Advanced - Split-root installation

Note: you might want to scroll down to "How do I do this?" for initial set-up if you're less interested in the theory and more in practice, and to "Upgrades" for subsequent `pkg upgrade`'s of the setup.

Another note: For other enhanced OI setup ideas see Advanced - Creating an rpool manually, Advanced - Manual installation of OpenIndiana from LiveCD media, Advanced - ZFS Pools as SMF services and iSCSI loopback mounts, Zones as SMF services or Using host-only networking to get from build zones and test VMs to the Internet. Not all of these articles are applicable and limited just to only what it says on the label

What are we doing?

The configuration below used to be relatively fragile in setup, so you should not do it without practice on remotely-accessed computers without some means of access to the console (be it IPMI or a colleague who can help as your hands and eyes over the phone). It is also not a required setup, though may be desired (and beneficial) in a number of cases.

Details on this procedure are tracked as illumos/OI issue #829, and hopes for its automation and support in the installer – as #4354.

This document may contain typos or factual errors. Test before you try. While great care has been taken to verify the sample commands, by making a full installation and split-rooting it by copy-pasting this page's instructions to verify it completely, some errors may still lurk... code is code

UPDATES: This has now been fully walked-through for modifying a fresh installation without leaving the LiveCD environment (in VirtualBox) by copy-pasting the commands from this page into a `root`'s shell (`bash` via `jack`'s "`sudo su -`"). Afterwards it was also tested that the instructions for `beadm`-cloning and using that as a multi-filesystem source, the `lofs` variant for a single-dataset origin, and "diving into snapshots", for a subsequent split-root procedure all worked. Updating the resulting alternate BEs (with reenabled compression after a `beadm create` cloning) also worked.

Only networked cloning remains to test (sending of OS image from an origin system over `rsync` into the prepared split-root hierarchy on the target system); but since this boils down to different `rsync` parameters within an overall same methodology – I don't expect any specific problems there. Just remember to think about what you copy-pase into where

UPDATE 20141130..20141208: Procedure was verified and worked "as is" with OmniOS bloody-151013 (installation from last week's USB image, in-place updated after boot to include split-root setup, `beadm-new.sh` and `pkg -R /a ...` update to install the latest bloody bits).

What?

It may be desirable for a number of reasons to install the OI global zone not as a single uncompressed dataset (as was required until recently – before LZ4 compression became supported in GRUB and `rpool` with oi_151a8 dev-release), but as an hierarchy of datasets with separate `/usr`, `/var`, `/opt` and maybe other datasets. While some such datasets contain parts of the OS installation, others like `/var/mail` or `/var/logs` contain "usual" data which you may want shared (not cloned) between the different BE's (Boot Environments). This way whenever you reboot into one BE or another, such as during development or tests of new releases (and perhaps switching back to a "stable" BE for some reason), your computer's logged history would be appended to the same file regardless of the BE switcheroo.

Note that while these instructions are tailored for OpenIndiana, the history of this procedure in my practice tracks back to Solaris 10 and OpenSolaris SXCE (with LiveUpgrade instead of `beadm`). Much of the solution is also applicable there, though some back-porting of the procedure may be needed.

Why?

What benefits can this bring? On one hand, greater compression (such as `gzip-9` for the main payload of the installed system binaries in `/usr`). The default installation of OI (fresh from GUI LiveCD) is over 3Gb, which can compress over 2.5x to about 1Gb with `gzip-9` applied to `/usr` and `/var`. Actually, given the possible SNAFUs with this setup and ability to compress just over 2x (down to 1.2Gb for a "monolithic" rootfs dataset on pools with `ashift=9`, or about 2Gb on pools with `ashift=12`) with `lz4` which is now supported for the root datasets, this one benefit may be considered moot. Still, either of these compressions brings some benefit to space-constrained systems, such as installations on cheap but small SSDs, or redistributable VM images (illumos-based appliances, demo's, etc.) with minimal download size. Also, less on-disk data means less physical IOs (both operations and transferred bytes) during reads of the OS and its programs, at the (negligible) cost of decompression.

Another benefit, which is more applicable to the shared datasets discussed above, is the ability to assign ZFS `quota` to limit some datasets from eating all of your `rpool`, as well as `reservations` to guarantee some space to a hierarchy based at some dataset, or `refreservations` to guarantee space to a certain dataset (excluding its children, such as snapshots, clones and datasets hierarchically "contained" in this one). This helps improve resilience of the system against behaviours which can overflow your `rpool` storage – such as the programs which you debug creating too many coredumps.

In case of SSD-based `rpools`, or especially of slow-media root pools such as ones on USB sticks or CF cards, it may also be beneficial to move some actively written datasets to another, HDD-based data pool, in order to avoid excessive wear of flash devices (and/or lags on USB devices). While this article does not go into such depths, I can suggest that and `/var/cores` can be relocated to another pool quite easily (maybe also `/var/crash` and some others). And in case of slow-media and/or space-constrained `rpools` you might want to relocate the `swap` and `dump` volumes as well (for `swap` it may suffice to add a volume on another pool and keep a small volume on `rpool` if desired – it is not required, though).

Also note that one particular space-hog over time can be `/var/pkg` with the cache of package component files, and its snapshots as parts of older BEs can be unavoidable useless luggage. Possibility of separating this into a dedicated dataset, within each BE's individual rootfs hierarchy or shared between BEs, is a good subject for some separate research.

What to avoid and expect?

Good question. Many answers :

- One problem for the split-root setup (if you want to separate out the `/usr` filesystem) is that OpenIndiana brings `/sbin/sh` as a symlink to `../usr/bin/i86/ksh93`. Absence of the system shell (due to not-yet-mounted `/usr`) causes `init` to loop and fail early in OS boot.
  When doing the split you **must** copy the `ksh93` binary and some libraries that it depends on from `/usr` namespace into the root dataset (`/sbin` and `/lib` accordingly), and fix the `/sbin/sh` symlink. The specific steps are detailed below, and *may* have to be repeated after system updates (in case the shell or libraries are updated in some incompatible fashion).

  > ⚠ My earlier research-posts suggested replacement of `/sbin/sh` with `bash`; however, this has the drawback that the two shells are slightly different in syntax, and several SMF methods need to be adjusted. We have to live with it now – `ksh93` is the default system shell, it just happens to be inconveniently provided in a non-systematic fashion. Different delivery of `ksh93` and the libraries it needs is worthy of an RFE for packagers (tracked as issue #4351).

- Another (rather cosmetic) issue is that many other programs are absent in the minimized root without `/usr`, ranging from `df`, `ls`, `less` and `cat` to `svc*` SMF-management commands, `vi` and so on. I find it convenient to also copy `bash` and some of the above commands from `/usr/bin` into `/sbin`, though this is not strictly required for system operation – it just makes repairs easier
- A much more serious consequence of the absence of programs from `/usr` is that some SMF method scripts which initialize the system up to the "single-user milestone", including both `default` and `nwam` implementations of `svc:/network/physical`, rely on some programs from `/usr`. The rationale is that network-booted miniroot images carry the needed files, and disk-based roots are expected to be "monolithic". It is possible to fix some of those methods (except NWAM in the default setup, at least), but a more reliable and less invasive solution is to mount the local ZFS components of the root filesystem hierarchy (and thus guarantee availability of proper `usr`) before other methods are executed. This is detailed below as the `svc:/system/filesystem/root-zfs:default` service with `fs-root-zfs` script as its method.
  NOTE for readers of earlier versions of the document: this script builds on my earlier customizations of the previously existing `filesystem` methods; now these legacy scripts don't need many modifications (I did add just the needed checks whether a filesystem has already been mounted).
- Separation of `/var/tmp` into a shared dataset did not work for me, at least some time in the past (before the new `fs-root-zfs` service) – some existing services start before `filesystem/minimal` completes (which mounts such datasets) and either the `/var/tmp` dataset can not mount into a non-empty mountpoint, or (if `-O` is used for overlay mount) some programs can't find the temporary files which they expect.
  It is possible that with the introduction of `fs-root-zfs` this would work correctly, but this is not thoroughly tested yet.
- Likewise, separation of `/root` home directory did not work well: in case of system repairs it might be not mounted at all and things get interesting
  It may suffice to mount a sub-directory under `/root` from a dataset in the shared hierarchy, and store larger files there, or just make an `rpool/export/home/root` and symlink to it from under `/root` (with the latter being individual to each BE).
- Cloning BE's with `beadm` currently does not replicate the original datasets' "local" ZFS attributes, such as `compression` or `quota` or `(ref)reservation`. If you use `pkg image-update` to create a new BE and update the OS image inside it, you're in for surprise: newly written data won't be compressed as you expected it to be – it will inherit compression settings from `rpool/ROOT` (uncompressed or LZ4 are likely candidates). While fixing `beadm` in this behaviour is a worthy RFE as well (issue numbers #4355 for `pkg` and #3569 for `beadm` and `zfs`), currently you should work around this by creating the new BE manually, re-applying the (compression) settings to the non-boot datasets (such as `/usr`), mounting the new BE, and providing the mountpoint to `pkg` commands. An example is detailed below.
  Note that the bootable dataset (such as `rpool/ROOT/oi_151a8`) must remain with the settings which are compatible with your GRUB's `bootfs` support (uncompressed until recently, or with `lz4` since recently).
- Finally, proper mounting of hierarchical roots requires modifications to some system SMF methods. Patches and complete scripts are provided along with this article, though I hope that one day they will be integrated into `illumos-gate` or OI distribution (issue number #4352), and manual tweaks on individual systems will no longer be required.

