Scratch Zone Packaging Design

James Carlson
Sun Microsystems, Inc.

version 1.4 2005/08/19

Abstract

The Solaris software packaging tools use running zones for security isolation when installing and removing packages in non-global zones. Zones configured in an alternate boot environment, though, cannot be booted or run normally. Thus, there's no way to add or remove packages from an alternate root and, since upgrade depends on use of an alternate root, no way to upgrade the system. This project provides a mechanism, dubbed a "Scratch Zone," that packaging tools can use to access zones within alternate root environments.

1 Background

After verifying user options, the packaging tools (/usr/sbin/pkgadd and /usr/sbin/pkgrm) do the actual system changes by execing `pkginstall` and `pkgremove`, respectively, from /usr/sadm/install/bin.

When used on a system with non-global zones, these internal tools must be invoked inside the zone that is to be modified (using the undocumented zone_enter system function). This is because the tools rely on data that can be modified by the zone administrator, such as `preremove` and `postremove` scripts. There must be no way for a malicious zone administrator to plant traps in the zone that can harm anything outside of the zone, which can happen if the scripts are run in the global zone.

To make this work, the packaging tools check the status of each zone in turn, and automatically invoke "zoneadm boot -s" on any zones that

aren't currently running, enter the zone to run the internal tools, and then "zoneadm halt" when done.

The packaging tools also support using an alternate root directory. This support is used to implement Live Upgrade, which mounts an alternate boot environment in a temporary directory and then directs the packaging tools to execute commands against that alternate root.

During a regular (not "live") upgrade operation, a miniroot environment is booted, which provides a read-only root file system. The system's real root directory is mounted on `/a`, and the packaging tools are directed to use "`/a`" as the alternate root.

In both types of upgrade operation, non-global zones cannot be booted. This is true because there's no guarantee that the zone configuration itself can be used on the current system (e.g., devices may be missing), the binaries in the zone cannot necessarily run on the current system (e.g., a whole-root zone with its own `libc.so.1` will fail), and the configured network interfaces will conflict with (have the same configured addresses as) the running zones on the system.

Because they can't be booted, and they must be booted to install or remove packages, upgrade of an alternate root environment cannot be handled.

2 Related Projects

- The Ashanti (interim Zones Upgrade solution) and Zulu (permanent Zones Upgrade solution) both depend on the features delivered by this design.

- The Rampart (Trusted Solaris 10) project is also making changes to the configuration and operation of non-global zones. The relationship between these two is described in 8.2 on page 32.

3 Existing Status and Problems

There are many different problems with the current design, as well as constraints on what possible solutions may do.

- The Zones toolset (libraries and commands) has hard-coded references to the root file system, and thus cannot access configuration inside an alternate boot environment.

- The non-global zones in an alternate root environment cannot be booted on the running system. Besides the likely clash with any currently-running zones, these zones may be based on a different kernel. While so-called "sparse-root" zones might work (the loopback "IPD"[1] mounts could be pointed to the running system), whole-root zones would not.

- For similar reasons, the zones that might be present on the system during a regular upgrade cannot be booted, nor (if somehow booted) can commands be run inside them.

- Mount points and devices may differ between the alternate boot environment to be upgraded and the running system.

- The packaging tools need access to executables and libraries that are consistent at all times with the running kernel. In a whole-root zone located in an alternate boot environment, there are no executables or libraries that are necessarily consistent with the running kernel.

- It must not be possible, under any circumstances, for an access to any file system object within an alternate boot environment to lead to read-write permissions on anything normally visible within the running environment. It should not be possible to do the same and get read-write access to storage within any other boot environment. This guarantee makes it possible to run Live Upgrade without affecting the running system.

- Live Upgrade currently does not mount shared file systems when presenting the alternate boot environment to the packaging tools. This has an unfortunate side-effect. If packages to be upgraded have bits that were delivered via a shared file system (an error, to be sure, but not a completely unusual one), packaging will attempt to update the bits anyway, leading to a "lost update."

[1] Inherit Package Directory: refers to a read-only loopback mount of a global zone directory.

For that last point, consider a shared file system mounted on "/foo" and a package that delivers "/foo/bar." /foo will not be mounted by the lumount command, leaving the mount point itself as an ordinary directory in the root file system. When pkgadd upgrades this package, it will find that /foo is a writable directory, and will create /foo/bar. When the system is switched over to this alternate boot environment, the shared file system will be mounted on /foo, obscuring the updated /foo/bar, and the old version of /foo/bar will be visible. When encountering a shared file or directory, the package operations should just fail instead.

This issue isn't specifically related to Zones, but becomes more acute with them due to the varying ways non-global zones can be configured to share resources. Thus, a global solution to the underlying problem is desired.

4 Architectural Evolution

I prepared an initial prototype based on just the Admin/Install gate tools and used it to flesh out the problems. This prototype added a "scratchzone" utility to /etc/lib/lu that could mount or unmount a zone in an alternate root environment.

To do so, it used chroot(2) to force libzonecfg to read configuration data from the alternate root (and some well-known fchdir(2) subterfuge to escape back). Then, it mounted up the zone's file systems and used zone_create with a unique alternate name to establish the zone in the kernel. The packaging tools were updated to use this "scratch" zone when the "-R" command specified an alternate root environment.

The problems solved with this prototype included:

- The system configuration or zone configuration (such as mounted file systems) may differ between the running system and the alternate boot environment. To deal with this, a new "-a" flag was added to lumount to cause it to mount all file systems within the alternate environment. Those that have "special" (see mount(1M)) nodes that are already used by other mounts (and therefore presumably shared with the current environment) are mounted as read-only loopback (lofs) mounts.

- The kernel deals with zones by name, but there could be a zone on the running system with the same name as the scratch zone being mounted. To deal with this, the mount command automatically tried "SUNWlu-NAME" first. If that were already present (either due to multiple boot environments being used at once or due to very long zone names aliasing together after truncation), then an arbitrary name of the form "SUNWlu.NUMBER" was chosen. (Note the "-" and "." usage in these two cases.) A separate text file held a translation table between these kernel zone names and the actual boot environment and user-level zone name.

  Note that the use of the "SUNW" prefix prevents accidental access to these scratch zones by Zones-related utilities. This is by design and is a helpful bit of foresight from the original Zones team.

- The key distinction for determining whether any mount point inside a zone is writable is not just whether the 'special' field points to a writable directory, but rather whether it is distinct from all other mounts on the system. In other words, what needs to be checked is whether the contents are "shared" or "unshared."

  For non-lofs mounts, it is shared if the contents of the special field matches any other currently mounted file system.

  For lofs (including IPD) mounts, I developed several simple rules:

  1. If the special field points to a directory that is either a subdirectory or the parent of any other lofs mount on the system, or if it has the same device/inode as the special field on some other mount, then it's shared.
  2. When mounting lofs entries into a scratch zone, the special node must be resolved as far up the tree as possible, for two reasons: the lower levels may be read-only and the top level forms a canonical name for conflict detection. This resolution is done by scanning for lofs mount points that form initial portions of the string. If one is found, then the special node for that mount is substituted and the process repeated. This treats lofs mounts as a hierarchical tree and finds the root node.
  3. If the root of the zone itself is shared, then the zone cannot be mounted. (This one rule may be relaxed if sharing root but

keeping essential portions of /etc and /var unshared makes sense.)

