revert the changes during shut-down, making the technique safe in the event of a system crash or power failure) is to substitute a modified copy using a *bind mount*[28]. A bind mount remounts part of the file system hierarchy (including a path that resolves to a regular file) to another location in the name space.

For example:

```
# Bind '/proc' to '/tmp/proc'
# mkdir /tmp/proc
# mount --bint /proc/ /tmp/proc
# ls -d /proc/1
/tmp/proc/1
# umount /tmp/proc
```

Or using a file path as the bind target:

```
# Create a file named 'bar' and bind it to 'foo'
# echo bar > bar
# touch foo
# mount --bind bar foo
# cat foo
bar
```

This allows a copy of the system "`fstab`" to be taken in another location, to which substitutions are made according to a specification. The modified copy is then substituted for the original by binding to "`/etc/fstab`". The modified copy will then remain visible in that name space for the duration of the mount (or until another mount to the same path occurs, or the system is halted). This has the benefit that the mount is torn down automatically, even in the event of an unclean shut-down.

7.3 Substituting individual mounts via bind mounts

It is also possible to use the bind mount mechanism to substitute individual mounts, either before or after, they are mounted from the location specified in the `fstab`. This has a number of disadvantages for the current proposal:

- Binding substitutions after the system has reached `local-fs.target` would leave the original file systems mounted (possibly with some open file descriptors belonging to long-running processes).

- Bind mounting a file system requires that it is already mounted somewhere in the name space: this would create additional steps to first mount, and then re-mount the substituted file systems.

- Tools that read the `fstab` may misbehave or give confusing output if the file does not appear to match the current mount state.

8 Snapshot Manager

The snapshot manager provides a higher level mechanism for creating, removing, and managing snapshots and their associated boot configuration and meta data. As well as providing a simplified interface to administrators, and combining multiple steps into a single operation, the manager provides an abstraction over the various low-level snapshot mechanisms and creates an integration point which may be used to introduce further features and services.

Part III
Implementation

9 Software Components

The proposed implementation involves creating three new software components:

1. Boot Manager
   (a) A command line tool to manage snapshot boot entries
   (b) Fixes to dracut required to support thin snapshots
2. File system substitution mechanism
   (a) A software component to perform file system substitutions passed on the kernel command line.
   (b) Integration with the Boot Manager to allow passing of a substitution specifications to boot entries.
3. Snapshot Manager
   (a) A command line tool to create, remove, and manipulate snapshot volumes and associated boot configuration
   (b) Support for creation of snapshots of multiple volumes simultaneously.
   (c) Policies to manage expiry and deletion of snapshots, based on:
      i. Snapshot count
      ii. Snapshot age
      iii. Preservation tags
      iv. Space consumption
   (d) Support for common file operations on snapshot data.

Initial development will focus on (1)and (2).

10 Development languages

As each component of the current proposal is an entirely new software element there is considerable freedom in selecting the language used for the implementation.

Where some components must execute in restricted environments (early user space, pre-`basic.target`) there may be restrictions on both specific languages and the types of languages (e.g. compiled vs. interpreted) that can easily be supported.

For interpreted languages, PYTHON has the advantage of widespread adoption in RED HAT distributions, and a large and active development community. Alternatives include PERL and RUBY but these have a smaller systems programming footprint in today's distributions.

The Go and Rust languages[29] [30]have gained popularity in recent years for systems programming and infrastructure. They are high-level, compiled languages heavily influenced by C and related languages, with the addition of features such as garbage collection, memory safety, type inference (Rust) and built in concurrency primitives[31]. While the popularity of Rust and Go seems set to increase they are still relatively young languages, and questions remain over long-term maintenance and support. In addition there is limited expertise in either language in the device-mapper and LVM2 developer community.

While C imposes additional burdens on development it is well-proven and the basis for much of the software in the distribution. Since the components that must be written in a compiled language are expected to be relatively small it seems to be the natural choice for these parts.

10.1 Boot Manager

The Boot Manager does not need to execute during early user space, or before the `init` daemon has fully initialised the system. The tools are expected to be used from one or more operating system installations present on the system, or a similar rescue environment or live image, and will have full access to the normal libraries and runtime environments those operating systems provide.