How does it work (and what was fixed by patches)?

As far I found out, the bootloader (GRUB) finds or receives with a keyword the bootable dataset of a particular boot environment. GRUB itself mounts it with limited read-only support to read the illumos kernel and mini-root image into memory and passes control to the kernel and some parameters, including the information about the desired boot device (device-path taken from the ZPOOL labels on the disk which GRUB inspected as the `rpool` component, and which should be used to start reconstruction of the pool – all cool unless it was renamed, such as from LegacyIDE to SATA... but that's a separate bug) and the rootfs dataset number. The kernel imports the specified pool from the specified device (and attaches mirrored parts, if any), and mounts the dataset as the root filesystem (probably `chroots` somewhere in the process to switch from the miniroot image into the `rpool`), but does not mount any other filesystem datasets

Then SMF kicks in and starts system startup, passes networking and `metainit` (for legacy SVM metadevice support, in case you have any filesystems located on those) and gets to `svc:/system/filesystem/root:default` (implemented in `/lib/svc/method/fs-root` shell script) which ensures availability of `/` and `/usr`, and later gets to `svc:/system/filesystem/usr:default` (/lib/svc/method/fs-usr) and `svc:/system/filesystem/minimal:default` (/lib/svc/method/fs-minimal) and `svc:/system/filesystem/local:default` (/lib/svc/method/fs-local) which mount other parts of the filesystems and do related initialization. Yes, the names are sometimes counter-intuitive.

In case of ZFS-based systems, `fs-root` does not actually mount the root filesystem (it is already present), but rather ensures that `/usr` is available, as it holds the bulk of programs used later on (even something as simple and frequent in shell scripting as `grep`, `sed` and `awk`). The default script expects `/usr` to be either a legacy mount specified explicitly in `/etc/vfstab` (make sure to provide `-O` mount option in this case), or a sub-dataset named `usr` of the currently mounted root dataset. Finally, the script mounts `/boot` (if specified in `/etc/vfstab`) and the `libc.so` hardware-specific shim, and reruns `devfsadm` to detect hardware for drivers newly available from `/usr/kernel`.

The patched `fs-root` script (earlier) or the replacement `fs-root-zfs` script (later) introduces optional console logging (enable by touching `/.debug_mnt` in the root of a BE), and enhances the case for ZFS-mounted root and `usr` filesystems by making sure that the mountpoints of sub-datasets of the root filesystem are root-based and not something like `/a/usr` (for all child datasets), and mounts `/usr` with overlay mode (`zfs mount -O` – this – takes care of the issue number #997 at least for the rootfs components) – too often have mischiefs like these two left an updated system unbootable and remotely inaccessible.

The `fs-usr` script deals with setup of `swap` and `dump`, and the patch is minor (verify that `dumpadm` exists, in case sanity of `/usr` was previously overestimated). For non-ZFS root filesystems in global zone, the script takes care of re-mounting the `/` and `/usr` filesystems read-write according to `/etc/vfstab`, and does some other tasks.

Then `fs-minimal` mounts certain other filesystems from `/etc/vfstab` or from the rootfs hierarchy. First it mounts `/var`, `/var/adm` and `/tmp` from the `/etc/vfstab` file (if specified) or from rootfs child datasets (if sub-datasets exist and if `mountpoint` matches). The script goes on to ensure `/var/run` (as a `tmpfs`) and mounts other not-yet-mounted non-`legacy` child datasets of the current rootfs in alphabetic order.

The patched `fs-minimal` script (earlier) or the replacement `fs-root-zfs` script (later) adds optional console logging (enable by touching `/.debug_mnt`), and allows mounting of the three mountpoints above from a shared dataset hierarchy. If the default mounting as a properly named and mountpointed child of the rootfs failed due to absense of a candidate dataset, other candidates are picked: the script now looks (system-wide, so other pools may be processed if already imported) for datasets with `canmount=on` and appropriate `mountpoint`. First, if there is just one match – it is mounted; otherwise, the first match from the current `rpool` is used, or in absense of such – the first match from other pools which have the default `altroot` (unset or set to `/`). Another fix concerns the "other not-yet-mounted non-`legacy` child datasets of the rootfs" – these are now mounted also in overlay mode (again, issue number #997), to avoid surprises due to non-empty mountpoints.

Finally, `fs-local` mounts the other filesystems from `/etc/vfstab` (via `mountall`) and generally from ZFS via `zfs mount -a` (this also includes the rest of the shared datasets, and note that errors are possible if mountpoints are not empty), and also sets up UFS quotas and `swap` if there is more available now. No patches here 😅

While the described patches (see fs-root-zfs.patch for the new solution, or reference fs-splitroot-fix.patch for the earlier solution) are not strictly required (i.e. things can work if you are super-careful about empty mountpoint directories and proper `mountpoint` attribute values, and the system does not unexpectedly or by your mistake reboot while you are in mid-procedure, or if you use `legacy` mountpoints and fix up `/etc/vfstab` in each new BE), they do greatly increase the chances of successful and correct boot-ups in the general case with dynamically-used boot environments, shared datasets and occasional untimely reboots. Also, some networking initialization scripts (notably NWAM) do expect `/usr` and maybe even `/var` to be mounted before they run, and the existing `filesystem` methods (which would mount `/usr`) happen to depend on them, However, `physical:default` does run successfully (most of the time, missing just the `cut` command which can be replaced by a `ksh93` builtin implementation).

Specifying which `bootfs` children or shared datasets to mount

There are several ways to specify which datasets should be mounted as part of the dedicated or shared split-root hierarchy. In the context of descriptions below, the "`bootfs` children" are filesystem datasets contained within the root filesystem instance requested for current boot via GRUB (explicitly, or defaulting to the value of the ZFS pool's `bootfs` attribute).

- "Legacy" filesystem datasets with `mountpoint=legacy` which are explicitly specified in the `/etc/vfstab` file located inside this `bootfs`. This allows to pass mount-time options (such as the overlay mount, before it was enforced by the fixed `fs-*` scripts):

  ```
  rpool/ROOT/oi_151a8/usr      -       /usr            zfs     -       no      -
  rpool/SHARED/var/adm         -       /var/adm        zfs     -       yes     -
  ```

  A drawback of this method for `bootfs` children is that the file must be updated after each cloning or renaming of the boot environment to match the actual ZFS dataset full name for the particular `bootfs`.
- For bootfs children with specified `mountpoint` paths (and, for the new `fs-root-zfs` method, a `canmount` value **other** than "`off`"), mounting happens automatically: for `/usr` as a step in `filesystem/root` service, for others as a step in `filesystem/minimal` service.
  Typically the bootfs children specify `canmount=noauto`, because after BE cloning the `rpool` would provide multiple datasets with the same mountpoints, causing errors (conflicts) of automatic mounts during pool imports.
  NOTE: Specifying `canmount=off` for such datasets with un-fixed old service method implementations in place would log errors due to inability to `zfs mount` such datasets; however, for datasets other than `/usr`, the return codes are not checked, so this should not cause boot failures.
  - The `filesystem` methods can use `/etc/vfstab` to locate over a dozen paths for mounting (backed by any of the supported filesystem types), many of which are not used in the default installations. Those which might be used in practice with ZFS include `/usr`, `/var`, `/var/adm` and `/tmp`; these blocks in the method scipts also include logic to mount such child datasets of the current `bootfs` if they exist and a corresponding path was not explicitly specified in `/etc/vfstab`.
    Extensions added by me into the fixed scripts (earlier solution) or provided as the new `fs-root-zfs` method, allow to mount such paths (except `/usr` and `/var`) also from a number of other locations as "shared" datasets – if they were not found as children of the current `bootfs`.
- For possibly "shared" datasets, other than the explicitly specified short list (above), the legacy `filesystem` methods only offer the call to "`zfs mount -a`" from `filesystem/local` (way after the "single-user" milestone). This implies specified (non-"`legacy`") `mountpoint` paths and `canmount=on`; other datasets are not mounted automatically.
  Extensions provided as the new `fs-root-zfs` method allow to mount datasets with such attribute values from `$rpool/SHARED` (where the `$rpool` name is determined from the currently mounted root filesystem dataset). This ensures availability of active shared datasets as part of the split-root filesystem hierarchy early in boot. In particular, following the "auto-mounting" requirements allows to use datasets with a specified `mountpoint` path and `canmount=off` as "containers" for the shared datasets to inherit the parent container's path automatically (i.e. a non-mounting `/var` node).

Below you can find a screenshot with examples of the non-legacy datasets, both children of the root and shared ones. There is no example of a "legacy" dataset passed through `/etc/vfstab` because I can't contrive a rational case where that would be useful today 😅

Examples?

The examples below assume that your currently installed and configured OS resides in `rpool/ROOT/openindiana` and you want to relocate it into `rpool/ROOT/oi_151a8` with a hierarchy of compressed sub-datasets for system files (examples below use variables to allow easy upgrades of the procedure to different realities), and shared files like `logs` and `crash` dumps will reside in a hierarchy under `rpool/SHARED`.

This procedure can be done as soon as you have installed a fresh system with the default wizard settings from the LiveCD/LiveUSB – right from the Live environment (if it is networked so that you can get the patched method scripts), or at any time in the future (including a clone of your live system – though note that some changes may be "lost" from the new BE in the timeframe between replicating and actually rebooting; to avoid this you might want to boot into another BE or into the Live media and do the procedure on the "cold" main BE).