As an example of all of this in action, consider a simple system with just root (/) and /export. The latter is shared with a boot environment called "test." Mounting this alternate boot environment up with "lumount -a test" results in these mounts:

```
/.alt.test read-write on /dev/dsk/c0t0d0s4
/.alt.test/export read-only on /export
```

Now suppose we have a zone called "myzone" in this boot environment, and its root directory is /export/myzone. Mounting that up with "scratchzone mount /.alt.test myzone" results in reading these files to fetch the zone configuration:

```
  /.alt.test/etc/zones/index
  /.alt.test/etc/zones/myzone.xml
```

That configuration provides us with boot-environment relative information. We prepend "/.alt.test" to the zone's specified root directory. This is where the lofs magic occurs. We now have "/.alt.test/export/myzone" and we find that "/.alt.test/export" is a lofs mount point on the system, and substitute in its special node, which is "/export." This gives us the base directory of "/export/myzone". Note that if the boot environment had a different mount point for /export, this would have resolved differently, so the result is not the same as just using the raw Zones data without prepending the boot environment root. (Though the result in this particular case happens to be the same.)

Thus, assuming /export/myzone itself isn't in use by a running zone (either the configured zone path in the alternate boot environment is different, or the zone just isn't running), we mount up the zone like this (in part; many items omitted for clarity):

```
/export/myzone/root/lib read-only on /.alt.test/lib
/export/myzone/root/usr read-only on /.alt.test/usr
/export/myzone/root/sbin read-only on /.alt.test/sbin
/export/myzone/root/tmp read-write on swap
```

This left several major issues unresolved, though, and I scrapped the prototype (as should happen to all good prototypes). The most important lingering issue was that the mount logic in "scratchzone" became quite complex, depended on many ON-private interfaces, and essentially mirrored exactly what "zoneadmd" already does when booting a zone. Thus, as an architectural matter, it made more sense to deliver this functionality via an ON patch (additional zoneadmd functionality) rather than saddle the Admin/Install gate with hard-to-manage dependencies.

The next most serious issue (and likely fatal flaw) was that the binaries that pkgadd/pkgrm would be invoking inside that mounted zone would be the very ones that were undergoing upgrade. In other words, it provided no access to executables and libraries consistent with the running system.

The second prototype, which consists of a set of patches to libzonecfg, zoneadm, zonecfg, zoneadmd, zonename, and zlogin in ON, and which forms the basis of the proposed design, uncovers and addresses the following issues:

1. The running system's binaries need to be accessible inside the scratch zone. This is remarkably similar to the issue faced by a standard Solaris upgrade: the binaries on the disk can't be run during the upgrade process, so we ship a "miniroot" that contains the necessary utilities.

   This problem is solved by adopting the same strategy for the scratch zones. Inside the scratch zone, the usual directories (/lib /sbin /usr) are loopback mounted from the running system, and the actual zone root to be upgraded is mounted on /a within the zone. Using /a in this case and patterning after standard upgrade minimizes risk because we know that a standard, uncomplicated upgrade itself must work. The simple way to create this environment is to mount tmpfs on a new directory for the zone root, and then mkdir the necessary mount points inside this scratch workspace.

2. Unfortunately, /etc is a bit of a problem. It cannot be loopback mounted from the global zone of the running system (as the other famous mounts are) because there are secrets present (for example, /etc/inet/secret/ and /etc/ssh/*_key). It cannot be mounted from the zone's own /etc in the alternate boot environment because there

are executables present; notably in /etc/lib/lu and /etc/fs. Finally, it can't easily be mounted from any other location because we don't necessarily have (or otherwise need) a pristine copy of /etc as is included in the miniroot.

To fix this, we'll need multiple strategies. Some clean-up is possible and arguably necessary. For /etc/lib/lu/, we can move these binaries to /lib/lu/ (as has already been done for all the other old denizens of /etc/lib/) and leave behind a symlink. Bugs should be filed to cover moving /etc/fs/ into /lib/fs/ (to mirror the existing /usr/lib/fs/) and /etc/security/lib/ to /lib/security/ and /etc/lp/alerts/ to some other planet.

However, that doesn't entirely fix the problem. The un-upgraded zone in the alternate boot environment won't have these fixes in it, and thus there's a chance that mounting the zone's /etc (the most logical choice) will still lead to accidents. To avoid this, we will check if any of these are directories (as opposed to symlinks or just missing) in the zone's /etc:

```
/etc/lib/lu
/etc/fs
/etc/security/lib
/etc/lp/alerts
```

If so, then we loopback mount (read-only) the running system's copies of these directories atop those locations.

3. The naming system used by the Zones console subsystem implementation is problematic. In order to establish an unambiguous rendezvous point, it uses the zone name in the device-tree configuration of the console device and the UNIX-domain socket. This could conceivably be adapted to use the kernel zone name, but the changes are substantial and there appears to be no reason to do this, as the console shouldn't be needed to make upgrade work. Thus, this is deferred.

4. Mounting up a zone is, as noted before, mostly similar to what is done when booting a zone. Thus, the same logic is used within zoneadmd. The changes necessary are:

- Create the $ZONEPATH/lu directory for the zone's root and mount tmpfs on it.
- Create a temporary name for the zone, call zone_create, and record the zone:kernel-zone:boot-environment mapping in the /tmp/.alt.lu-zone-map file.
- Create all of the following directories inside the tmpfs root:

  ```
  /system /system/contract /proc /dev /tmp /a
  /etc /var /etc/lib /etc/fs /lib /sbin
  /platform /usr
  ```

  and symlink bin to ./usr/bin.
- Skip over resource control setup, devfsadm registration, and network interface configuration.
- Mark the kernel zone as "ready" so that zone_enter from the packaging commands will succeed. The zone_boot function is not called, so "init" is not run in the zone.
- Fix libzonecfg so that this new state ("mounted") is returned to user applications that deal with zone state. The new state, ZONE_STATE_MOUNTED, is detected when the kernel state is ZONE_IS_READY, but the root path for the zone ends in "/lu" rather than "/root."

  This was initially implemented using just state "ready" in the prototype, but that proved to be problematic when extending to `zlogin` and other utilities.

It would be possible to create the "$ZONEPATH/lu" directory as part of normal zone installation or as part of lucreate or similar operations. However, there's no guarantee that the user will necessarily perform any of those actions after installing the new tools. In fact, to operate with the miniroot environment used for the interim solution, the system must be prepared to perform an upgrade operation on an untouched S10 FCS system that lacks the "/lu" mount points.

For those reasons, it must be included in `zoneadmd`.

The existence of the directory should probably be mentioned in release notes, so that nervous administrators who inspect the implementation details of Zones aren't surprised by it.

5. `Lusync` (an internal implementation detail of Live Upgrade) reads the `synclist(4)` file and determines which files to copy or modify during the actual activation sequence. The administrator can add extra files to this list to solve synchronization issues in particularly important files, such as those used for application configuration.

   Unfortunately, the paths that the non-global administrator places in this file may be compromised. For instance, a malicious administrator might place a reference to a sequence of "../.." links anywhere in the path (or in any link traversed by the path specified) in order to trick a global-zone process into copying a sensitive global file (such as `/etc/shadow`) into the zone. Fixing this would mean cumbersome tests on each element to make sure that none were compromised to reach outside the zone, and yet somehow avoiding any timing-based attacks (such as rapidly renaming and symlinking directories during the `lusync` process).

   A simpler mechanism is to make sure that `lusync`, like the internal `pkginstall` and `pkgremove` tools, runs inside the zone. `Lusync` could enter the scratch zone and the real zone separately and somehow pipe the data between the two, but a better way to do this is to provide `lusync` with a means to access the contents of the same zone in the other boot environment directly through a mount point inside the zone.

   There are two cases to consider here. One occurs during activation. In this case, the work is done inside the scratch zone and access to the same zone on the running system is needed. To do this, we will introduce a new "/b" mount point inside the zone that "zoneadm mount" automatically establishes to point to the running zone, if present. If the same zone is not present on the running system, then "/b" will not exist at all, so "if [-d /b]; then" will do the right thing inside the scratch zone.

