BrandZ Design Document

7/11/06

1 Introduction

This project is delivering two sets of functionality:

- The BrandZ infrastructure, which enables the creation of zones that provide alternate operating environment personalities, or *brands* on a Solaris(tm) 10 system.
- *lx*, a brand that supports the execution of Linux applications

This document describes the design and implementations of both BrandZ and the *lx* brand.

2 BrandZ Infrastructure

2.1 Zones Integration

It is a goal of this project that all brand management should be performed as simple extensions to the current zones infrastructure:

- The zones configuration utility will be extended to configure zones of particular brands.
- The zone administration tools will be extended to display the brand of any created zone, as well as to list the brand types available.

BrandZ adds functionality to the zones infrastructure by allowing zones to be assigned a brand type. This type is used to determine which scripts are executed when a zone is installed and booted. In addition, a zone's brand is used to properly identify the correct application type at application launch time. All zones have an associated brand for configuration purposes; the default is the 'native' brand.

2.1.1 Configuration

The `zonecfg(1M)` utility and `libzonecfg` have been modified to be brand-aware by adding a new "brand" attribute, which may be assigned to any zone. This brand may be assigned explicitly:

```
# zonecfg -z newzone
newzone: No such zone configured
Use 'create' to begin configuring a new zone.
zonecfg:newzone> create -b
zonecfg:newzone> set brand=lx
```

Or it may be set implicitly, by using a brand-specific template:

```
# zonecfg -z newzone
newzone: No such zone configured
Use 'create' to begin configuring a new zone.
zonecfg:newzone> create -t SUNWlx
```

A zone's brand, like its zonepath, may only be changed prior to installing the zone.

The rest of the user interface for the zone configuration process remains the same. To support this, each brand delivers a configuration file at `/usr/lib/brand/<name>/config.xml`. The file describes how to install, boot, clone (and so on) the zone. It also identifies the name of the kernel module (if any) that provides kernel-level functionality for the brand, and lists the privileges that are, or may be, assigned

to the zone. A sample of the file follows:

```
<!DOCTYPE brand PUBLIC "-//Sun Microsystems Inc//DTD Brands//EN"
      "file:///usr/share/lib/xml/dtd/brand.dtd.1">

   <brand name="lx">
      <modname>lx_brand</modname>
      <initname>/sbin/init</initname>
      <install>/usr/lib/brand/lx/lx_install %z %R %*</install>
      <boot>/usr/lib/brand/lx/lx_boot %R %z</boot>
      <halt>/usr/lib/brand/lx/lx_halt %z</halt>
      <verify_cfg></verify_cfg>
      <verify_adm></verify_adm>
      <postclone></postclone>

      <privilege set="default" name="contract_event" />
      <privilege set="default" name="contract_observer" />
      <privilege set="required" name="proc_fork" />
      <privilege set="required" name="sys_mount" />
</brand>
```

2.1.2 Installation

The current zone install mechanism is hardwired to execute `lucreatezone` to install a zone. To support arbitrary installation methods, the `config.xml` file contains a line of the form:

```
<install>/usr/lib/lu/lucreatezone -z %z</install>
```

For the *lx* brand, this line will refer to `/usr/lib/brand/lx/lx_install` instead of `lucreatezone`. All of the command tags support the following substitutions '%z' (for zonename) and '%R' (for zone root path). In addition, the zoneadm(1M) command will pass additional arguments to the program as specified on the command line. This will allow runtime installation arguments, such as the location of source media:

```
# zoneadm -z linux install -d /net/installsrv/redhat30
```

2.1.3 Virtual Platform

The virtual platform consists of internal mounted filesystems, as well as the device tree and any network devices. The virtual platform is controlled by a brand-specific `platform.xml` file. The properties it controls are outlined below.

2.1.3.1 Mounted Filesystems

Before handing off control to the virtual platform, `zoneadmd` does some work of its own to set up well-known mounts and do some basic verification. It performs Solaris-specific mounts (`/proc`, `/system/contract`, etc.), which need to be defined for each brand. We also need to mount native versions of some files in order to support the interpretation layer (for native access to `/proc`, for example).

The set of filesystems mounted by `zoneadmd` are specified in the virtual platform configuration file:

```
<platform name="lx">
      <!-- Global filesystems to mount when booting the zone -->
      <global_mount special="/lib" directory="/native/lib"
            opt="ro" type="lofs" />
      <global_mount special="/usr/lib" directory="/native/usr/lib"
            opt="ro" type="lofs" />
      ...
      <!-- Local filesystems to mount when booting the zone -->
      <local_mount special="fd" directory="/dev/fd" type="fd" />
      <local_mount special="proc" directory="/native/proc" type="proc" />
      ...
```

2.1.3.2 Device Configuration

To create device nodes within a zone, `zoneadmd` calls `devfsadm`. The devfsadm process walks the current list of devices in `/dev`, and calls into `libzonecfg` to determine if the device should be exported.

libzonecfg will consult a brand-specific platform configuration file that describes how to apply the following semantics:

- **Policy restrictions**: Control which devices appear in the non-global zone. This is the only aspect currently implemented. This is currently hardcoded in `libzonecfg`.
- **Device inheritance**: Describes a device which matches exactly a device in the global zone. Currently this is assumed behavior, but it needs to be configurable.
- **Device renaming**: Allows devices to appear in diffent locations within the zone. This is needed to support devices that are emulated via layered drivers.