This can also be done during a migration of an older system to a new `rpool` for example, including a setup based on a clone of the Live media (see Advanced - Manual installation of OpenIndiana from LiveCD media), so that the hierarchical setup is done on your new `rpool` right from scratch.

Here is an illustration of what we are trying to achieve:

```
:; zfs list -o compression,compressratio,used,refer,canmount,mountpoint,name
COMPRESS  RATIO   USED  REFER  CANMOUNT  MOUNTPOINT               NAME
     lz4  2.32x   4.1G    44K        on  /rpool                   rpool
     lz4  2.32x  2.18G    31K       off  legacy                   rpool/ROOT
     lz4  2.68x   971M   164M    noauto  /a                       rpool/ROOT/oi_151a8
  gzip-9  1.00x    31K    31K    noauto  /opt                     rpool/ROOT/oi_151a8/opt
  gzip-9  2.93x   793M   792M    noauto  /usr                     rpool/ROOT/oi_151a8/usr
  gzip-9  1.00x    31K    31K    noauto  /usr/local               rpool/ROOT/oi_151a8/usr/local
  gzip-9  1.01x  12.0M  11.9M    noauto  /var                     rpool/ROOT/oi_151a8/var
     off  1.00x  2.96G  2.89G    noauto  /                        rpool/ROOT/openindiana
     lz4  2.05x  1.42G  1.42G    noauto  /                        rpool/ROOT/openindiana-lz4
  gzip-9  3.07x   355K    31K       off  legacy                   rpool/SHARED
  gzip-9  3.26x   324K    31K       off  /var                     rpool/SHARED/var
  gzip-9  10.68x 56.5K  56.5K        on  /var/adm                 rpool/SHARED/var/adm
  gzip-9  1.00x    31K    31K        on  /var/cores               rpool/SHARED/var/cores
  gzip-9  1.00x    31K    31K        on  /var/crash               rpool/SHARED/var/crash
  gzip-9  3.31x  49.5K  49.5K        on  /var/log                 rpool/SHARED/var/log
  gzip-9  1.00x    32K    32K        on  /var/mail                rpool/SHARED/var/mail
  gzip-9  1.00x    93K    31K       off  /var/spool               rpool/SHARED/var/spool
  gzip-9  1.00x    31K    31K        on  /var/spool/clientmqueue  rpool/SHARED/var/spool/clientmqueue
  gzip-9  1.00x    31K    31K        on  /var/spool/mqueue        rpool/SHARED/var/spool/mqueue
     off  1.00x  1.00G    16K         -  -                        rpool/dump
  gzip-9  8.54x   563K    32K        on  /export                  rpool/export
  gzip-9  8.83x   531K    32K        on  /export/home             rpool/export/home
  gzip-9  1.00x    31K    31K        on  /export/home/admin       rpool/export/home/admin
  gzip-9  9.47x   468K   468K        on  /jack                    rpool/export/home/jack
     off  1.00x  1.06G    16K         -  -                        rpool/swap
```

As you can see in the above example, the installed default OS (`rpool/ROOT/openindiana`) and its LZ4-compressed copy (`rpool/ROOT/openindiana-lz4`) are much larger than the split-root variant (`rpool/ROOT/oi_151a8`). This may be an important difference on some space- or I/O-constrained storage options. The shared filesystems include containers for logs, mailboxes, OS crash and process coredump images, and GZ mailqueues (rebooting into another BE does not mean you don't want those messages delivered, right?) – these can be restricted with quotas or relocated to other pools.

This particular system also spit off `/usr/local` in order to allow easy creation of clones delegated into local zones – so as to provide modifiable sets of unpackaged programs with little storage overhead. This is not a generally needed scenario 😅

How do I do this?

Now we're down to the dirty business ;)

Like in other low-level manuals, the user is expected to run as `root` (prepend `sudo` or `pfexec` as desired, if you run as a non-root), and the shell prompt for commands you enter is `:;` for ease of copy-pasting.

Envvars

Let's start by preparing some environment variables:

```
### The new rpool name and the hub datasets
:; RPOOL="rpool"
:; RPOOL_ROOT="$RPOOL/ROOT"
:; RPOOL_SHARED="$RPOOL/SHARED"

### The origin (or old) rpool, by default - same as new
:; OPOOL="$RPOOL"
:; OPOOL_ROOT="$OPOOL/ROOT"
:; OPOOL_SHARED="$OPOOL/SHARED"

### The rpool "altroot" mountpoint
:; RPOOLALT="/a"    ### For imports from live media
:; RPOOLALT=""      ### For the currently running system
:; OPOOLALT="$RPOOLALT"

### Boot Environment names, identical to rootfs dataset name component
:; BEOLD="openindiana"
:; BENEW="oi_151a8"
:; BEOLD_DS="$OPOOL_ROOT/$BEOLD"
:; BENEW_DS="$RPOOL_ROOT/$BENEW"

### Mountpoints, relative to the pool's altroot
:; BEOLD_MPT="/"
:; BENEW_MPT="/a"
### Mountpoints, "absolute" in current OS
:; BEOLD_MNT="$OPOOLALT$BEOLD_MPT"
:; BENEW_MNT="$RPOOLALT$BENEW_MPT"

### For the case of remote cloning, one or both of these can be defined
### like "$USER@HOST:" for ssh/rsh tunneling or even "rsync://$USER@HOST:"
:; SRC=""
:; TGT=""
### For the case of remote cloning, this can be defined to use alternate
### port for SSH, or even use old RSH if circumstances require and permit
#:; RSH="--rsh='/bin/ssh -p 2222'"
#:; RSH="--rsh=/bin/rsh"
:; RSH=""

### Prefer Sun commands over GNU "equivalents"
:; PATH="/sbin:/bin:/usr/sbin:$PATH"
:; export PATH
```

Note that these settings can be defined differently on the source and target hosts, if you clone the installation onto another machine, i.e. from a production system onto a new one (or a VM) currently booted with LiveCD which has the newly created `rpool` alt-mounted under `/a`.

New rpool

Optional step – creation of the new `rpool`.

If you are migrating an installation to a new root pool, be it change of devices or cloning of an existing installation to a remote machine, you can take advantage of the new layout right away. If your devices have large native sectors or pages and would benefit from aligned access, then first you should settle on the partitioning and slicing layout which would ensure alignment of the `rpool` slice. This is a separate subject, see Advanced - Creating aligned rpool partitions.

Then go on to Advanced - Creating an rpool manually and return here when done 😎

Unfortunately, you can not use the existing or newly (and manually) created `rpool` for the initial installation of OpenIndiana with its official installer (issue number #4353), although you can precreate the partitioning you deem needed for your hardware (i.e. to ensure alignment of the future `s0` slice which the installer would recreate, with the hardware sectors). You might try to follow the procedure described in Advanced - Manual installation of OpenIndiana from LiveCD media to populate a dataset – or a split-root hierarchy – with a new installation, but this is currently even more experimental than the split-root procedure.

New hierarchies

So, here the fun begins. One way or another, I assume that you have a (target) `rpool` created and initialized with some general options and datasets you deemed necessary. This includes the case of splitting the installation within one machine and one `rpool`, where you just continue to use the other datasets (such as `swap`, `dump` and the default admin-home tree under `/export`).

Base root filesystem dataset

Create the base rootfs, note that its compression should match GRUB's support:

```
### If this has not been done yet...
:; zpool import -N -f -R "$RPOOLALT" "$RPOOL"
:; zfs create -o mountpoint=legacy -o canmount=off "$RPOOL_ROOT"

### On single-disk systems you might want to use more copies of data
### to protect the OS installation from single-sector errors
#:; zfs set copies=2 "$RPOOL_ROOT"

### Ensure that direct child datasets of rpool/ROOT will be created
### with compression (or lack thereof) compatible with GRUB
:; zfs set compression=off "$RPOOL_ROOT"
:; zfs set compression=lz4 "$RPOOL_ROOT"

##########################
### New hierarchy root ###
##########################
:; zfs create -o mountpoint="$BENEW_MPT" -o canmount=noauto "$BENEW_DS"
:; zfs set org.openindiana.caiman:install=busy "$BENEW_DS"

### This is where you can customize stuff for the root-BE dataset only; except compression.
### BE UUID is random, individual for each BE; maintained by beadm in live systems
:; zfs set org.opensolaris.libbe:uuid=c2c6c968-9866-c662-aac1-86c6cc77c2c9 "$BENEW_DS"
:; zfs mount -O "$BENEW_DS"

:; df -k "$BENEW_MNT"
### This should display the newly created filesystem, something like this:
Filesystem           1024-blocks        Used   Available Capacity  Mounted on
rpool/ROOT/oi_151a8      8193024          31     3738900     1%    /a
```

If any unexpected errors were returned or the filesystem was not mounted – deal with it (find the causes, fix, redo the above steps).

Child filesystems of rootfs

Now that you have the new root filesystem, prepare it for children, using your selection of sub-datasets. These will be individual to each OS installation, cloned and updated along with their BE. Generally this includes all locations with files delivered by "system" packages, which are likely to be updated in the future. Also included below is `/opt/local` as the path used by Joyent PKGSRC releases usable on most illumos distributions and likely to consume lots of space.