   The second case occurs after boot. In this case, we need to complete the synchronization operation before the zone begins running "for

real." To do this, we need access to the data in the old boot environment. This can be provided by having the post-reboot LU script (which runs before zones are booted) mount up both a scratch zone for the old BE and the current BE. Then it can explicitly lofs-mount the old BE's root into the current BE, and run `lusync` inside the zone. Once this is done, it can unmount the zones to clean up and allow normal boot to proceed.

6. In performing the `lusync`, we wish to be certain that the source and destination for the copy are exactly the same zone. The user, though, could conceivably perform `lucreate`, then delete and recreate the zone on the running system using `zoneadm` with a completely different configuration but the same name, and then attempt to run `luupgrade`. We will mistake those two zones as being the same, when they are not.

   To avoid this corner case, which would likely result in unrecoverable (and perhaps undetectable) corruption of the zone contents, we need a way to tell two zones with the same name apart. One way to do this is by including a "unique id" (UUID; from `libuuid`[2]) for each zone. This has a number of useful side effects for the future, including the ability to handle renamed zones in a transparent manner, and the ability to identify zones that have been "migrated" from one system to another.

   Thus, we will add a new UUID field to the /etc/zones/index file, and enhance the tools so that each zone is guaranteed to have a UUID when either a zone is created, zone configuration is changed, or an LU copy operation takes place. (It would be sufficient to perform this as a post-patch operation as part of preparing for LU operation.)

   This field will not be added to the DTD, as it's intended for use in searching and indexing non-global zones, not as a part of the configuration of the zone itself.

[2]UUIDs are 128-bit numbers. IPv6 automatic address configuration also needs unique "Interface ID" numbers, but these are 64-bit numbers. This design does not address the IPv6 needs because the structure of addresses is much more complex than just identification, and likely needs administrative involvement.

7. `lusync` is a script and executes among other scripts. To make entry into the scratch zones simple in this context, `zlogin` will be enhanced to support "-R" as a way to specify an alternate boot environment, and to allow entry into zones that are in state ZONE_STATE_MOUNTED.

This results in a complex set of mounts:

```
/export/myzone/lu read-write on swap
/export/myzone/lu/etc read-write
  on /export/myzone/root/etc
/export/myzone/lu/var read-write
  on /export/myzone/root/var
/export/myzone/lu/etc/lib read-only on /etc/lib
/export/myzone/lu/etc/fs read-only on /etc/fs
/export/myzone/lu/lib read-only on /lib
/export/myzone/lu/sbin read-only on /sbin
/export/myzone/lu/platform read-only on /platform
/export/myzone/lu/usr read-only on /usr
/export/myzone/lu/tmp read-write on swap
/export/myzone/lu/a read-write on /export/myzone/root
/export/myzone/lu/b read-only
  on /export/myzone-old/root
/export/myzone/lu/dev read-write on /export/myzone/dev
/export/myzone/lu/a/lib read-only on /.alt.test/lib
/export/myzone/lu/a/platform read-only
  on /.alt.test/platform
/export/myzone/lu/a/sbin read-only on /.alt.test/sbin
/export/myzone/lu/a/usr read-only on /.alt.test/usr
/export/myzone/lu/proc read-write on proc
/export/myzone/lu/system/contract read-write on ctfs
/export/myzone/lu/etc/svc/volatile read-write on swap
/export/myzone/lu/etc/mnttab read-write on mnttab
/export/myzone/lu/dev/fd read-write on fd
```

Note that /export/myzone/lu (the main tmpfs mount) is used as the root for the zone, the zone's root is mounted on /a inside the zone, the running zone's root is mounted on /b, and binaries from the alternate boot environment are intentionally mounted for the IPD entries inside /a.

See figure 1 on page 14 for a closer graphical look at the mounts inside of a scratch zone.

See also figure 2 on page 15 and figure 3 on page 16. In those diagrams, the dashed line shows the lofs resolution, and the different shaded areas show the various collections of mounts created (for lumount, IPDs, and scratch zone construction). The only distinction between the two is that /export/home (the file system containing the zone roots) is shared between the BEs in one case, and not in the other.

Along with this, the packaging tools are modified such that if "-R" is presented, that is used as the root of the alternate boot environment (as expected), but a fixed "-R /a" is passed to `pkginstall` and `pkgremove` when those are invoked inside the zone and the zone is in "mounted" state.

Marking the zone as just "mounted" (rather than booting it) prevents users from booting the zone or shutting it down while the packaging tools are modifying the zone contents.

5 Miniroot Usage

The same set of scratch zone tools are intended to be used inside the miniroot environment, used for both "standard" (as opposed to "Live") upgrade and for the interim patch-based solution.

In this case, the system's root file system is ordinarily mounted on /a and the booted system itself runs over NFS to the boot server. The scratch zone tools treat "/a" in the global zone as an alternate boot environment.

As with the new `lumount` "-a" option, the miniroot will need to be enhanced to mount up the other file systems under the /a tree as well, instead of just the root file system, so that any other mount points that contain the zone roots will be accessible.

5.1 File System Layout

The same procedure as for Live Upgrade alternate boot environments is followed, but since the sharing issues aren't present, the results are slightly different.

The process starts by finding the zone's path and prepending the alternate root location. Since /a won't be a lofs mount for a shared file system,