The platform configuration file will have elements to perform all the above tasks. A sample scheme could be:

```
      <!-- Devices to create under /dev -->

      <device match="pts/*" />
      <device match="ptmx" />
      <device match="random" />
      <device match="urandom" />

      ...

      <!-- Symlinks to create under /dev -->

      <symlink source="stderr" target="./fd/2" />
      <symlink source="stdin" target="./fd/0" />
      <symlink source="stdout" target="./fd/1" />

      <symlink source="systty" target="zconsole" />
      <symlink source="log" target="/var/run/syslog" />

      <!-- Renamed devices to create under /dev -->

      <device match="brand/lx/ptmx_linux" name="ptmx" />
```

2.1.4 Booting a Branded Zone

In addition to the install-time command attribute, the configuration file can also provide an optional 'boot' script. This script will be run after the zone has been brought to the 'Ready' state - immediately before the zone is booted. This allows a brand to perform any needed setup tasks after the virtual platform has been established, but before the first branded process has been launched, giving the brand complete flexibility in how the zone is brought online. If the boot script fails, the boot operation as a whole fails, and the zone is halted.

2.1.4.1 Spawning init

The `init(1M)` binary is spawned by the kernel, allowing the kernel to control its restart. The name and path of a zone's init(1M) binary is a per-zone attribute, which is stored in the brand configuration file and passed to the kernel by zoneadm immediately prior to booting the zone.

2.1.5 Cloning a Branded Zone

All zone clone operations start the same way: by making a copy of the zone. This copy may be done using either the `cpio(1M)` utility or, if the zone is rooted on its own ZFS filesystem, by using the filesystem cloning functionality built into ZFS. After the zone is copied, the zones infrastructure will execute an external post-processing utility provided by the brand and identified in the brand's `config.xml` file. If no utility is provided, the clone operation is considered complete as soon as the copy operation succeeds.

2.1.6 Verifying a Branded Zone

There are two different times in which a zone may need to be verified: when the zone is originally configured, and when the zone is booted. The first verification ensures that the zone configuration is legal and consistent. The second verification ensures that all the resources needed to boot the zone are available on the system. zonecfg(1M) and zoneadm(1M) will perform common verfication tasks, such as ensuring that the zone name contains no illegal characters, that the zone's root path exists and has t he proper permissions, and so on. A brand may provide external utilities to perform additional verification for either, or both, verification points. These utilities are identified in the brand's `config.xml` file.

2.2 Kernel Integration

2.2.1 Brand Framework

Brands are loaded into the system in the form of brand modules. A brand module is defined through a new linkage type:

```
struct modlbrand {
      struct mod_ops    *brand_modops;
      char              *brand_linkinfo;
      struct brand      *brand_branddef;
};
```

The brand module is automatically loaded by the kernel the first time a branded zone is booted.

Each brand must declare `struct brand` as part of the registration process.

```
typedef struct brand {
      int                    b_version;
      char                   *b_name;
      struct brand_ops       *b_ops;
      struct brand_mach_ops  *b_machops;
} brand_t;
```

This structure declares the name of the brand, the version of the brand interface against which it was compiled, and ops vectors containing both common and platform-dependent functionality. The `b_version` field is currently used to determine whether a brand is eligible to be loaded into the system. If the version number does not match the one compiled into the kernel, then we simply refuse to load the brand module. In theory, this version number could also be used to interpret the contents of the two ops vectors, allowing us to continue supporting older brand modules.

It is important to note that even though we have defined a linkage type for brands and have implemented a versioning mechanism, we are not defining a formal ABI for brands. The relationship between brands and the kernel is so intimate that we cannot hope to properly support the development of brands outside the ON consolidation. This does not mean that we will do anything to prevent the development of brands outside of ON, but we must minimize the possibility of an out-of-date brand accidentally damaging something within the kernel.

2.2.2 System Call Interposition

The system call invocation mechanism is an implementation detail of both the operating system and the hardware architecture. On SPARC machines, a system call is initiated via a trap (Solaris chooses to use trap numbers 8 and 9.) On x86 machines, there are a variety of different methods available to enter the kernel: `sysenter` and `syscall` instructions, `lcalls`, and software-triggered interrupts. Solaris has used all of these mechanisms at various points, and maintaining binary compatibility requires that we continue to support them all.

Supporting a different version of Solaris requires interposing on each of these mechanisms. Before we begin executing the system call support code for the native version of Solaris, we must ensure that the process does not belong to a brand that has a different implementation of those calls. To do that, we must check the proc structure of the initiating process to determine whether it belongs to a foreign brand, and if so, whether that brand has chosen to interpose on that entry point. This check is carried out by a `BRAND_CALLBACK()` macro placed at the top of the handling routine for each of the existing entry points. If the brand wishes to interpose on the entry point, control is passed to that brand's kernel module. If not, control returns to the handler and the standard Solaris system call code is executed.

Linux has a single means of entering the kernel: executing an 'int 80' instruction. Since this mechanism is not used by any version of Solaris, there is no existing mechanism on which we can interpose. Therefore the creation of a Linux brand requires that we provide a new entry point into the kernel specifically for this brand. As with the existing entry points, the `BRAND_CALLBACK()` macro is executed by this handler. In this case, there is no standard system call code to execute if the handler returns. Instead the program is killed with a General Protection Fault.