To follow the example settings defined above:

```
:; cd "$BENEW_MNT" && for D in \
    usr var opt usr/local var/pkg opt/local \
   ; do mkdir "$D" && /bin/chmod S+ci "$D" && \
        zfs create -o canmount=noauto -o compression=gzip-9 "$BENEW_DS/$D" && \
        zfs mount -O "$BENEW_DS/$D" || break; \
   done

:; /bin/df -k | grep " $BENEW_MNT"
### Example listing:
rpool/ROOT/oi_151a8           8193024     34     3738737     1%    /a
rpool/ROOT/oi_151a8/usr       8193024     32     3738737     1%    /a/usr
rpool/ROOT/oi_151a8/var       8193024     31     3738737     1%    /a/var
rpool/ROOT/oi_151a8/opt       8193024     31     3738737     1%    /a/opt
rpool/ROOT/oi_151a8/usr/local 8193024     31     3738737     1%    /a/usr/local
rpool/ROOT/oi_151a8/var/pkg   8193024     31     3738737     1%    /a/var/pkg
```

In the example above, mountpoint directories are protected from being written into by being made immutable. Note that this requires the Solaris (not GNU) `chmod`, and that this does not work in Solaris 10 (if you backport the procedure – which mostly works). Also note that `/var/pkg` is relevant for IPS-based distributions like OpenIndiana, and you might want to omit it when applying the procedure to some other OS in the Solaris family.

Also note that at this point the sub-datasets inherit the `/a` prefix in their mountpoints, and will fail to mount "as is" with the currently default scripts (`fs-root` and `fs-minimal`), unless you later unmount this tree and change the rootfs to use `mountpoint=/`.

Shared filesystems

Next we prepare the shared filesystems. To follow the example above:

```
:; zfs create -o mountpoint=legacy -o canmount=off -o compression=gzip-9 "$RPOOL_SHARED"
:; zfs create -o mountpoint="$BENEW_MPT"/var -o canmount=off "$RPOOL_SHARED/var"
:; zfs create -o canmount=off "$RPOOL_SHARED/var/spool"
```

This prepares the "container" datasets with predefined compression and mountpoint attributes; you can choose to define other attributes (such as `copies`) at any level as well. These particular datasets are completely not mountable so as to not conflict with OS-provided equivalents, they are only used to contain other (mountable) datasets and influence their mountpoints by inheritance, as well as set common quotas and/or reservations. Also note that currently the shared `var` components are not mounted into the `rpool` altroot, but are offset by `"$BENEW_MPT"` prefix. This will be fixed later, after data migration.

Now we can populate this location with applied datasets. Continuing with the above example of shared parts of the namespace under `/var`, we can do this:

```
:; cd "$BENEW_MPT"/var && \
   for D in adm cores crash log mail spool/clientmqueue spool/mqueue ; do \
     mkdir -p "$D" && /bin/chmod S+ci "$D"; \
     zfs create -o canmount=on "$RPOOL_SHARED/var/$D"; \
   done

### Verify success of the previous operation(s) before proceeding
:; /bin/df -k | grep " $BENEW_MNT"

:; for D in cores crash ; do \
     zfs set quota=5G "$RPOOL_SHARED/var/$D" ; \
     zfs set com.sun:auto-snapshot=false "$RPOOL_SHARED/var/$D" ; \
   done

:; for D in spool/clientmqueue spool/mqueue ; do \
     zfs set quota=2G "$RPOOL_SHARED/var/$D" ; done
```

NOTE: Don't blindly split off `/var/tmp` like this, at least not unless you are ready to test this as much as you can. It was earlier known to fail, though it may work better now dependent on the distribution features, SMF dependency order and other such variables. It actually works on my system, but I am not ready to "guarantee" this for others. Since the problem was that in legacy setups some services wrote into this directory before the dedicated dataset was mounted (thus either blocking the mount, or losing access to written files), now there should be no problem since mounting is done before other services as enforced by SMF dependencies – unless you store your `/var/tmp` on a non-root pool and then that pool import fails at boot. If you do find that the temporary directories over dedicated ZFS datasets (whether as `/var/tmp` or in some differently-named paths perhaps stored on a separate user-data pool) work well for you, consider adding some security and performance options into the mix, for example:

```
:; mkdir "$BENEW_MPT"/var/tmp && /bin/chmod S+ci "$BENEW_MPT"/var/tmp
:; zfs create -o canmount=on -o setuid=off -o devices=off -o sync=disabled -o atime=off "$RPOOL_SHARED"/var/tmp
:; chmod 1777 "$RPOOL_SHARED"/var/tmp
### Set quota or don't set it, as you see fit
```

The example above creates the immutable mountpoint directories in the rootfs hierarchy's version of `/var`, then creates and mounts the datasets into the new hierarchy's tree. Afterwards some typically acceptable quotas (YMMV) are set to protect the root file system from overfilling with garbage. Also, `zfs /auto-snapshot` service is forbidden to make autosnaps of the common space-hogs `/var/cores` and `/var/crash`, so that deletion of files from there to free up `rpool` can proceed unhindered.

Migrating the data

Now that the hierarchies have been created and mounted, we can fill them with the copy of an installation.

This chapter generally assumes that the source and target data may be located on different systems connected by a network, and appropriate clients and servers (SSH or RSH) are set up and working so that you can initiate the connection from one host to another. The case of local-system copying is a degenerate case of multi-system, with `SRC` and `TGT` components and the `RSH` flag all empty 😇

First of all, you need to provide the original filesystem image to copy. While a mounted alternate BE would suffice, the running filesystem image "as is" usually contains `libc.so` and possibly other mounts, which makes it a poor choice for the role of clone's origin. You have a number of options, however, such as diving into snapshots, creating and mounting a full BE clone, or `lofs`-mounting the current root to snatch the actual filesystem data (this case being especially useful back in the days of migration of Solaris 10 roots from UFS to ZFS).

The procedure may vary, depending on your original root filesystem layout – whether it is monolithic or contains a separate `/var`, for example.

All of the examples use `rsync` – it does the job well, except maybe for lack of support for copying ZFS/NFSv4 ACLs until (allegedly) rsync-3.0.10, which is not relevant for a default installation. Flags used below include:

- `-x` – single-filesystem traversal (only copy objects from source filesystem, don't dive into sub-mounts – you should manually verify and ensure that mountpoints like `/tmp` or `/proc` should ultimately exist on targets);
- `-avPHK` – typical recursive replication with respect for soft- and hard-links and verbose reports;

- `-z` – if you copy over a slow network link, this would help by applying compression to the transferred data (not included in examples below);
- The `rsync` program is executed in a loop, so if something breaks (i.e. out of memory on LiveCD environment) it would pick up and proceed until success.

You also have an option to initiate the `rsync` process from either the source system (where the original data tree resides) or from the new system (on which the split-root structure is formed and written). The choice depends on networking (routing, firewalls, etc.) among other things, either way is possible and this is in essence a feasible step in the way to clone pre-installed systems. Single-system copying is just an edge case here, where origin and target are the same and networking may be avoided (the `$RSH` variable is empty).

BE cloning

This example is for systems with `beadm` applicable to the selected source dataset (i.e. the source BE resides in the currently active origin `rpool`).

Prepare the source file tree; basically this allows to use a clone of the current root into which no run-time additions would land, and without user datasets and other overlays mounted inside:

```
### On origin system - optionally clone the current BE
### (if it is the source) and mount the selected BE
:; beadm create -e "$BEOLD" "$BEOLD-split"
:; beadm mount "$BEOLD-split" "$BEOLD_MNT.split" && \
   /bin/df -k | grep " $BEOLD_MNT.split"

### Verify that all needed filesystems are indeed mounted, and
### if any extras are there (like zone roots) - unmount them or
### define exclusions in the rsync command. Note "-x" is off.
```

Run on source or single system:

```
### Initiate copying from the origin system (target is the SSH/RSH/RSYNC server or local system)
:; cd "$BEOLD_MNT.split/" && while ! eval rsync -avPHK $RSH ./ "$TGT$BENEW_MNT/" ; do sleep 1; done
```

... **OR** run on target system:

```
### Initiate copying from the target system (origin is the SSH/RSH/RSYNC server)
:; cd "$BENEW_MNT/" && while ! eval rsync -avPHK $RSH "$SRC$BEOLD_MNT.split/" ./ ; do sleep 1; done
```

Snapshot-diving

In this example you can use ZFS snapshots as the read-only sources for `rsync` copy process. One substantial difference is that for any child datasets of the origin system (and note that this refers to the origin – which may indeed have no child datasets, or might have a separate `var` child) you have to reiterate separate `rsync` runs.

Prepare the source:

```
### On origin system
:; zfs snapshot -r "$BEOLD_DS@$BENEW-split"
```

Run on source or single system:

```
### Initiate copying from the origin system (target is the SSH/RSH/RSYNC server or local system)
:; cd "$BEOLD_MNT/.zfs/snapshot/$BENEW-split" && \
   while ! eval rsync -avPHK $RSH ./ "$TGT$BENEW_MNT/" ; do sleep 1; done


### Rinse and repeat for child datasets of origin, like /var, if any, i.e.:
:; for D in var ; do \
   cd "$BEOLD_MNT/$D/.zfs/snapshot/$BENEW-split" && \
    while ! eval rsync -avPHK $RSH ./ "$TGT$BENEW_MNT/$D/"; do sleep 1; done; \
   done
```

... **OR** run on target system:

```
### Initiate copying from the target system (origin is the SSH/RSH/RSYNC server)
:; cd "$BENEW_MNT/" && \
   while ! eval rsync -avPHK $RSH "$SRC$BEOLD_MNT/.zfs/snapshot/$BENEW-split/" ./; do sleep 1; done

### Rinse and repeat for child datasets of origin, like /var, if any, i.e.:
:; cd "$BENEW_MNT/" && for D in var ; do \
    while ! eval rsync -avPHK $RSH "$SRC$BEOLD_MNT/$D/.zfs/snapshot/$BENEW-split/" "./$D/"; do sleep 1; done \
   done
```

`lofs`-mounting

This allows to use `lofs` as a means of producing an unmodified source filesystem without interference of overlay-mounts. Historically this is the approach which helped migrate from UFS roots onto ZFS.