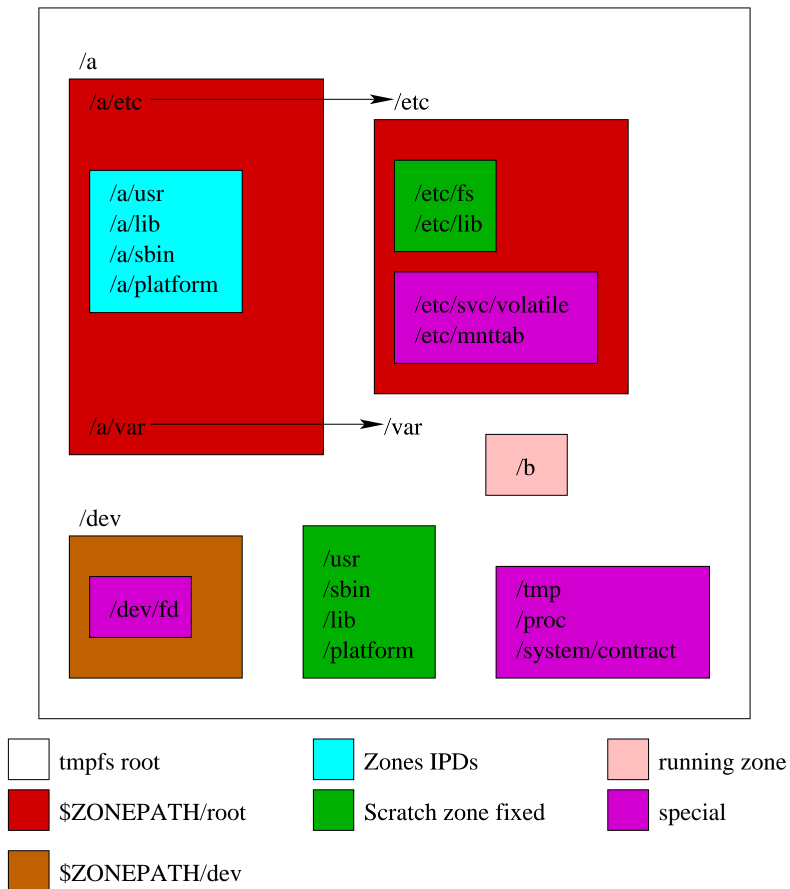

Figure 1: Mount points inside scratch zone

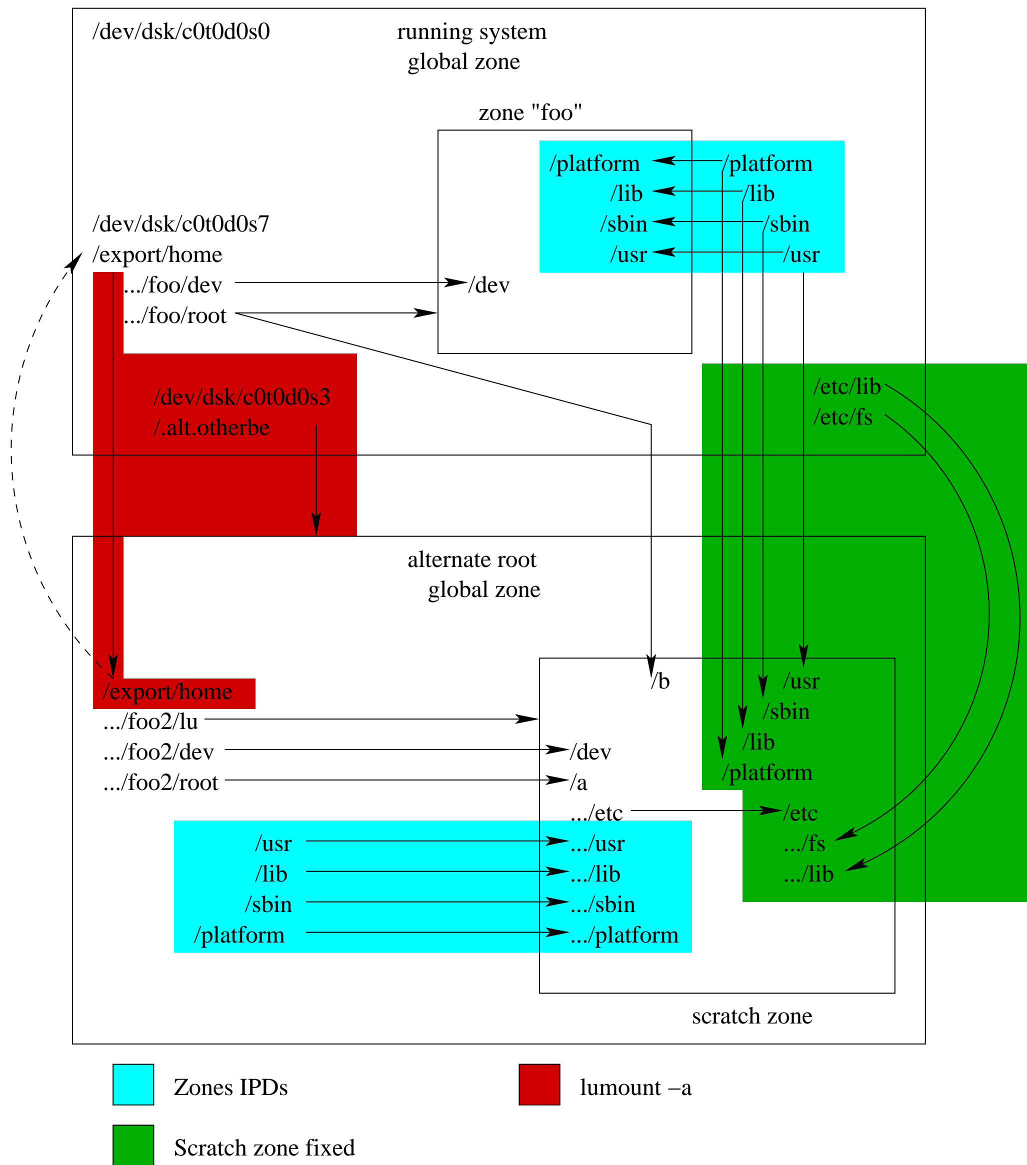

Figure 2: Scratch zone on shared file system

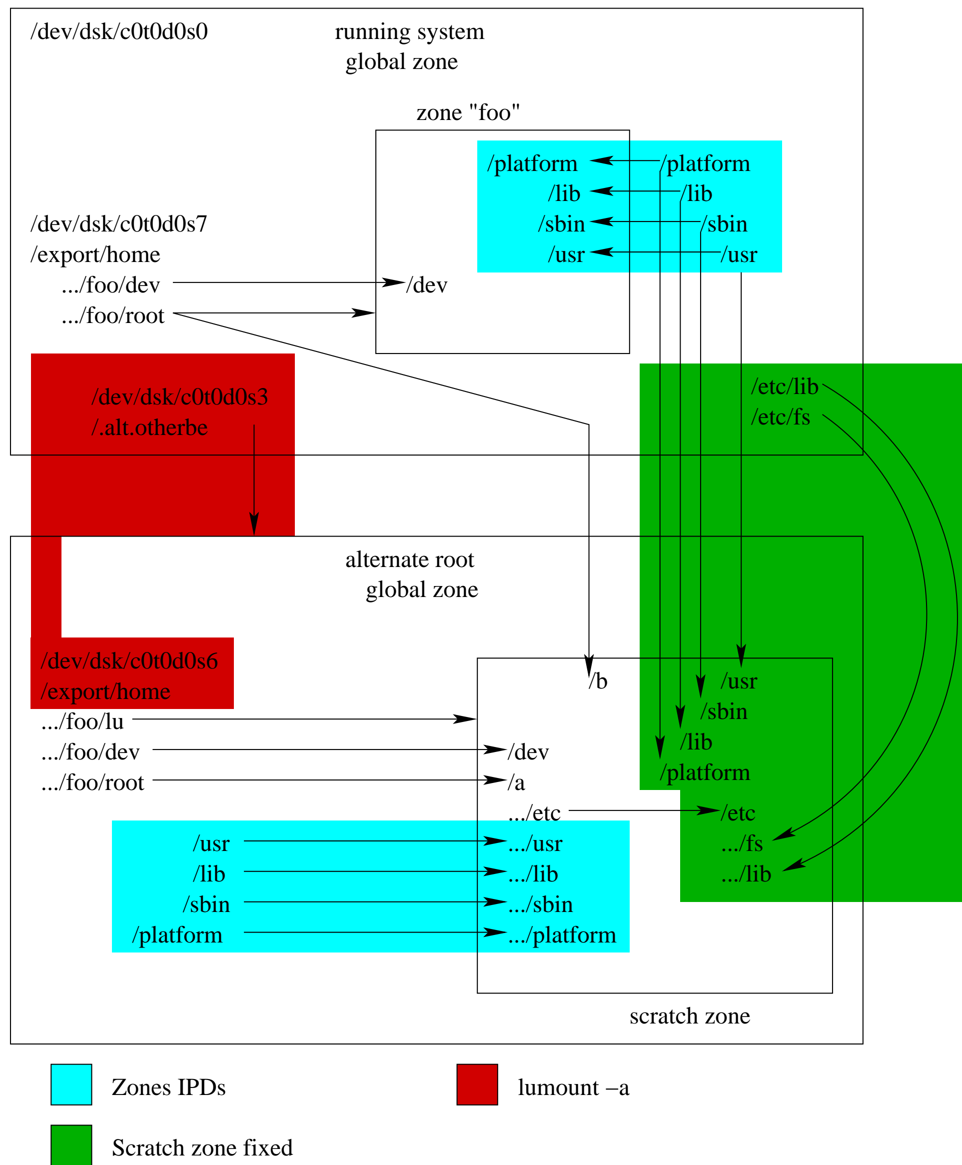

Figure 3: Scratch zone on unshared file system

this path ("/a/$ZONEPATH") ends up being used as-is. And because there's no "other" zone to speak of (and no `lusync` operation with which to contend), the "/b" mount point is not present.

```
/a/export/myzone/lu read-write on swap
/a/export/myzone/lu/etc read-write
  on /a/export/myzone/root/etc
/a/export/myzone/lu/var read-write
  on /a/export/myzone/root/var
/a/export/myzone/lu/etc/lib read-only on /etc/lib
/a/export/myzone/lu/etc/fs read-only on /etc/fs
/a/export/myzone/lu/lib read-only on /lib
/a/export/myzone/lu/sbin read-only on /sbin
/a/export/myzone/lu/platform read-only on /platform
/a/export/myzone/lu/usr read-only on /usr
/a/export/myzone/lu/tmp read-write on swap
/a/export/myzone/lu/a read-write
  on /a/export/myzone/root
/a/export/myzone/lu/dev read-write
  on /a/export/myzone/dev
/a/export/myzone/lu/a/lib read-only on /a/lib
/a/export/myzone/lu/a/platform read-only
  on /a/platform
/a/export/myzone/lu/a/sbin read-only on /a/sbin
/a/export/myzone/lu/a/usr read-only on /a/usr
/a/export/myzone/lu/proc read-write on proc
/a/export/myzone/lu/system/contract read-write on ctfs
/a/export/myzone/lu/etc/svc/volatile read-write
  on swap
/a/export/myzone/lu/etc/mnttab read-write on mnttab
/a/export/myzone/lu/dev/fd read-write on fd
```

This is depicted graphically in figure 4 on page 18.

5.2 Zones and NFS

The miniroot environment (used for regular upgrade and for the interim solution) is based on NFS; the system itself boots over NFS and all of the tools available in the miniroot are mounted from the install server.

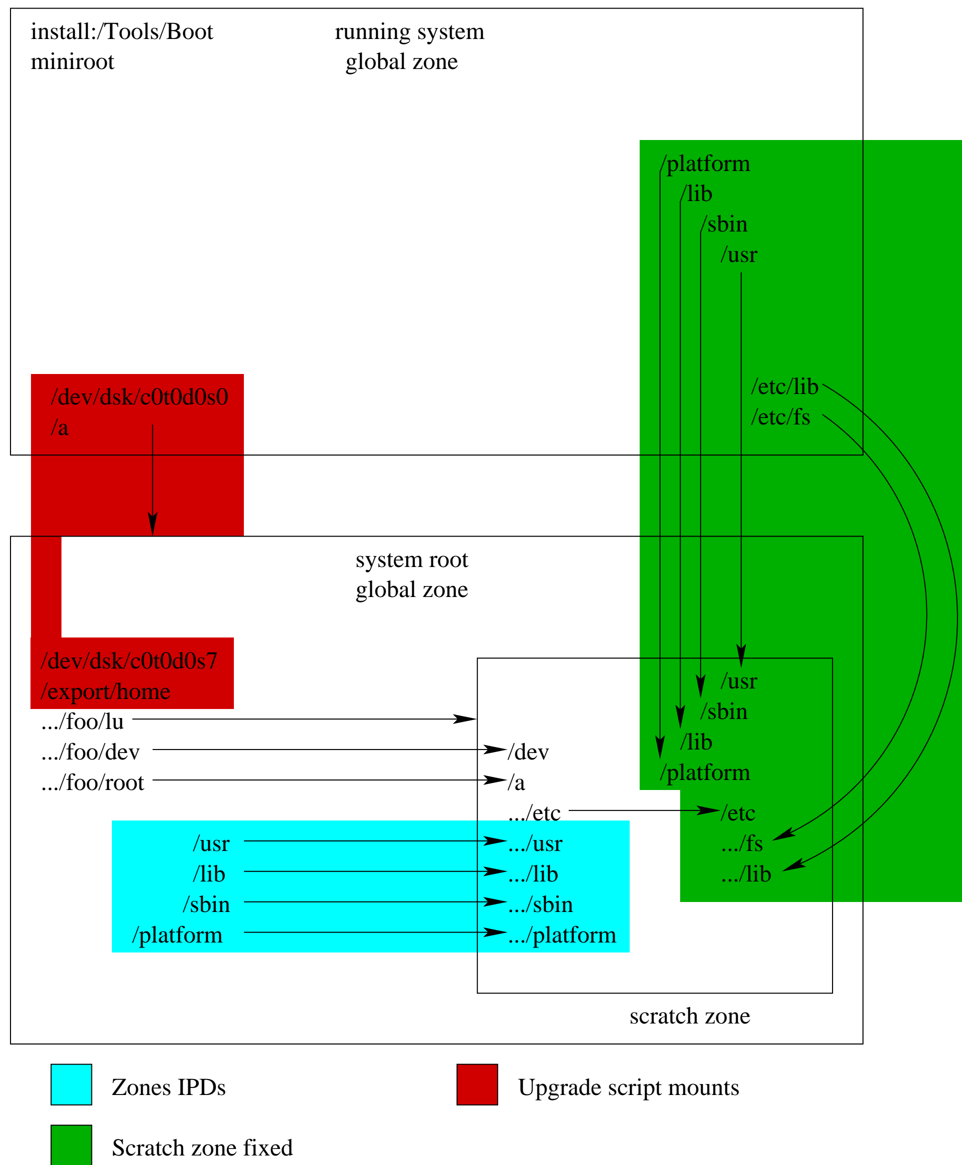

Figure 4: Scratch zone in miniroot environment

Unfortunately, Solaris Zones and NFS are not entirely compatible with each other. In particular, the fix for CR 5041169 prohibits a process from entering a zone if it has any NFS-resident files open or if any part of its address space mapped on an NFS-resident file. This means that when `zoneadmd` attempts to use zone_enter, it will get EFAULT in return unless that executable and all of the shared libraries it references are on something other than NFS.