While introducing a new entry point is a trivial task within the Solaris engineering organization, it again demonstrates that the ability for third parties to distribute radically new brands will be limited. A third party may certainly distribute a modified version OpenSolaris that includes the changes they need, but any brands that depend on those changes would not work properly with our Solaris until/unless those changes were adopted by Sun.

For performance reasons, the `BRAND_CALLBACK()` macro is only invoked for branded processes. Each system call mechanism actually has two different entry points defined: one for branded processes and one for non-branded processes. A context handler is used to modify the appropriate CPU registers or structures to switch to the appropriate set of entry points during a context switch operation. The callback macro is only a dozen or so instructions long, but this switching mechanism ensures that non-branded processes are not subject to even that minimal overhead.

This interposition mechanism is executed before any other code in the system call path. Specifically, it is executed before the standard system call prologue. In a sense, this means that we are executing in kernel mode, but don't consider ourselves to have entered the kernel. This may sound esoteric, but it has concrete

implications for the brand-specific code. The brand code cannot take any locks, cannot do any I/O, or do anything else that might cause the thread to block. This means that the brand code can only do the simplest computations or transformations before returning to userspace or to the standard system call flow.

As described below, the 'lx' brand returns immediately to userspace, where the bulk of the emulation takes place. To implement a brand in the kernel, the interposition routine could transform the incoming system call into a brand() system call and return to the normal system call path. After executing the standard system call prologue, control would be vectored to the brand's xxx_brandsys() routine, where the emulation could be carried out.

2.2.3 Other Interposition Points

Other interposition points will be placed in the code paths for process creation/exit, thread creation/teardown, and so on.

The interposition points are identified by means of a pair of ops vectors (one generic and one platform-specific), similar to those used by VFS for filesystems and the VM subsystem for segments. The generic ops vector used by BrandZ is shown below:

```
struct brand_ops {
        int     (*b_brandsys)(int, int64_t *, uintptr_t, uintptr_t,
                uintptr_t, uintptr_t, uintptr_t, uintptr_t);
        void    (*b_setbrand)(struct proc *);
        void    (*b_copy_procdata)(struct proc *, struct proc *);
        void    (*b_proc_exit)(struct proc *, klwp_t *);
        void    (*b_exec)();
        void    (*b_lwp_setrval)(klwp_t *, int, int);
        int     (*b_initlwp)(klwp_t *);
        void    (*b_forklwp)(klwp_t *, klwp_t *);
        void    (*b_freelwp)(klwp_t *);
        void    (*b_lwpexit)(klwp_t *);
        int     (*b_elfexec)(struct vnode *vp, struct execa *uap,
                struct uarg *args, struct intpdata *idata, int level,
                long *execsz, int setid, caddr_t exec_file,
                struct cred *cred);
};
```

A brief description of each entry follows:

- `b_brandsys`: Routine that implements a per-brand 'brandsys' system call that can be used for any brand-specific functionality.
- `b_setbrand`: Used to associate a new brand type with a process. This is called when the zone's init process is hand-crafted, or when a process uses zone_enter() to enter a branded zone.
- `b_copy_procdata`: Copies per-process brand data at fork time.
- `b_proc_exit`: Called at process exit time to free any brand-specific process state.
- `b_exec`: Called at process exec time to initialize any brand-specific process state.
- `b_lwp_setrval`: set the syscall() return value for the newly created lwp before the creating fork () system call returns.
- `b_initlwp`: Called by `lwp_create()` to initialize any brand-specific per-lwp data.
- `b_forklwp`: Called by `forklwp()` to copy any brand-specific per-lwp data from the parent to child lwps.
- `b_freelwp`: Called by `lwp_create()` on error to do any brand-specific cleanup.
- `b_lwpexit`: Called by `lwp_exit()` to do any brand-specific cleanup.
- `b_elfexec`: Called to load a branded executable.

The x86-specific ops vector is:

```
struct brand_mach_ops {
      void        (*b_sysenter)(void);
      void        (*b_int80)(void);
      void        (*b_int91)(void);
      void        (*b_syscall)(void);
      void        (*b_syscall32)(void);
      greg_t      (*b_fixsegreg)(greg_t, model_t);
};
```

The first 5 entries of this vector allow a brand to override the standard system call paths with their own interpretations. The final entry protects Solaris from brands that make different use of the segment registers in userspace, and vice-versa.

The SPARC-specific ops vector is:

```
struct brand_mach_ops {
      void  (*b_syscall)(void);
      void  (*b_syscall32)(void);
};
```

Of these routines, only the `int80` entry of the x86 vector is needed for the initial *lx* brand. The other entries are included for completeness and are only used by a trivial 'Solaris N-1' brand used for basic testing of the BrandZ framework on systems without any Linux software.

These ops vectors are sufficient for the initial Linux brand , but supporting a whole new operating system such as FreeBSD or Apple's OS X would almost certainly require modifying this interface.

3 The *lx* Brand

3.1 Brands and Distributions

The *lx* Brand is intended to emulate the kernel interfaces expected by the Red Hat Enterprise Linux 3 userspace. The freely available CentOS 3 distribution is built from the same SRPMs as RHEL, so it is expected to work as well.