Prepare the source:

```
### On origin system - lofs-mount the active root filesystem
:; mkdir /mnt/root
:; mount -F lofs -o nosub "$BEOLD_MNT" /mnt/root
```

Run on source or single system:

```
### Initiate copying from the origin system (target is the SSH/RSH/RSYNC server or local system)
:; while ! eval rsync -xavPHK $RSH --exclude=/mnt/root /mnt/root/ "$TGT$BENEW_MNT/"; do sleep 1; done

### For optional subsequent datasets/filesystems (i.e. var), on origin
:; for D in var; do \
   mount -F lofs -o nosub "$BEOLD_MNT/$D" "/mnt/root/$D" && \
    while ! eval rsync -xavPHK $RSH --exclude=/mnt/root "/mnt/root/$D/" "$TGT$BENEW_MNT/$D/"; do sleep 1; done
; \
   done
```

... **OR** run on target system:

```
### Initiate copying from the target system (origin is the SSH/RSH/RSYNC server)
:; while ! eval rsync -xavPHK $RSH --exclude=/mnt/root "$SRC/mnt/root/" "$BENEW_MNT/"; do sleep 1; done
```

Tuning the split-root OS image

Now that you are done replicating the source filesystem image, don't rush to boot it. There are some more customizations to make which ensure that it would actually work.

Snapshot first

Just in case you mess up in the steps below, have something to roll back to:

```
:; zfs snapshot -r "$RPOOL_SHARED@postsplit-01"
:; zfs snapshot -r "$BENEW_DS@postsplit-01"
```

Copying /sbin/sh and others

First of all, if you have split out the `/usr` filesystem, you should make sure that `/sbin/sh` is a valid working copy of the `ksh93` shell (or whichever is default for your system, in case of applying these instructions to another distribution). Some other programs, such as `bash` or `ls`, may also be copied from `/usr/bin` into `/sbin` (paths relative to your new rootfs hierarchy) at your convenience during repairs without a mounted `/usr`, but are not strictly required for OS operation.

```
### Use chroot into the new BE so that ldd would be limited by local
### namespace, including our subsequent changes to it in /lib
:; chroot "$BENEW_MNT" "/usr/bin/bash"
:; cd "/sbin" && ls -la sh

lrwxrwxrwx   1 root     root          20 Jul 21 15:41 sh -> ../usr/bin/i86/ksh93
### So here it is - a symlink out of the root filesystem into /usr filesystem

### Part one: copy the binary file
:; cp -pf ../usr/bin/i86/ksh93 .
:; mv sh sh.orig
:; ln -s ksh93 sh

### Part two: copy its dependency libraries
:; for F in /sbin/ksh93; do ldd "$F" | awk '{print $NF}' | egrep '^/usr/lib/' | sed 's,^/usr/lib/,,' | \
   while read L; do echo "/usr/lib/$L" && cp -pf "../usr/lib/$L" "../lib/$L"; done; done

/usr/lib/libshell.so.1
/usr/lib/libcmd.so.1
/usr/lib/libdll.so.1
/usr/lib/libast.so.1
/usr/lib/libsum.so.1

### Optionally copy other useful programs - not required and may add headache in later OS upgrades
#:; cp ../usr/bin/bash .

### Exit the chroot
:; exit

### Snapshot again
:; zfs snapshot -r "$RPOOL_SHARED@postsplit-02"
:; zfs snapshot -r "$BENEW_DS@postsplit-02"
```

Patching the scripts

Make backups of originals and get the files attached to this article. Examples below use `wget` for internet access, but a non-networked system might require other means (like a USB stick transfer from another, networked, computer).

For the oi_151a8 release and several releases before it, the system-provided scripts did not change, so the full scripts can be the easier choice to download: fs-root-zfs, fs-root and fs-minimal. As described above, the `fs-root-zfs` script includes all the logic needed to detect and mount the local ZFS-based root filesystem hierarchy (and skips any non-ZFS filesystems and mountpoints under them), and the existing method scripts are just slightly fixed to expect that the paths they try to manage may have already been mounted. Also, unlike the earlier existing scripts, the `fs-root-zfs` script explicitly mounts the shared datasets (`$rpool/SHARED`) early in the system initialization to ensure the complete root filesystem hierarchy to other methods, such as `network` initialization scripts.

For other releases and distributions it may be worthwhile to get the patches as fs-root-zfs.patch and apply them.

To introduce the new service `svc:/system/filesystem/root-zfs:default` as a dependency for SMF services whose methods rely on a proper filesystem early in boot, you'll also need the service manifest fs-root-zfs.xml. I hope that ultimately this logic will make it upstream and patching your installation will no longer be necessary 😎

```
### Make backups
:; cd "$BENEW_MNT/lib/svc/method" && for F in fs-root-zfs fs-root fs-minimal; do
    N="$F.orig-$BEOLD"; [ -f "$F" -a -f "$N" ] || cp -pf "$F" "$N"; done; ls -la fs-*
### Verify success of the previous operation(s) before proceeding

### REPLACEMENT SCRIPTS
### For oi_151a8 (and in fact many other releases) it may be suitable to replace the scripts
### For other releases, including those newer than this post (Nov 2013) verify contents first
:; for F in fs-root-zfs fs-root fs-minimal; do \
    rm -f "$F.splitroot"; wget -O "$F.splitroot" \
      "http://wiki.openindiana.org/download/attachments/27230229/$F" && \
    cat "$F.splitroot" > "$F"; chmod +x "$F"; done; ls -la fs-*

### OR - PATCH OLD SCRIPTS
### In case of patch-files - try to apply them. This is probably more portable and future-proof
### (i.e. if your distribution has deviated from the specific script versions used in the example
### above, and/or some different patches were applied).
:; wget -O "fs-root-zfs.patch" \
     "http://wiki.openindiana.org/download/attachments/27230229/fs-root-zfs.patch" && \
   gpatch --dry-run -p4 < "fs-root-zfs.patch" && \
   gpatch -p4 < "fs-root-zfs.patch" && chmod +x "fs-root-zfs"; ls -la fs-*

### Also get the manifest for the new service
:; wget -O "$BENEW_MNT/lib/svc/manifest/system/filesystem/fs-root-zfs.xml" \
   http://wiki.openindiana.org/download/attachments/27230229/fs-root-zfs.xml

### Install the service manifest into the new BE right away
echo "repository $BENEW_MNT/etc/svc/repository.db
import $BENEW_MNT/lib/svc/manifest/system/filesystem/fs-root-zfs.xml
exit" | svccfg
```

The scripts include an ability to log all the decisions done regarding mounting or not mounting specific datasets, fixing mountpoints, etc. which go to the console (physical or serial, per your setup and kernel boot-time parameters), as well as into SMF (check `/var/svc/log/system-filesystem-root-zfs:default.log` or `/etc/svc/volatile/system-filesystem-root-zfs:default.log` for copies of the relevant entries). To enable such logging just go:

```
:; touch "$BENEW_MNT/.debug_mnt"
```

Fixing /etc/vfstab

Verify that `$BENEW_MNT/etc/vfstab` does not reference filesystems which you expect to mount automatically – such as the shared filesystems or non-legacy children of the rootfs du-jour. A reference to `rpool/swap` is okay:

```
:; cat "$BENEW_MNT/etc/vfstab" | egrep "$RPOOL|$OPOOL"
```

Fixing ssh

The `svs:/network/ssh:default` SMF service for the Secure SHell server normally depends on quite an advanced system startup state – with all user filesystems mounted and `autofs` working. For us admins `ssh` is a remote management tool which should be available as early as possible, especially for cases when the system refuses to mount some filesystems and so start some required dependency services.

For this administrative access to work in the face of failed `zfs mount -a` (frequent troublemaker), we'd replace the dependency from `filesystem/local` to `filesystem/usr` which ensures that the SSH software is already accessible at least for admins:

```
:; echo "repository $BENEW_MNT/etc/svc/repository.db
select ssh
setprop fs-local/entities = fmri: svc:/system/filesystem/usr
select default
refresh
exit" | svccfg
```

- To fix the running system you wouldn't need the `repository` line.