The roots of this problem are complex. In short, though, NFS associates the network I/O operations with the zone that initiates the file system I/O operations, even when someone else actually mounted the file system.

In order to work around this, we could copy the miniroot into some local storage (tmpfs or the root slice of the system to be upgraded), or we could create a UFS image of the miniroot in a file presented by lofi to mount in the zone. The problem with all of these options is expense: they require either a large amount of additional RAM to hold the whole miniroot, or additional disk space on the system to be upgraded, above what is normally required for upgrade.

Instead, we will deliver a simple patch to NFS (and rpc, ktli, and tcp) that allows the NFS client operations to take place entirely in the global zone when nfs_global_client_only is set in /etc/system. This solution will require a contract between admin/install and ON for this Project Private flag. It is not expected that this solution will suffice for any other NFS-and-Zones combination other than the miniroot.[3]

6 User Interface

6.1 LU

The only new LU-related feature proposed is the "-a" flag to `lumount`. This flag causes "all" of the file systems to be mounted within the specified alternate boot environment. File Systems that are shared with the current boot environment are mounted as read-only, so that packaging and other administrative commands cannot accidentally modify the running system.

```
# lumount -a mybe
# luumount mybe
```

[3]In particular, it will not solve the existing SunGrid and Zones deployment problems.

6.2 Zones

The zoneadm command is enhanced with a "-R" flag that specifies the root of the alternate boot environment to use, and two new subcommands: mount and unmount. These new subcommands are intended to be private to admin/install (for now, at least) and work as follows:

```
# zoneadm -R /.alt.mybe -z test1 mount
# zoneadm -R /.alt.mybe -z test1 verify
# zoneadm -R /.alt.mybe list -v
  ID NAME       STATUS       PATH
   0 global     installed    /.alt.mybe
   - test1      mounted      /.alt.mybe/export/test1
# zoneadm -R /.alt.mybe -z test1 unmount
```

The other subcommands (ready, boot, halt, reboot, install, and uninstall) are not supported in an alternate root environment. (Of these, only install and uninstall make any possible sense for alternate root environments, and these aren't currently needed.)