The interface between the kernel and userspace is largely encapsulated by the version of *glibc* that ships with the distribution. As such, the interface we emulate will likely support other distributions that make use of glibc 2.3.2. For example, while we will not be supporting the Debian distribution officially, Debian 3.1 uses this version of *glibc* and has been shown to work with the *lx* brand.

It should be noted that supporting a new distribution will always require a new install script. For RPM-based distributions, it might be sufficient to update the existing scripts with new package lists. Adding support for distributions such as Debian, which do not use the RPM packaging format, will require entirely new install scripts to be created. It is relatively simple to have a variety of different install scripts within a single brand, so simply changing the packaging format does not require the creation of a new brand.

Since all the Red Hat and CentOS versions we support include the same kernel version, the uname(2) information reported to processes in a Linux zone is hardcoded into the *lx* brand.

```
kingston$ uname -a
Linux kingston 2.4.21 BrandZ fake linux i686 i686 i386 GNU/Linux
kingston$ more /proc/version
Linux version 2.4.21 (Sun C version 5.7.0) #BrandZ fake linux SMP
```

If we were to add support for other distributions, this information would have to be made zone-specific.

3.2 Installation of a Linux zone

Most Linux distributions' native installation procedures start by probing and configuring the hardware on the system and partitioning the hard drives. The user then selects which packages to install and sets configuration variables such as the hostname and IP address. Finally the installer lays out the filesystems, installs the packages, modifies some configuration files, and reboots.

When installing a Linux zone, we can obviously skip the device probing and disk partitioning. For the remainder of the installation procedure, there are several different approaches we could take.

- One approach is to execute a distribution's installation tool directly. Most of them are based on shell scripts, Python, or some other interpreted language, so we could theoretically run the tools before having a Linux environment running. This approach relies on the existing install tools being fairly robust and would be hard to sustain between releases if the tools change.
- Another option is to develop our own package installation tool, which extracts the desired software from a standard set of distribution media and copies it into the zone.
- A third option would be to adopt a flasharchive-like approach, in which we would simply unpack a prebuilt tarball or cpio image into the target filesystem. This image could be built by us, or by a customer that already had an installed Linux system. If we were to build this image ourselves, this approach would give us complete control over the final image and and allow us to manually handle any particularly ugly early-stage installation issues. This would also make it trivial for a customer to "just try it out," an approach that has been successful for VMware, Usermode Linux, and QEMU.

It is our intention to support the last two options.

We will provide an installation script that will extract a known set of RPMs from a known set of Red Hat or CentOS installation media. We will also allow a zone to be installed from a user-specified tarball. We will document how a user can construct an installable tarball that can be used for flasharchive-like installations.

We will provide a “silent” installation mode as well, allowing for non-interactive installs.

3.3 Execution of Linux Binaries

When executing Linux binaries, we follow the same architectural approach taken by the SunOS(tm) Binary Compatibility Project (SBCP), which provides support for pre-Solaris 2.0 binaries.

3.3.1 Loading Linux Binaries

When an ELF binary is execed inside a branded zone, the brand's elfexec handler is given control. The *lx* brand-specific elfexec handler execs the brand's Solaris support library (akin to the 'sbcp' command) and maps the non-native ELF executable and its interpreter into the address space.

The handler also places all the extra information needed to exec the Linux binary on the stack in the form of aux vector entries. Specifically, the handler passes the following values to the support library via the aux vector:

```
AT_SUN_BRAND_BASE:      Base address of non-native interpeter
AT_SUN_BRAND_LDDATA:    Address of non-native interpreter's debug data
AT_SUN_BRAND_LDENTRY:   Non-native interpreter's entry point
AT_SUN_BRANDNAME:       Name of the process's brand
AT_SUN_BRAND_PHDR:      Non-native executable ELF information needed by
AT_SUN_BRAND_PHENT:     non-native interpreter
AT_SUN_BRAND_PHNUM:
AT_SUN_BRAND_ENTRY:     Entry point of non-native executable
```

You cannot run   Solaris binaries inside a Linux zone. Any attempt to do so will fail with `"No such file or directory"`, as it does on a real Linux system.

3.3.2 Running Linux Binaries

Rather than executing the Linux application directly, the exec handler starts the program execution in the brand support library. The library then runs whatever initialization code it needs before starting the Linux application.

For its initialzation, the `lx` brand library uses the `brandsys(2)` system call to pass the following data structure to the kernel:

```
typedef struct lx_brand_registration {
      uint_t      lxbr_version;     /* version number */
      void        *lxbr_handler;    /* base address of handler */
      void        *lxbr_traceflag;  /* address of trace flag */
} lx_brand_registration_t;
```

This structure contains a version number, ensuring that the brand library and the brand kernel module are compatibile with one another. It also contains two addresses in the application's address space: the starting address of the system call emulation code and the address of a flag indicating whether the process is being traced. The use of these two addresses are discussed in sections 3.4 and 3.9 respectively.

Once the support library has finished initialization, it constructs a new aux vector for the Linux interpreter to run, and jumps to the interpreter's entry point. The brand's exec handler replaces the standard aux vector entries with the corresponding values above, clears the above vectors (by setting their type to AT_IGNORE), resets the stack to its pre-Solaris-linker state, and then jumps to the non-native interpreter which then runs the executable as it would on its own native system.