Note that while the block above would work for a copy of an installed and fully configured system, if you apply this article's tweaks to a freshly installed system right from the LiveCD environment, then the installed image does not have the `ssh` service in the repository database, yet. Instead, you can modify the XML manifest file:

```
:; cd "$BENEW_MNT/lib/svc/manifest/network" && \
   [ ! -s "ssh.xml.orig" ] && cp -pf ssh.xml ssh.xml.orig; \
   [ -s "ssh.xml.orig" ] && cat ssh.xml.orig | \
     sed 's,svc:/system/filesystem/local,svc:/system/filesystem/usr,g' > ssh.xml; \
   ls -la ssh.xml*
```

Fixing coreadm

This is not strictly related to split-roots, but since we set up the `/var/cores` dataset here – this is still a good place to advise about its nice system-wide setup. The configuration below enables the global zone to capture all process coredumps, including those which happen in the local zones, and place them into the common location. This way admins can quickly review if anything went wrong recently (until this location gets overwhelmed with data). Create the `/etc/coreadm.conf` file and it will be sucked in when the `coreadm` service next starts up in the new BE:

```
:; echo '#
# coreadm.conf
#
# Parameters for system core file configuration.
# Do NOT edit this file by hand -- use coreadm(1) instead.
#
COREADM_GLOB_PATTERN=/var/cores/%f.%z.%n.%t.%p.core
COREADM_GLOB_CONTENT=default
COREADM_INIT_PATTERN=core
COREADM_INIT_CONTENT=default
COREADM_GLOB_ENABLED=yes
COREADM_PROC_ENABLED=no
COREADM_GLOB_SETID_ENABLED=yes
COREADM_PROC_SETID_ENABLED=no
COREADM_GLOB_LOG_ENABLED=yes
' > "$BENEW_MNT/etc/coreadm.conf"
```

Symlink /etc/grub.conf

Okay, I admit this is a "linuxism" – but a convenient one:

```
:; ln -s "../$RPOOL/boot/grub/menu.lst" "$BENEW_MNT/etc/grub.conf"
```

It also makes sense to go over `$RPOOLALT$RPOOL/boot/grub/menu.lst` at this time, and clone the GRUB menu entry definitions to reference the new BE (don't forget to set the `default` entry number sooner or later, too). Or you can make an copy of a menu entry without the `bootfs` line which would load the BE referenced by the `rpool`'s `bootfs` attribute, like this:

```
title default_bootfs syscon
findroot (pool_rpool,0,a)
kernel$ /platform/i86pc/kernel/$ISADIR/unix -B $ZFS-BOOTFS
module$ /platform/i86pc/$ISADIR/boot_archive

title default_bootfs sercon
findroot (pool_rpool,0,a)
kernel$ /platform/i86pc/kernel/$ISADIR/unix -B $ZFS-BOOTFS,console=ttya
module$ /platform/i86pc/$ISADIR/boot_archive
```

Running bootadm

This is a simple one – just in case, run `bootadm` on the new rootfs hierarchy:

```
:; touch "$BENEW_MNT/reconfigure"
:; bootadm update-archive -R "$BENEW_MNT"
```

Unmounting and cleaning up (reset mountpoints)

This is a pretty important step in making sure that datasets are mountable as expected on a subsequent boot:

```
### Don't block the mountpoints we are trying to release
:; cd /

### On a Live-Media system you can just unmount everything:
:; zfs umount -a

### VARIANT A:
### On a working system which did not use rpool/SHARED yet, you should
### go over all the datasets of the new hierarchy:
:; /bin/df -k | egrep "^($BENEW_DS|$RPOOL_SHARED)" | awk '{print $NF}' | sort -r | \
   while read M; do echo "=== $M"; umount -f "$M"; done

### VARIANT B:
### On a working system which did use rpool/SHARED (i.e. you are rebalancing a
### split-root configuration), you should not unmount the used shared datasets:
:; /bin/df -k | egrep "^($BENEW_DS)" | awk '{print $NF}' | sort -r | \
   while read M; do echo "=== $M"; umount -f "$M"; done

### Fix the new roots AFTER the new hierarchy datasets are unmounted successfully
:; zfs set mountpoint="/" "$BENEW_DS"
:; zfs set mountpoint="/var" "$RPOOL_SHARED/var"

:; zfs set org.openindiana.caiman:install=ready "$BENEW_DS"
```

Make a snapshot, again

We've done this before, can do it again:

```
:; zfs snapshot -r "$RPOOL_SHARED@postsplit-03"
:; zfs snapshot -r "$BENEW_DS@postsplit-03"
```

bootfs and grub

This sets up the default root filesystem for booting (if not specified explicitly in GRUB menu file):

```
### Set up the GRUB bootloader (if the rpool devices had changed)
:; zpool status "$RPOOL" | grep ONLINE | awk '{print $1}' | egrep '^c.+s.$' | \
   while read SLICE; do echo "=== $SLICE"; \
     /sbin/installgrub /boot/grub/stage1 /boot/grub/stage2 "/dev/rdsk/$SLICE"; \
   done

:; zpool set failmode=continue "$RPOOL"
:; zpool set bootfs="$BENEW_DS" "$RPOOL"
```

Verify importability of the new rpool

If you are doing this all in a LiveCD environment, it makes sense to verify that there are no conflicts in mountpoints. Note that the LiveCD also places a hold on the `dump` and `swap` volumes (at least, if it has just created the installation), and these resources must be freed to actually export the `rpool`:

```
:; zpool export $RPOOL
pool busy

:; dumpadm -d none

:; swap -l
swapfile             dev    swaplo   blocks     free
/dev/zvol/dsk/rpool/swap 96,2         8  4188152  4188152
:; swap -d /dev/zvol/dsk/$RPOOL/swap

:; zpool export $RPOOL
:; zpool import -N -R /a -f $RPOOL && zfs mount -O $BENEW_DS && \
   zfs mount -O $BENEW_DS/var && zfs mount -a
```

Upgrades

As discussed earlier, this hierarchy also requires (or benefits from) a bit of special procedure to upgrade the installation. While it is customary to have the `pkg` command create all needed BE datasets and proceed with the upgrade in the newly cloned BE, we'd need to reenable compression and maybe some other attributes first.

Don't forget to verify (or just redo) the copying of `/sbin/sh` and its related libraries, especially if they have changed, revise the patched filesystem method scripts and other customizations discussed above (as well as others you do on your systems).

beadm clone fixup and package upgrading

In order for your OS updates to enjoy the disk-space savings, the compression attributes should be appropriately applied to the cloned datasets. Unfortunately, current `beadm clone` does not take care of that. The most simple approach is to create and fix the new BE first, then use it as a target for the package upgrades.

The three mini-chapters below go from the most-automated to the most-manual description of essentially the same procedure (inverse order of evolution as examples from this page got scripted). Generally the first snippet should be used in practice, while the others are more of interest for further development of audit of the procedure. The concluding mini-chapter covers destruction of such BEs as they become un-needed, because it also becomes slightly more complicated.

The environment variables involved in the procedures or scripts below are similar to ones used in the manual above, but there are less of them set, since we are playing within one `rpool` (no networked copies are implied).

Script-automated rootfs/BE cloning and upgrading

The most automated help can be received from beadm-upgrade.sh which automates BE cloning with beadm-clone.sh (described below) and then issues IPS and PKGSRC package updates in the new BE:

- For a fully-automatic job, download the scripts:

```
:; wget -O /root/beadm-clone.sh "https://github.com/jimklimov/illumos-splitroot-scripts/raw/master/bin
/beadm-clone.sh" && \
   chmod +x /root/beadm-clone.sh
:; wget -O /root/beadm-upgrade.sh "https://raw.githubusercontent.com/jimklimov/illumos-splitroot-scripts
/master/bin/beadm-upgrade.sh" && \
   chmod +x /root/beadm-upgrade.sh
```

- Run the upgrader (optionally pre-set and export the envvars described all around this text); the script prints the variables it is going to use and pauses before proceeding (press ENTER to go on):

```
:; /root/beadm-upgrade.sh
```

  If all was ok – activate (copy-paste the BE name from last lines of output of `beadm-upgrade.sh`) and gracefully reboot:

```
:; beadm activate "$BENEW" && \
   init 6
```

Script-automated rootfs/BE cloning

The attached script beadm-clone.sh (Git master: beadm-clone.sh) automates most of the logic described in the text below, and uses the same environment variables. You can execute it as a shell script as well as just "source" it into your current (root) shell – but beware that it can `exit` upon errors; execution requires that you "export" the envvars you need, while "sourcing" would set whatever remains to guesswork in the shell context which remains current and would not redefine them in subsequent runs.

As a point-and-shoot solution that requires no pre-configuration, it can clone the currently running BE suffixing it with a timestamp.

In a second layer of usability it may suffice that you only set `BEOLD` and/or `BENEW` and it should guess the rest.