Transition from "installed" to "mounted" state, and then "mounted" back to "installed" are permitted. Other transitions are not permitted.

Note that it is possible to "mount" and "unmount" installed zones in the current boot environment. This results in the same basic set of mount points as for a scratch zone in an alternate boot environment, except that "/b" is never mounted, and the zone name used by the kernel is the administrative zone name; no "SUNWlu" scratch name is required.

6.3 Packaging

The pkgadd and pkgrm commands already have a "-R" argument. This usage is retained, but enhanced so that if zones are found in the root specified, they're used.

A new "-O zonelist=⟨names...⟩" private option is added to these commands as well. By default, pkgadd and pkgrm install or remove the named packages in every zone on the system. This option limits the actions to take place only on the specified zones.

If any of the zones specified doesn't exist, then an error is returned and the command does nothing. The global zone may be specified if desired,

and has the obvious effect. The actual order in which the zones are processed is not affected by this option; the order is determined internally.

```
# pkgadd -R /.alt.mybe -O zonelist='test1 test2' \
  -d /tmp/pkgs SUNWfoo
```

The reason that zone ordering isn't affected is that with the existing packaging command design, for an install, the global zone must be processed first, and for a remove, it must be processed last. Violating that ordering will cause packaging scripts to fail to operate properly. In addition, the actual ordering of the zones to be processed during the install of a set of packages doesn't matter, because there's no way to "pause" in the middle of the install. The operation itself is effectively atomic.

Note that the other packaging commands (pkgadm, pkgchk, pkgask, pkginfo, and pkgparam) support the "-R" option, but do not have specific support for Zones. None is proposed here. (If it's required, it would be outside the scope of the scratch zone design and would be part of the packaging tools plan.)

7 Implementation

7.1 libzonecfg

This library is used by both the Zones tools and by the packaging tools (via libinst). It provides private (undocumented) interfaces for reading and writing zone configuration files.

Two main features are added in this design:

- Support functions for alternate root environments are added along with internal changes to make the library use the supplied root path when necessary. This is provided by three new functions:

```
void zonecfg_set_root(const char *);
const char *zonecfg_get_root(void);
boolean_t zonecfg_in_alt_root(void);
```

The first two functions set and get the (absolute) path to the alternate root environment. This path is “” (the null string) when not set. The last function is just a convenience wrapper; it’s equivalent to testing this expression:

```
*zonecfg_get_root() != '\0'
```

The expected usage of these functions looks like:

```
while ((val = getopt(argc, argv, "R:")) != EOF) {
    switch (val) {
    case 'R':
        zonecfg_set_root(optarg);
        break;
    }
}
```

- Support for the scratch zone name mapping functionality is added. Although somewhat specific to the operation of LU, this information must be accessed and maintained by both ON and admin/install tools, and thus should be implemented in one common location.

```
FILE *zonecfg_open_scratch(const char *rootpath,
    boolean_t createfile);
void zonecfg_close_scratch(FILE *);
int zonecfg_lock_scratch(FILE *);
int zonecfg_get_scratch(FILE *, char *zonename,
    size_t zonenamelen, char *kernname,
    size_t kernnamelen, char *altroot,
    size_t altrootlen);
int zonecfg_find_scratch(FILE *,
    const char *zonename, const char *altroot,
    char *kernname, size_t kernnamelen);
int zonecfg_reverse_scratch(FILE *,
    const char *kernname, char *zonename,
```

```
    size_t zonenamelen, char *altroot,
    size_t altrootlen);
boolean_t zonecfg_is_scratch(
    const char *kernname);
int zonecfg_add_scratch(FILE *,
    const char *zonename, const char *kernname,
    const char *altroot);
int zonecfg_delete_scratch(FILE *,
    const char *kernname);
```

These functions fall into three categories: file access, mapping look-up, and map modification.

The file access functions are `zonecfg_open_scratch`, to open the mapping file; `zonecfg_close_scratch`, to close the mapping file; and `zonecfg_lock_scratch`, to apply a write lock to an open mapping file. By default, the mapping file is always opened with a read lock using fcntl(fd, F_SETLKW, F_RDLCK), so accesses to the file always return consistent data.

The mapping look-up functions are `zonecfg_get_scratch`, to get sequential entries in the file; `zonecfg_find_scratch`, to return a kernel zone name given a zone and alternate root; `zonecfg_reverse_scratch`, to return zone name and alternate root given a kernel zone name; and `zonecfg_is_scratch`, to determine if a given zone name from the kernel is a scratch zone. The "is scratch" determination is done with a simple strncmp, so it can be used to avoid reading the scratch mapping file when not needed.

The two modification functions are `zonecfg_add_scratch`, to add a new mapping entry to the file, and `zonecfg_delete_scratch`, to remove an entry. The caller must hold a write lock on the file when calling these functions.

The expected usage pattern for installing a new scratch zone looks like this:

```
fp = zonecfg_open_scratch("", B_TRUE);
zonecfg_lock_scratch(fp);
```

```
if (zonecfg_find_scratch(fp, zonename, altroot,
    NULL, 0) == 0) {
    /* handle error; zone is already mounted */
}
/* mount zone here and create it in the kernel */
zonecfg_add_scratch(fp, zonename, kernname,
    altroot);
zonecfg_close_scratch(fp);
fp = zonecfg_open_scratch(zoneroot, B_TRUE);
ftruncate(fileno(fp), 0);
zonecfg_add_scratch(fp, zonename, kernname, "/");
zonecfg_close_scratch(fp);
```

The special `rootpath` argument to `zonecfg_open_scratch` is provided so that the mapping can be written inside the zone as well. This is a convenience feature, so that tools running inside the scratch zone can learn the zone's "true" name. This allows the `zonename(1)` utility to work as expected inside a scratch zone and avoids any special case logic in the lookup functions.

Note that there is intentionally no "unlock" function. The usage model is very simple: open the file, get the information needed, close it. To support that usage, the open routine automatically applies a read lock to the file. If we were to support an "unlock" function, the implication would be that some user intends to keep the file open after having done some logically complete operation. This is itself a usage error, because the file should be closed at that point, not just unlocked.

- Support for the UUID values is added. This consists of new lookup functions:

```
int zonecfg_get_byuuid(const char *uuid,
    zone_dochandle_t);
int zonecfg_get_snapshot_byuuid(const char *uuid,
    zone_dochandle_t);
int zonecfg_create_snapshot_byuuid(
    const char *uuid);
```

```
int zonecfg_get_uuid(zone_dochandle_t, char *uuid,
    size_t uuidlen);
```

No additional calls are needed; a UUID is automatically assigned any time a zone entry is read from a file and it doesn't already have a UUID.

These lookup functions are used by zoneadmd to make sure that the zone root mounted into the scratch zone (as /b) is the same zone as mounted for scratch.

7.2 zonename

This change is trivial. For performance reasons, the utility calls `zonecfg_is_scratch` to determine whether the name of the current zone read from the kernel represents a scratch zone. If so, then it calls `zonecfg_reverse_scratch` to find out the real (administrator-specified) name for the zone.

7.3 zoneadm

In order to communicate with `zoneadmd`, `zoneadm` creates a door in /var/run/zones as $ZONENAME.zoneadmd_door. This door will be moved into $ALTROOT when "-R" is used, so that `zoneadm` doesn't need to predict the kernel zone name before `zoneadmd` establishes it (an impossibility) and so that the same zone in multiple boot environments won't conflict.

Specific and sundry issues:

- The options parsing needs to handle "-R" to set the alternate root, and signal that scratch zones are in use.
- The fetch_zents function, which reads the list of zones in the kernel, needs to be updated to understand scratch zones. When used on an alternate root, the scratch mapping functions will be used to convert the kernel's zone name back to the expected zone name.