The advantages of this design are:

- No modifications to the Solaris runtime linker are necessary.
- Virtually all knowledge of branding is isolated to the kernel brand module and userland brand support library.
- It keeps us aligned with the de-facto standard for non-native emulation established with SunOS 4 BCP.

3.4 System Call Emulation

The most common method for applications to interact with the operating system is through system calls. Therefore, the bulk of the work required to support the execution of foreign binaries is to emulate the system call interface to which those binaries have been written.

Linux applications do not use the `syscall/sysenter` instructions. Instead, they use interrupt #80 to initiate a system call. Because Solaris has no current handler for that interrupt, one had to be added as part of this project. As noted above, the handler in the core Solaris code does nothing but pass control to the int80_handler routine in the brand module. It is then up to that brand to interpret and execute the system call.

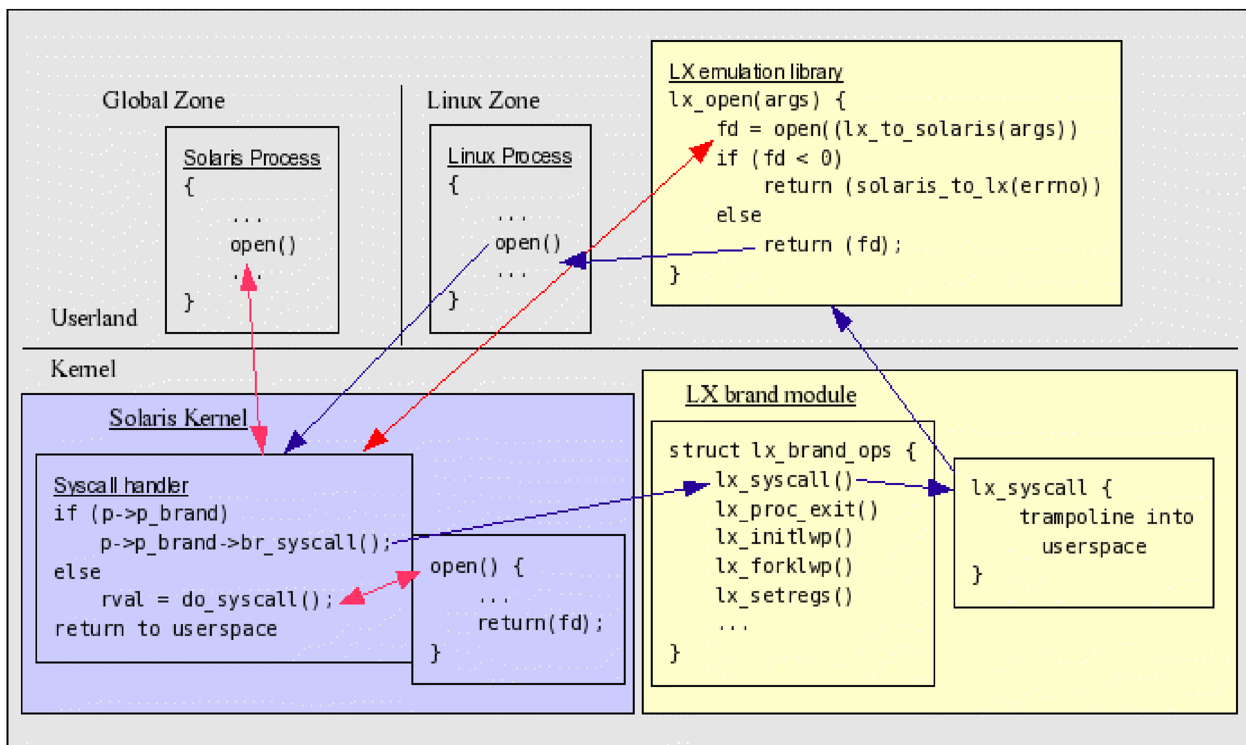
As with executable launching, the approach we have chosen to take is the one originally implemented by the SBCP. In this model, the trap handler in the kernel is nothing more than a trampoline, redirecting the execution of the system call to a library in userspace. The library performs any necessary data mangling and then calls the proper native system call. Ideally, this method would require almost no code in the kernel and would have no impact whatsoever on the behavior of a system with no SunOS binaries running.

In practice, the user-space approach turns out to be less clean and self-contained for Linux than for the original SunOS project. In that case, the binary model being emulated was simpler than Solaris, less fully featured, and still closely related. In the Linux case, there are system calls that must be supported that do not have obvious equivalents in Solaris (e.g., futex()) and there are differences in fundamental OS abstractions (Linux 'threads' are almost full blown processes, each with its own PID).

The steps involved in emulating a fairly straightforward Linux system call are as follows:

1. The Linux application marshalls parameters into registers and issues an `int80`
2. The Solaris int80 handler checks to see if the process is branded. An unbranded process will continue along the standard code path. For `int80`, there is no standard behavior so the process would die with a General Protection fault. Thus, a Solaris application cannot successfully execute any Linux system calls.
3. Solaris passes control to the brand-specific routine indicated in the brandops structure.
4. The *lx* brand module immediately trampolines into the user-space emulation library.
5. The emulation library does any necessary argument manipulation and calls the appropriate Solaris system call(s).
6. Solaris carries out the system call and returns to the brand library.
7. The brand library performs any necessary manipulation of the return values and error code.
8. The brand library returns directly to the Linux application; it does not return through the kernel.