For just the BE cloning with the script do:

- Download the script:

```
:; wget -O /root/beadm-clone.sh "https://github.com/jimklimov/illumos-splitroot-scripts/raw/master/bin
/beadm-clone.sh" && \
   chmod +x /root/beadm-clone.sh
```

- Source it into the current shell so it sets all the variables as it goes (by default it will propose a new BE name based on the first token of the current BE before a separator such as the dash character, and suffix it with current timestamp); the script prints the variables it is going to use and pauses before proceeding (press ENTER to go on):

```
:; . /root/beadm-clone.sh
```

  Alternately, don't source but rather run the script and copy-paste the reported variable values into your shell.
- When the script is done cloning and has reported no errors, copy-paste the suggestions from the end of its output, i.e.:

```
:; pkg -R "$BENEW_MNT" image-update --deny-new-be --no-backup-be && \
   touch "$BENEW_MNT/reconfigure" && \
   bootadm update-archive -R "$BENEW_MNT" && \
   beadm umount "$BENEW"

:; TS="`date -u "+%Y%m%dZ%H%M%S"`" && \
   zfs snapshot -r "$RPOOL_SHARED@postupgrade-$TS" &&\
   zfs snapshot -r "$BENEW_DS@postupgrade-$TS"
```

- If all was ok – activate and gracefully reboot:

```
:; beadm activate "$BENEW" && \
   init 6
```

  Hopefully, everything goes up nicely and quickly, and a `` `df -k /` `` would show the new root dataset 👒

Manual rootfs/BE cloning

If explicit control over the procedure is desired (or if it is problematic to download the script and you'd rather copy-paste code), you can define everything as detailed below:

```
:; RPOOL="rpool"
:; RPOOL_ROOT="$RPOOL/ROOT"
:; RPOOL_SHARED="$RPOOL/SHARED"

:; RPOOLALT=""      ### For the currently running system
:; BEOLD="oi_151a8"
:; BENEW="oi_151a9"

:; BEOLD_MPT="/"
:; BENEW_MPT="/a"

:; BEOLD_DS="$RPOOL_ROOT/$BEOLD"
:; BENEW_DS="$RPOOL_ROOT/$BENEW"
:; BEOLD_MNT="$RPOOLALT$BEOLD_MPT"
:; BENEW_MNT="$RPOOLALT$BENEW_MPT"
```

So, we clone the current BE (or the one from which we want to upgrade):

```
:; beadm create -e "$BEOLD" "$BENEW" && \
   beadm mount "$BENEW" "$BENEW_MNT"

:; df -k | tail -20
```

This should create snapshots and clones of the rootfs dataset and its children – but alas, the process (currently) loses most of the ZFS locally defined attributes, such as `compression`.

```
:; EXCLUDE_ATTRS='org.opensolaris.libbe:uuid|canmount|mountpoint'

### If you use this procedure for something else (i.e. cloning zones)
### you might want to not exclude any attributes. Then use this value:
#:; EXCLUDE_ATTRS='^$'

### Verify custom attributes other than those known to be set by beadm
:; zfs list -H -o name -r "$BEOLD_DS" | while read Z; do \
    S="`echo "$Z" | sed "s,^$BEOLD_DS,,"`"; echo "=== '$S'"; \
    zfs get all "$BEOLD_DS$S" | egrep ' (local|received)'; \
    echo ""; zfs get all "$BENEW_DS$S" | egrep ' (local|received)'; \
   done | egrep -v "$EXCLUDE_ATTRS"
### If any custom attributes pop up in the newly created BE, these should
### probably also be excluded from replication (EXCLUDE_ATTRS) before proceeding

:; zfs list -H -o name -r "$BEOLD_DS" | while read Z; do \
    S="`echo "$Z" | sed "s,^$BEOLD_DS,,"`"; echo "=== '$S'"; \
    zfs get all "$BEOLD_DS$S" | egrep ' (local|received)' | \
    egrep -v "$EXCLUDE_ATTRS" | while read _D A V _T; do \
      echo "$A=$V"; zfs set "$A=$V" "$BENEW_DS$S"; \
   done; done

=== ''
compression=lz4
=== '/opt'
compression=gzip-9
=== '/usr'
compression=gzip-9
=== '/usr/local'
compression=gzip-9
=== '/var'
compression=gzip-9
```

This took care of proper compressions, and maybe other customizations.

Now you can update the new BE and retain the savings thanks to your chosen compression rate, and it should go along these lines:

```
:; pkg -R "$BENEW_MNT" image-update --deny-new-be --no-backup-be

:; touch "$BENEW_MNT/reconfigure"
:; bootadm update-archive -R "$BENEW_MNT"

:; beadm umount "$BENEW"

:; zfs snapshot -r "$RPOOL_SHARED@postupgrade-01"
:; zfs snapshot -r "$BENEW_DS@postupgrade-01"
```

A successful update should result in activation of the new BE. There is a couple of ways:

- The official method is `beadm activate` which updates the GRUB menu and possibly does other housekeeping; when it is done, you should gracefully⚠️ reboot (when time comes):

  ```
  :; beadm activate "$BENEW"
  Activated successfully

  :; init 6
  ```

  In particular, the updated GRUB menu entries allow you to easily fall back and boot an older BE, without hacking at the console to enter the bootfs you want as active.
- Still, if you are oldschool and rely on default `bootfs` (referenced from GRUB menu as the default choice), just update it and reboot (when time comes), and hope that this suffices 🤞 in the release du-jour:

  ```
  :; zpool set bootfs="$BENEW_DS" "$RPOOL"
  :; init 6
  ```

Good luck!

Removing a BE prepared by the above procedure

It is perfectly possible that you don't get everything the way you wanted on the first attempt, and would like to retry. An update attempt might not find any packages to update and the new BE is thus useless.

In these or any similar cases you should use the "`-s`" flag to `beadm destroy` because after the procedure above (and/or after some life-time on a system with Time-Slider or equivalent technology), the new BE contains several snapshots which block "normal" removal:

```
:; beadm destroy -s "$BENEW"
```

If an update was successful and well-tried in practice, so you no longer need an old BE... do be careful in its removal:

> **WARNING:** Before doing recursive ZFS removals (which is what `beadm destroy -s` should be doing), remember that this action can impact all child datasets that are not blocked by being mounted and files open, by running `zfs send` sessions or by a `zfs hold`, for example. Beside the snapshots and sub-datasets in the hierarchy which you do intend to remove, such "children" may include ZFS clones such as newer BE's.
>
> There is a difference between `zfs destroy -r` and `zfs destroy -R` commands that lies essentially just in this aspect – whether clones are also removed.
>
> Do verify first what your particular OS distribution and version does to destroy old BEs, or resort to destruction of datasets snapshot-by-snapshot (and mind that `beadm destroy dataset@snapshot` syntax does offer a means to automate that). Alternately, consider using `zfs promote` to ensure that a newer clone is considered to be the master (inspect `zfs list -o origin,name -r rpool/ROOT` output to see the current relationships between datasets on your system).

On simply using "pkg upgrade"

Unfortunately, if you've issued a simple `pkg upgrade` call which results in a cloned BE automatically (due to package flags requiring a reboot), the new BE would currently have default compression and other per-dataset settings. Still, you have a chance to catch the new BE and fix it as early as you can "in flight".

Note that the package upgrade first refreshes its catalog of the packages in repositories, then downloads the new files into a local area (under `/var/pkg` in the current rootfs, apparently), and only then does it create a new cloned BE based on the current one. The BE name would be generated at this time; if your current one ended with a number (like `oi_151a8-20140101`) this number would be increnmented into a unique available number (like into `oi_151a8-20140102`, so don't expect current dates to be used automagically). For names without a number, one would just be appended (i.e. `openindiana-1` for a default installation's first substantial upgrade).

If you figure out the expected BE name, you can leave the following loop running in an alternate shell to catch the creation of the new BE and to fix its dataset attributes:

```
:; RPOOL="rpool" ; RPOOL_ROOT="$RPOOL/ROOT"
:; BEOLD="oi_151a8" ; BEOLD_DS="$RPOOL_ROOT/$BEOLD"
:; EXCLUDE_ATTRS='org.opensolaris.libbe:uuid|canmount|mountpoint'
:; BENEW="oi_151a9"
:; BENEW_DS="$RPOOL_ROOT/$BENEW" ; while ! zfs list -r "BENEW_DS"; do \
   sleep 1; done; sleep 1; zfs list -H -o name -r "$BEOLD_DS" | while read Z; do \
     S="`echo "$Z" | sed "s,^$BEOLD_DS,,"`"; echo "=== '$S'"; \
     zfs get all "$BEOLD_DS$S" | egrep ' (local|received)' | egrep -v "$EXCLUDE_ATTRS" | \
     while read _D A V _T; do echo "$A=$V"; zfs set "$A=$V" "$BENEW_DS$S"; done; done;
```

Do not despair if you've lost the moment or mis-guessed the `$BENEW` name, and an uncompressed clone was instantiated – you can just destroy it and redo the process (possibly, with `pkg upgrade --no-refresh`) using the same new package data that you've already downloaded, so the process should be cheap and fast now (that is, **if** you did not specify an alternate root with `-R` so that the current BE's `/var/pkg` repositroy was used to cache the package data). On the upside, now you know what `$BENEW` name the system would actually use (or you'd have a new chance to enforce one with `--be-name`).

Why bother for upgrades?