- The grab_lock_file and get_doorname functions need to be updated to handle an alternate root path.
- The -R flag needs to be passed to `zoneadmd` so that it knows what alternate root environment to use.
- Commands not allowed in an alternate root environment (such as actually booting the zone) need to be prohibited.
- Verification of resource controls, pools, and network interfaces (the items intentionally not used in scratch zones) needs to be skipped.

7.4 zoneadmd

The `zoneadmd` changes span two files (zoneadmd.c and vplat.c). In general, zoneadmd.c has the high-level functionality, while vplat.c has lower-level portions. (This isn't strictly true, as, for example, the mounting of file systems that must take place within the zone is done in zoneadmd.c, while those mounted from outside the zone are done in vplat.c.)

In zoneadmd.c, we split out the special "mount early" file systems (/proc, /system/contract, /etc/svc/volatile, and /etc/mnttab) from `zone_bootup` into a new function. This is because we need to mount these in the scratch zone, but we don't want to actually boot the zone.

Normally, the `zoneadmd` process hangs around as long as the system console is alive. This is done by calling `serve_console`. As noted earlier (see item 3 on page 8) though, the scratch zone has no console. As an ugly kludge, we just sleep for five seconds (waiting for the first and only door call from `zoneadm`) rather than calling the `serve_console` function.

As with `zoneadm`, the door is moved to $ALTROOT and handling is added for "-R".

The most intricate changes are in vplat.c. Here we have several major components:

- A new function, `resolve_lofs`, which was originally developed as part of the first prototype, is added to handle the resolution of paths involving lofs mount points up to the root of the tree.

  In the original prototype, this function read the table of mounted file systems and iterated over the path supplied, removing lofs paths until no more changes were possible.

The problem with this strategy is that it could climb "too far" out of a box prepared by the administrator. For example, if the administrator created /export/notouch and then did a read-only loopback mount of /export/commonstuff on top of it, and exported only /export/notouch to the zones, we'd incorrectly resolve back to the writable /export/commonstuff.

The resolution is to temper the loop described above and only resolve the read-only lofs mounts created by "lumount -a" for shared file systems, as that's what was original intended. In order to do that, we observe the actual root of the alternate boot environment: any lofs mount that has a special node outside of that path and a mount point inside that path is used as the last entry to resolve the external path.

As part of the return value, any boot environment internal read-only lofs mounts encountered cause the path to be treated as read-only.

- In general, before using a path, we call `resolve_lofs`, but only if we're in an alternate root environment. It's isolated to this case for both performance and testing impact reasons.
- The root of the zone is normally $ZONEPATH/root. In the scratch zone, though, this is changed to $ZONEPATH/lu, so that $ZONEPATH/root can be mounted onto /a inside the zone. Thus, every place that deals with the zone root subdirectory needs to be updated to deal with the special path to the lu subdirectory instead.
- IPDs (loopback mounts from the global zone) are special. We handle these by doing a loopback mount of the specified directory from the alternate root environment, rather than from the running machine. This makes the contents of /a inside the zone fairly represent the "zone to be upgraded."
- When mounting other things (as specified in the configuration files) into the zone, we must make two changes. First, if a block special file is referenced, this may well be something that's already mounted on the system (e.g., in a zone of the same name on the running system) and can't be mounted twice. We scan for an already-mounted file system and, if one is found, these entries are converted into read-only loopback mounts on the already-mounted file system.

Second, if there are explicit lofs mounts configured for the zone, then these need to be resolved and, if duplicated, converted to read-only.

- The contents of the zone itself is built essentially ad-hoc when needed because there seems to be no reason to cause the system to carry around this unusual directory structure in every zone. To do this, we create (mkdir) $ZONEPATH/lu and mount tmpfs on it. Inside that new root directory, we:
  - Create the directories described in item 4 on page 9.
  - Create a symlink from /bin to usr/bin.
  - Mount up $ZONEPATH/root on /a.
  - Find the root for the zone in the current boot environment. If found, then create /b and mount the current zone's root on this as read-only for support of `lusync`. If not found, then don't create /b at all.
  - Mount /etc and /var in the zone from read-write loopback on /a/etc and /a/var.
  - Mount /etc/lib, /etc/fs, /lib, /sbin, /platform, and /usr inside the zone as read-only loopback on the same directories from the running system.
  - Write the mapping entry into the zone itself at /tmp inside the zone.
- When creating the zone in the kernel, we must come up with a scratch name for the zone and establish it.
  - We check that the zone root isn't already mounted somewhere (which would indicate that the user has the zone path specified on a shared file system) and that it isn't writable by anyone else (which would mean that it's loopback mounted elsewhere in the system).
  - We attempt to create "SUNWlu-$ZONENAME" in the mapping file first. If this fails, we just use random() to generate names of the form "SUNWlu.%16s" until one succeeds.

  – We skip over resource controls, devfsadm registration, and network interface configuration.

An unresolved design issue is that after doing zone_destroy, the zone is mostly gone (and is in fact on death row), but there may still be creds floating about in the system that reference the zone_t, and which pin down zone_rootvp causing the tmpfs root to be held busy.

Worse still, tmpfs doesn't honor MS_FORCE and it does go out of its way to avoid implementing it, so all we can reasonably do here is loop for a while and retry the umount2 call until it either succeeds or we give up in frustration and log an error.

7.5 liblu

This library is used by `lumount`, and the changes here are to support the "-a" option to that command.

To do this, we modify `lu_beoGetFstblFilterSwapAndShared` so that if all file systems are requested, duplicates (those that are already in the system) are converted to lofs mounts.

We also mount up /var/run and /tmp inside the alternate root environment for completeness. The latter is used by the `zoneadm` door, but the former is not strictly required.

The actual "allfs" flag is passed into the following functions by `lumount`:

```
lu_beoGetFstblToMountBe
lu_beoMountBeByBeName
lu_beoMountBeByIcfFile
```

In support of this, the `lu_tsfCopyEntry` function is enhanced so that, instead of returning a boolean pass/fail response, it returns the index number of the newly-added entry. This allows the convert-to-lofs logic to find the entry easily and update it.

7.6 lumount

This utility comes in two parts. One part is in ludo_mount.c. This handles the new "-a" flag and passes it to the second part as an XML tag ("allFS=yes/no").

The second part extracts the XML flag and, if present, passes the derived boolean allFS flag into the appropriate lu_beoMountBeBy* function.

7.7 libinst

This library gets substantial modifications to deal with scratch zones. This is where the bulk of the work for packaging support is done. The main issue to deal with is that there are now two names for each zone that need to be kept separated and used when appropriate: one is the name of the zone as the administrator knows it (the "real" name), and the other is the kernel's name for the zone (the "scratch" name).

The real name is used when communicating with the user and when invoking the `zoneadm` tool. The scratch name is used when communicating with the kernel (specifically when doing zone_enter) and when a system-wide unique name is required (as when creating the file names for "lock objects").

One simplification is made: set_inst_root is made to set the libzonecfg root, so that it's no longer necessary to pass the root directory pointer into any of the zones.c-related interfaces.

Some specific features:

- New functions to handle the "-O zonelist" feature:

  ```
  int z_set_zone_spec(const char *zonelist);
  int z_verify_zone_spec(void);
  boolean_t z_on_zone_spec(const char *zonename);
  boolean_t z_global_only(void);
  ```

- A function is added to get the scratch (kernel) zone name that corresponds to a particular zone in a zoneList_t: z_zlist_get_scratch (naming and signature is similar to existing z_zlist_get_* functions).

  Longer term, Admin/Install should just be ignorant of the internal implementation details of Zones. It should be able to deal in just the basics of the zone location (its name as known to the user and boot environment), and not need access to details such as the kernel's name for the zone. To make that happen, libinst/zones.c needs to be

minimized so that it just execs the necessary Zones utilities to do the work needed, rather than linking to libzonecfg.