The diagram below illustrates these steps:



When debugging an application with mdb, any attempt to step into or over a system call will leave the user in the BrandZ emulation code.

Each Linux system call can be divided into one of three types: pass-through, simple emulation, and complex emulation.

3.4.1 Pass-Through

A pass-through call is one that requires no data transformation and for which the Solaris 10 semantics match those of the Linux system call. These can be implemented in userland by immediately calling the equivalent system call.

For example:

```
int
lx_read(int fd, void *buf, size_t bytes)
{
	int rval;

	rval = read(fd, buf, bytes);

	return (rval < 0 ? -errno : rval);
}
```

Other examples of pass-through calls are close(), write(), mkdir(), and munmap().

Although the arguments to the system call are identical, the method for returning an error to the caller differs between Solaris and Linux. In Solaris, the system call returns -1 and the error number is stored in the thread-specific variable `errno`. In Linux, the error number is returned as part of the `rval`.

There are also differences in the error numbers between Solaris and Linux. The `lx_read()` routine is called by `lx_emulate()`, which handles the translation between Linux and Solaris error codes for all system calls.

3.4.2 Simple Emulation

One step up in complexity is a simple emulated system call. This is a call where either the original arguments and/or return value require some degree of simple transformation from the Solaris equivalent. Simple transformations include changing data types or the moving of values into a structure. These calls can be built entirely from standard Solaris system calls and userland transformations.

For example:

```
int
lx_uname(uintptr_t p1)
{
	struct lx_utsname *un = (struct lx_utsname *)p1;
	char buf[LX_SYS_UTS_LN + 1];

	strlcpy(un->sysname, LX_SYSNAME, LX_SYS_UTS_LN);
	strlcpy(un->release, LX_RELEASE, LX_SYS_UTS_LN);
	strlcpy(un->version, LX_VERSION, LX_SYS_UTS_LN);
	strlcpy(un->machine, LX_MACHINE, LX_SYS_UTS_LN);
	gethostname(un->nodename, sizeof (un->nodename));
	if ((sysinfo(SI_SRPC_DOMAIN, buf, LX_SYS_UTS_LN) < 0))
		un->domainname[0] = '\0';
	else
		strlcpy(un->domainname, buf, LX_SYS_UTS_LN);
	return (0);
}
```

Other examples of simple emulated calls are stat(), mlock(), and getdents().

3.4.3 Complex Emulation

The calls requiring the most in-kernel support are the complex emulated system calls. These are calls that:

- Require significant transformation to input arguments or return values

- Are partially or wholly unique within the *lx* brand implementation
- Possibly require a new system call that has no underlying Solaris counterpart

Some examples of complex emulated calls are `clone()`, `sigaction()`, and `futex()`. The implementation of the `clone()` system call is described below.

3.5 Other Issues

3.5.1 Linux Threading

Linux implements threads via the `clone()` system call. Among the arguments to the call is a set of flags, four of which determine the level of sharing within the address space: CLONE_VM, CLONE_FS, CLONE_FILES, and CLONE_SIGHAND. When all four flags are clear, the clone is equivalent to a fork; when they are all set, it is the equivalent to creating another lwp in the address space. Any other combination of flags reflects a thread/process construct that does not match any existing Solaris model. Since these other combinations are rarely, if ever, encountered on a system, this project will not be adding the abstractions necessary to support them.

The following table lists all of the flags for the `clone(2)` system call, and whether the 'lx' brand supports them. If an applications uses an unsupported flag, or combination of flags, a detailed error message is emitted and ENOTSUP is returned.

Flag	Supported?
CLONE_VM	Yes
CLONE_FS	Yes
CLONE_FILES	Yes
CLONE_SIGHAND	Yes
CLONE_PID	Yes
CLONE_PTRACE	Yes
CLONE_PARENT	Partial. Not supported for fork()-style clone() operations.
CLONE_THREAD	Yes
CLONE_SYSVSEM	Yes
CLONE_SETTLS	Yes
CLONE_PARENT_SETTID	Yes
CLONE_CHILD_CLEARTID	Yes
CLONE_DETACH	Yes
CLONE_CHILD_SETTID	Yes

When an application uses `clone(2)` to fork a new process, the `lx_clone()` routine simply calls `fork1(2)`. When an application uses `clone(2)` to create a new thread, we call the `thr_create (3C)` routine in the Solaris `libc`.

The Linux application provides the address of a function at which the new thread should begin executing as an argument to the system call. However, the Linux kernel does not actually start execution at that address. Instead, the kernel essentially does a `fork(2)` of a new thread which, like a forked process, starts with exactly the same state as the parent thread. As a result, the new thread starts executing in the middle of the `clone(2)` system call, and it is the `glibc` wrapper that causes it to jump to the user-specified address.

This Linux implementation detail means that when we call `thr_create(3C)` to create our new thread, we cannot provide the user's start address to that routine. Instead, all new Linux threads begin by executing a routine that we provide, called `clone_start()`. This routine does some final initialization, notifies the brand's kernel module that we have created a new Linux thread, and then returns to `glibc`.