To conclude with an example, I have re-tested this procedure with the OI Hipster distribution which has frequent re-rolls of packages. For example, last night some 579 packages became obsolete, involving about 265MB of downloads and 3700 replaced files. An upgrade with re-enabled `gzip-9` and one without (still inheriting `lz4` from the root) differed by about 100MB in the `/usr` child dataset alone... that's a 1/3 difference relative to the download size (and even more in comparison to lack of compression), including less pressure from those smaller blocks of the OS files onto system caches. Just to give a tangible example:

```
# zfs list -o compression,refcompressratio,refer,logicalreferenced,name -r rpool/ROOT | egrep '/usr$'
gzip-9 2.91x 1.18G 3.38G rpool/ROOT/hipster-20140214/usr
gzip-9 2.90x 1.21G 3.43G rpool/ROOT/hipster-20140416/usr
gzip-9 2.90x 1.21G 3.43G rpool/ROOT/hipster-20140417-gzip9/usr
lz4    2.70x 1.30G 3.43G rpool/ROOT/hipster-20140417-lz4/usr
off    2.25x 1.55G 3.43G rpool/ROOT/hipster-20140417-nocomp/usr
```

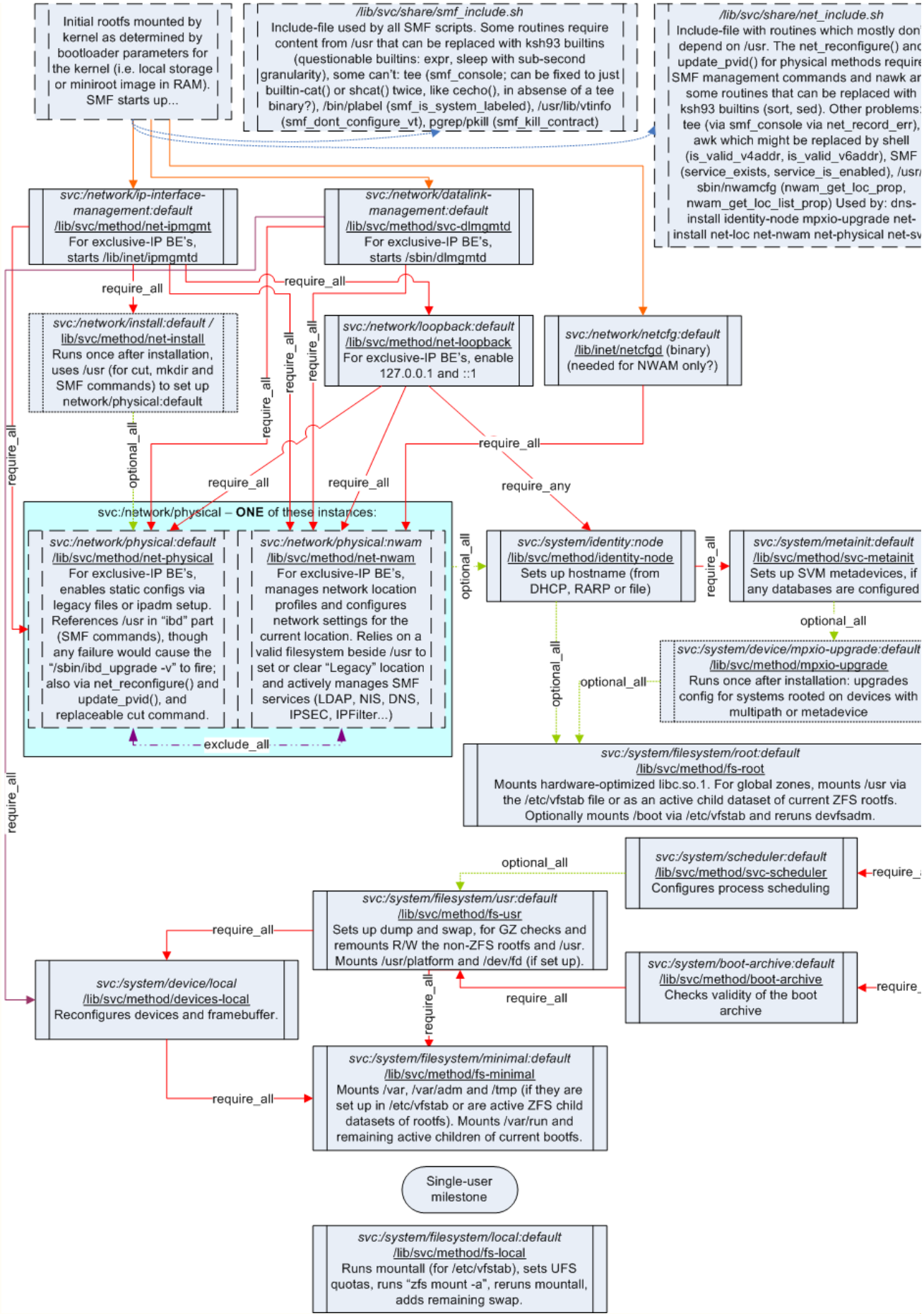
Note that "nocomp" still has the compression in place for files that were not changed since the original dataset from the day before, only the new files are not compressed.

HTH,
//Jim Klimov

An earlier note about naïve split-rooting approach which sometimes failed me - and why

⚠ It was recently discovered that NWAM network auto-configuration does not work with split-root config based on earlier modifications of `fs-root`, `fs-usr` and `fs-minimal` scripts (hopefully fixed with the recent rehaul to `fs-root-zfs` as the single solution for this use-case).

Tracing the system scripts has shown that a substantial part of them depends on availability of `/usr` or even more (in case of NWAM – rather on `filesystem/minimal` with a proper `/var` tree), yet services like `network/physical` are dependencies needed for startup of `filesystem/root` (which mounts and guarantees to provide the `/usr`). Most of the methods "broken" in this manner can be amended to use `ksh93` builtins and shell constructs instead of external programs and rely only on `/sbin` (after relocation of `ksh93` as `/sbin/sh`); other solutions are also possible and are now being discussed in the mailing list and the issue tracker. The legacy network method "for servers" (`svc:/network/physical:default`) happens to work successfully with both static configurations and DHCP, that's why the error was not found for years 😅


Initial rootfs mounted by kernel as determined by bootloader parameters for the kernel (i.e. local storage or miniroot image in RAM). SMF starts up...
/lib/svc/share/smf_include.sh
Include-file used by all SMF scripts. Some routines require content from /usr that can be replaced with ksh93 builtins (questionable builtins: expr, sleep with sub-second granularity), some can't: tee (smf_console; can be fixed to just builtin-cat() or shcat() twice, like cecho(), in absense of a tee binary?), /bin/plabel (smf_is_system_labeled), /usr/lib/vtinfo (smf_dont_configure_vt), pgrep/pkill (smf_kill_contract)
/lib/svc/share/net_include.sh
Include-file with routines which mostly don
depend on /usr. The net_reconfigure() and
update_pvid() for physical methods require
SMF management commands and nawk ar
some routines that can be replaced with
ksh93 builtins (sort, sed). Other problems:
tee (via smf_console via net_record_err),
awk which might be replaced by shell
(is_valid_v4addr, is_valid_v6addr), SMF
(service_exists, service_is_enabled), /usr/
sbin/nwamcfg (nwam_get_loc_prop,
nwam_get_loc_list_prop) Used by: dns-
install identity-node mpxio-upgrade net-
install net-loc net-nwam net-physical net-sv
svc:/network/ip-interface-management:default
/lib/svc/method/net-ipmgmt
For exclusive-IP BE's, starts /lib/inet/ipmgmtd
svc:/network/datalink-management:default
/lib/svc/method/svc-dlmgmtd
For exclusive-IP BE's, starts /sbin/dlmgmtd
require_all
svc:/network/install:default / lib/svc/method/net-install
Runs once after installation, uses /usr (for cut, mkdir and SMF commands) to set up network/physical:default
svc:/network/loopback:default
/lib/svc/method/net-loopback
For exclusive-IP BE's, enable 127.0.0.1 and ::1
svc:/network/netcfg:default
/lib/inet/netcfgd (binary)
(needed for NWAM only?)
optional_all
require_any
svc:/network/physical – ONE of these instances:
svc:/network/physical:default
/lib/svc/method/net-physical
For exclusive-IP BE's, enables static configs via legacy files or ipadm setup. References /usr in "ibd" part (SMF commands), though any failure would cause the "/sbin/ibd_upgrade -v" to fire; also via net_reconfigure() and update_pvid(), and replaceable cut command.
svc:/network/physical:nwam
/lib/svc/method/net-nwam
For exclusive-IP BE's, manages network location profiles and configures network settings for the current location. Relies on a valid filesystem beside /usr to set or clear "Legacy" location and actively manages SMF services (LDAP, NIS, DNS, IPSEC, IPFilter...)
exclude_all
svc:/system/identity:node
/lib/svc/method/identity-node
Sets up hostname (from DHCP, RARP or file)
svc:/system/metainit:default
/lib/svc/method/svc-metainit
Sets up SVM metadevices, if any databases are configured
svc:/system/device/mpxio-upgrade:default
/lib/svc/method/mpxio-upgrade
Runs once after installation: upgrades config for systems rooted on devices with multipath or metadevice
svc:/system/filesystem/root:default
/lib/svc/method/fs-root
Mounts hardware-optimized libc.so.1. For global zones, mounts /usr via the /etc/vfstab file or as an active child dataset of current ZFS rootfs. Optionally mounts /boot via /etc/vfstab and reruns devfsadm.
svc:/system/scheduler:default
/lib/svc/method/svc-scheduler
Configures process scheduling
svc:/system/filesystem/usr:default
/lib/svc/method/fs-usr
Sets up dump and swap, for GZ checks and remounts R/W the non-ZFS rootfs and /usr. Mounts /usr/platform and /dev/fd (if set up).
svc:/system/boot-archive:default
/lib/svc/method/boot-archive
Checks validity of the boot archive
svc:/system/device/local
/lib/svc/method/devices-local
Reconfigures devices and framebuffer.
svc:/system/filesystem/minimal:default
/lib/svc/method/fs-minimal
Mounts /var, /var/adm and /tmp (if they are set up in /etc/vfstab or are active ZFS child datasets of rootfs). Mounts /var/run and remaining active children of current bootfs.
Single-user milestone
svc:/system/filesystem/local:default
/lib/svc/method/fs-local
Runs mountall (for /etc/vfstab), sets UFS quotas, runs "zfs mount -a", reruns mountall, adds remaining swap.