Some initial prototyping work has been carried out in Python and this is a commonly used language for similar development projects in Red Hat Enterprise Linux and Fedora: considerable expertise exists among these communities that can be exploited if required.

10.2 File System Substitution

The component responsible for substituting file system mounts according to a specification passed on the command line must run during early phases of system boot: either during early user space (initial ramfs), or during the `init` phase of boot, but before the system has reached the `local-fs` or `basic` targets. This means that local file systems may not be mounted, and some libraries and runtime environments may not be available.

This suggests a need for this component to be either so simple that it is guaranteed to run in all of these contexts (a POSIX shell script), or to be written in a compiled language with strict control over library usage in order to produce a binary that can be executed during these phases of system initialisation.

10.3 Snapshot Manager

The Snapshot Manager is currently the most distant, and least certain aspect of the proposal: many decisions can be deferred until other aspects of the work are complete, including the choice of implementation language. The ideal model may involve a small set of core components written in a compiled language, with more peripheral elements (for example, a graphical user interface) written in other higher-level languages as appropriate.

11 Library Interfaces

Both the Boot Manager and Snapshot Manager should export functionality via clearly defined programmatic interfaces, in addition to the user-oriented command line interface.

The file system substitution mechanism only offers a single. simple interface with substitution specifications passed on the kernel command line. No need is anticipated for a means to re-configure the facility at runtime since it is expected to be configured once, at boot time, and to remain in place until system shut-down.

References

[1] Stratis Storage https://stratis-storage.github.io/

[2] Stratis Design https://stratis-storage.github.io/StratisSoftwareDesign.pdf

[3] Dracut https://dracut.wiki.kernel.org/index.php/Main_Page

[4] Dracut PR#223 https://github.com/dracutdevs/dracut/pull/223

[5] Snapper http://snapper.io/

[6] Snapper Issue#178 https://github.com/openSUSE/snapper/issues/178

[7] Snapper layout discussion https://bbs.archlinux.org/viewtopic.php?id=194491

[8] beadm documentation https://docs.oracle.com/cd/E23824_01/html/E21801/index.html#scrolltoc

[9] bootadm documentation https://docs.oracle.com/cd/E26505_01/html/E29492/gglaj.html

[10] Creating and copying snapshots https://docs.oracle.com/cd/E23824_01/html/E21801/snapshot.html#scrolltoc

[11] ZFS license compatibility https://sfconservancy.org/blog/2016/feb/25/zfs-and-linux/

[12] new-kernel-pkg(8) https://linux.die.net/man/8/new-kernel-pkg

[13] update-bootloader(1) http://www.polarhome.com/service/man/?qf=update-bootloader&af=0&sf=0&of=0penSuSE&tf=2

[14] Plymouth https://www.freedesktop.org/wiki/Software/Plymouth/

[15] plymouth(8) https://linux.die.net/man/8/plymouth

[16] Grubby https://github.com/rhinstaller/grubby

[17] grubby(8) https://linux.die.net/man/8/grubby

[18] grubby-TODO https://github.com/rhinstaller/grubby/blob/master/TODO

[19] BootLoaderSpec https://www.freedesktop.org/wiki/Specifications/BootLoaderSpec/

[20] Freedesktop.org https://www.freedesktop.org/wiki/

[21] Rufus https://github.com/pbatard/rufus

[22] multiboot https://www.gnu.org/software/grub/manual/multiboot/multiboot.html

[23] Debian os-prober https://joeyh.name/code/os-prober/

[24] Valgrind instrumentation framework http://valgrind.org/

[25] imgbased_https://github.com/fabiand/imgbased

[26] BootLoaderSpec-mjg59 https://www.freedesktop.org/wiki/MatthewGarrett/BootLoaderSpec/

[27] libguestfs http://libguestfs.org/guestfs.3.html

[28] Bind and shared subtree support https://github.com/torvalds/linux/blob/master/Documentation/filesystems/sharedsubtree.txt

[29] Go Language https://golang.org/

[30] Rust Language https://www.rust-lang.org/en-US/

[31] Go Description https://en.wikipedia.org/wiki/Go_(programming_language)