A by-product of threads implementation in Linux is that every thread has a unique PID. To mimic this behavior in the *lx* brand, every thread created by a Linux binary reserves a PID from the PID list. This reservation is performed as part of the `clone_start()` routine. (Note: to prevent a pidreserve bomb from crippling the system, the `zone.max-lwps rctl` may be used to limit the number of PIDs allocated by a zone.)

This reserved PID is never seen by Solaris processes, but it is used by Linux processes. When a Linux thread calls `getpid(2)`, it is returned the standard Solaris PID of process. When it calls `gettid(2)`, it is returned the PID that was reserved at thread creation time. Similarly, `kill(2)` sends a signal to the entire process represented by the supplied PID, while `tkill(2)` sends a signal to the specific thread represented by the supplied PID.

The Linux thread model supported by modern RedHat systems is provided by the Native Posix Threads Library (NPTL). NPTL uses three consecutive descriptor entries in the Global Descriptor Table (GDT) to manage thread local storage. One of the arguments to the `clone()` is an optional descriptor entry for TLS. More commonly used is the `set_thread_area()` system call, which takes a descriptor as an argument and returns the entry number in the GDT in which it has been stored. The NPTL then uses this to initialize the `%gs` register. The descriptors are per thread, so they have to be stored in per thread storage and the GDT entries must be re-initialized on context switch. This is done via a `restore` ctx operation.

Since both NPTL and the Solaris `libc` rely on `%gs` to access per-thread data, we have added code to virtualize its usage. The first thing our user-space emulation library does is:

```
/*
* Save the Linux libc's %gs and switch to the Solaris libc's %gs
* segment so we have access to the Solaris errno, etc.
*/
pushl %gs
pushl $LWPGS_SEL
popl  %gs
```

This sequence ensures that we always enter our Solaris code using the well-known value used for our `%gs`. We also stash the current value of `%gs` on the stack, so we can restore it prior to returning to Linux code.

3.5.2 EFAULT/SIGSEGV

If the user-space emulation library were to access an argument from a system call which had an invalid address, a SIGSEGV signal would be generated. For proper Linux emulation, the desired result in this situation is to generate an error return from the system call with an EFAULT errno.

To deliver the expected behavior, we will introduce a new system call (`uucopy()`), which copies data from one user address to another. Any attempt to use an illegal address will cause the call to return an error. Otherwise, the data will be copied as if we had performed a standard `bcopy()` operation.

For example:

```
int
lx_system_call(int *arg)
{
      int   local_arg;
```

```
        int   rval;

        /*
         * catch EFAULT
         */
        if ((rval = uucopy(arg, &arg, sizeof (int))) < 0)
              return (rval);    /* errno is set to EFAULT */

        /*
         * transform the arg, now in local_arg, to Solaris format
        */
        return (solaris_system_call(&local_arg));
}
```

This functionality seems to be generically useful, so the `uucopy()` call will be implemented in `libc`, where it will be available to any application.

If the overhead imposed by this system call dramatically limits performance, we may include an environment variable that causes the brand library to perform a standard userspace copy rather than the kernel-based copy. Setting this variable would lead to higher performance, but some system calls would segfault rather than returning EFAULT.

3.5.3 Linux LDT usage

On x86 machines, Linux uses the LDT to support thread-local storage. Some applications use the LDT directly for various purposes.

The Linux kernel provides several interfaces to `glibc` to facilitate the TLS implementation. These interfaces hide the fact the Linux is using the LDT, making that a simple implementation detail. We implement these interfaces using the GDT, avoiding the performance overhead that comes with maintaining per-process LDTs. To this end, we reserve 3 slots in the GDT for use by the brand infrastructure. When an application uses the `get_thread_area()/set_thread_area()` system calls, we use these GDT-indexed segments.

Linux also provides an interface to create/remove entries from the LDT directly: `modify_ldt(2).` We implement this system call in the lx brand module, building on the existing Solaris support for per-process LDTs. This interface allows us to run (among other things) Linux media player applications l ike `xine` and `mplayer` that use Windows plugin codec modules.

3.5.4 /etc/mnttab

Linux keeps the current filesystem mount state in /etc/mtab. This file is a plain text file and its contents are maintained by the mount command. Applications trying to determine what mounts currently exist on the system normally access this file via setmntent(3) call. Linux also exports the current system mount state via /proc/mounts, ut most applications don't access this file. For applications that do attempt to access this file it is emulated in lx_procfs.

3.5.5 ucontext_t

The Linux and Solaris `ucontext_t` structures are slightly different. The most significant differences are that the Linux `ucontext_t` contains a signal mask and a copy of the x86 `%cr2` register. The signal mask is maintained by glibc – not the Linux kernel, so there is no extra work required of us. When we deliver a SIGSEGV, we fill in the `%cr2` field using information available in the `siginfo_t`.

3.6 Signal Handling

Delivering signals to a Linux process is complicated by differences in signal numbering, stack structure and contents, and the action taken when a signal handler exits. In addition, many signal-related structures, such as `sigset_ts`, vary between Solaris and Linux.

The simplest transformation that must be done when sending signals is to translate between Linux and Solaris signal numbers.