- Special handling of the `zoneadm` invocation is added to include the "-R" option when referring to an alternate boot environment and using mount/unmount (at all times) instead of boot -s/halt. Note that this results in a substantial simplification of the admin/install code, as these new commands are completely synchronous and thus the logic that sleeps and polls Zones and SMF state variables is eliminated.

- The handling of internal states is extended to include ZONE_STATE_MOUNTED for scratch zones.

7.8 pkgadd/pkgrm

Support is added to handle the hidden "-O zonelist" option, and pass the scratch zone name to functions that must enter the zone.

When an alternate root is in use, these two programs must invoke `pkginstall` and `pkgremove` using "-R /a". This allows the internal packaging utility function, which is run within the scratch zone constructed by `zoneadmd`, to access the zone's mounted root file system.

The $SUNW_PKG_INSTALL_ZONENAME variable (set by _zonesCreateLocalZoneBootEnv) must be set to the administrative name for the zone, not the scratch name.

8 Other Design Issues

8.1 Installation

Fortunately, `zoneadmd` was designed to be stopped and restarted on a running system, and the new binaries use the same door message formats. This means that we can install the required ON patch without shutting down the system, or even disturbing any of the running zones. Given the nature of the fix and the problems involved with downtime, it's likely that customers will want to avoid multiple reboots during upgrade.

The required procedure is fairly simple:

- Unpack the new binaries into temporary names in the same file system with the old binaries. For example:

```
usr/bin/zonename.new
usr/sbin/zlogin.new
usr/sbin/zoneadm.new
usr/sbin/zonecfg.new
usr/lib/sparcv9/libzonecfg.so.1.new
usr/lib/libzonecfg.so.1.new
usr/lib/zones/zoneadmd.new
```

- Back up the existing (old) binaries so that the patch can be backed out if desired.
- Move (`mv`; not copy) the new binaries into place, starting with libzonecfg, then zonename, zoneadm, zonecfg, and finally zoneadmd.
- Optionally determine if any copies of `zoneadmd` are running (not necessary, but helpful to avoid error messages).
- "pkill -x zoneadmd"
- Optionally start up new copies of `zoneadmd`, if they were checked above.

8.2 Design for Rampart

The project containing the scratch zone feature is scheduled to deliver before Rampart (Trusted Solaris) delivers. Thus, no changes to the scratch zone design are needed for the first delivery. However, the two are changing the same areas of the code, and thus, in order to make the scratch zone project complete, we need a plan for how Rampart will adopt these changes.

The first issue is that Rampart assigns each zone a unique label, and relies on being able to locate a zone in the system by searching on the label. This is done inside networking as part of the "polyinstantiated port" functionality. Since scratch zones are intentionally not designed to do any networking (they don't have configured addresses that are distinct from

running zones, and thus can't access the network without conflicting with the running zones), this should not be a problem. The zone_create function should detect the "SUNW" name on the zone, and not insert it into (or check for it on) the `zonehashbylabel` list.

Since labels represent security classifications, we must also check them in `zoneadmd`. When starting up a scratch zone for an alternate boot environment, we must check that the label on the running zone is the same as that configured on the scratch zone. If not, then omit the "/b" mount point, as it would allow leakage of data from one label to another. This should be handled in the same logic that does the UUID check.

Finally, in order to get the label for a zone in an alternate boot environment, `zoneadmd` will need to open the /etc/security/tsol/tnzonecfg file using the root path provided by the user, and use the tsol_fgettpent() interface to bypass the normal directory service lookup.

8.3 Leaking Into Whole-Root Zones

During review of this document, one of the reviewers noted that information can leak from the global zone into whole-root zones that would otherwise not be present.

The scratch zone's root has mounted only those things from the global zone that would already ordinarily be present in a sparse-root zone due to the IPD mounts. However, whole-root zones do not use the IPD mounts, and thus the only things visible there are the things that the administrator explicitly installed into the zone.

Thus, if the administrator has special items in the global zone (for example, some non-Sun licensed software that installs to /usr) and he uses Live Upgrade to upgrade the system, these things will become visible during the upgrade process, allowing a malicious zone administrator to copy these things out of the global zone.

This problem can be addressed by creating a "pristine" zone with the usual tools, and then mounting the usual directories into the scratch zones from this temporary. We would then have to tell `zoneadmd` to use the special zone (perhaps by giving it a well-known name) during scratch zone construction.

Doing this will be expensive. Installing a zone on S10 FCS requires installing some 1008 packages (containing 2.95GB of data in some 123,828

files), and takes around an hour to complete on current systems.

Because we know how to solve the problem, but the expense of solving it is high and the customer interest in having this solved doesn't appear to be present, we plan to defer this item until later. However, the question ("do you use whole-root zones rather than sparse-root zones to exclude particular software from ever showing up within a zone, even in an unconfigured and unused state?") will be included in the detailed customer survey for the overall project.

9 Unsolved and Out-Of-Scope Issues

9.1 Packaging Features

Currently, the "-O zonelist" option that specifies a particular set of zones in which to add or remove packages is undocumented and won't be exposed to customers. It's quite likely, though, that it will be needed by users who are either repairing problems or just doing normal administration on the system (particularly with whole-root zones).

Moreover, this zone list feature is actually a superset of the existing "-G" and (unused but reserved) "-z" options. Given the inflexibility of these other options, it would make sense to migrate to the zone list feature instead.

9.2 Command Exposure

Should the new `zoneadm` subcommands and scratch zone features be made user-visible? Currently, they're not, and they're intended to be implementation details of the `luupgrade` feature. However, it's likely that to recover from failures, users will want to mount zones individually and modify them.

Also, it will be possible to mount a scratch zone for a zone configured (but not booted) on the running system, and have the zone's root show up in /a. This will likely be useful in the future, as it will allow the administrator to restore an old zone from tape and then run an upgrade on it without using LU at all. Without a /a mount, this would be impossible; for the same reason that scratch zones must use the /a mechanism when dealing with an alternate boot environment.

9.3 Zone Configuration

It would be possible to add a -R option to `zonecfg` so that zone configuration can be modified within a mounted alternate boot environment. Although simple, this does not appear to be necessary at this time. (This is also linked to Command Exposure.)

9.4 Context

As long as the user sticks to just `lucreate` and a few of the other LU-family commands, usage (with repeated "-n bename" options on each command) isn't too awful. However, using "-R" to access a mounted alternate boot environment in the Zones and packaging tools is cumbersome and gets bad when multiple commands are required to do the job.

There should be some other way to use common administrative commands with alternate boot environments. There should be a way for an administrator to specify that a set of configuration commands (creating, destroying, or configuring zones, for instance, or manipulating filesystems) are to be applied against a given boot environment.

It would be nice if this worked in some sort of high-concept "zlogin" manner, where the boot environment itself could be "entered" for purposes of administration. The difficult part is that the system binaries (commands and libraries) need to come from the running system rather than the alternate boot environment, so this isn't just a straightforward chroot.

It's also worth noting that zones and boot environments have many semantic similarities, and having different models and sets of tools for each is at least confusing.

This needs more investigation.

9.5 NFS

We are clearly punting on the issue of NFS inside non-global zones. This will require a much bigger project to fix.

The underlying issue is that the current model assumes that the zone of the process doing the file system I/O operation is the right one to use for the underlying network I/O. This is not true in all cases; and perhaps isn't true in many cases. Instead, administrators expect that the network I/O will be based on the zone that mounted the file system.

(After all, it doesn't matter whether the non-global zone has access to the raw disk slice behind a UFS file system; I/O still works from inside the zone.)