Major signal number differences between Linux and Solaris

Number	Linux	Solaris
10	SIGUSR1	SIGBUS
12	SIGUSR2	SIGSYS
16	SIGSTKFLT	SIGUSR1
17	SIGCHLD	SIGUSR2
18	SIGCONT	SIGCHLD
19	SIGSTOP	SIGPWR
20	SIGTSTP	SIGWINCH
21	SIGTTIN	SIGURG
22	SIGTTOU	SIGPOLL
23	SIGURG	SIGSTOP
24	SIGXCPU	SIGTSTP
25	SIGXFSZ	SIGCONT
26	SIGVTALARM	SIGTTIN
27	SIGPROF	SIGTTOU
28	SIGWINCH	SIGVTALARM
29	SIGPOLL	SIGPROF
30	SIGPWR	SIGXCPU
31	SIGSYS	SIGXFSZ

When a Linux process sends a signal using the `kill(2)` system call, we translate the signal into the Solaris equivalent before handing control off to the standard signalling mechanism. When a signal is delivered to a Linux process, we translate the signal number from Solaris back to Linux. Translating signals both at generation and at delivery time ensures both that Solaris signals are sent properly to Linux applications and that signals' default behavior works as expected.

One issue is that Linux supports 32 real time signals, with `SIGRTMIN` typically starting at or near 32 (`SIGRTMIN`) and proceeding to 63 (`SIGRTMAX`) (`SIGRTMIN`) is "at or near" 32 because glibc usually "steals" one ore more of these signals for its own internal use, adjusting `SIGRTMIN` and `SIGRTMAX` as needed.) Conversely, Solaris actively uses signals 32-40 for other purposes and and only supports 7 realtime signals, in the range 41 (`SIGRTMIN`) to 48 (`SIGRTMAX`).

At present, attempting to translate a Linux signal greater than 39 (corresponding to the maximum real time signal number Solaris can support) will generate an error. We have not yet found an application that attempts to send such a signal.

Branded processes are set up to ignore any Solaris signal for which there is no direct Linux analog,

preventing the delivery of untranslatable signals from the global zone.

3.6.1 Signal Delivery

To support user-level signal handlers, BrandZ uses a double layer of indirection to process and deliver signals to branded threads.

In a normal Solaris process, signal delivery is interposed on for any thread registering a signal handler by libc. Libc needs to do various bits of magic to provide thread-safe critical regions, so it registers its own handler, named `sigacthandler`, using the `sigaction(2)` system call. When a signal is received, `sigacthandler()` is called, and after some processing, libc calls the user's signal handler via a routine named `call_user_handler()`.

Adding a Linux branded thread to the mix complicates things somewhat. First, when a thread receives a signal, it could be running with a Linux value in the x86 %gs segment register as opposed to the value Solaris threads expect; if control were passed directly to Solaris code, such as libc's `sigacthandler()`, that code would experience a segmentation fault the first time it tried to dereference a memory location using %gs.

Second, the signal number translation referenced above must take place. (As an example, `SIGCONT` is equivalent in function in Linux and Solaris, but Linux' `SIGCONT` is signal 18 while Solaris' is signal *25*.) Further, as was the case with Solaris libc, before the Linux signal handler is called, the value of the %gs segment register must be restored to the value Linux code expects.

This need to translate signal numbers and manipulate the %gs register means that while with standard Solaris libc, following a signal from generation to delivery looks something like:

```
kernel ->
      sigacthandler() ->
            call_user_handler() ->
                  user signal handler
```

for BrandZ Linux threads, this instead would look like this:

```
kernel ->
      lx_sigacthandler() ->
            sigacthandler() ->
                  call_user_handler() ->
                        lx_call_user_handler() ->
                              Linux user signal handler
```

The new addtions are:

- **lx_sigacthandler()**
  This routine is responsible for setting the %gs segment register to the value Solaris code expects, and jumping to Solaris' libc signal interposition handler, sigacthandler().
- **lx_call_user_handler()**
  This routine is responsible for translating Solaris signal numbers to their Linux equivalents, building a Linux signal stack based on the information Solaris has provided, and passing the stack to the registered Linux signal handler. It is, in effect, the Linux thread equivalent to libc's `call_user_handler`.

Installing `lx_sigacthandler()` is a bit tricky, as normally libc's `sigacthandler()` routine is hidden from user programs. To facilitate this, a new private function was added to libc, `setsigacthandler()`:

```
void setsigacthandler(void (*new_handler)(int, siginfo_t , void),
```

```
void (old_handler **)(int, siginfo_t, oid *))
```

The routine works by modifying the per-thread data structure that libc already maintains that keeps track of the address of its own interposition handler with the address passed in; the old handler's address is set in the pointer pointed to by the second argument, if it is non-NULL, mimicking the behavior of `sigaction()` itself. Once `setsigacthandler()` has been executed, all future branded threads this thread may create will automatically have the proper interposition handler installed as the result of a normal `sigaction()` call.

Note that none of this interposition is necessary unless a Linux thread registers a user signal handler, because the default action for all signals is the same between Solaris and Linux save for one signal, `SIGPWR`. For this reason, BrandZ always installs its own internal signal handler for `SIGPWR` that translates the action to the Linux default, to terminate the process. (Solaris' default action is to ignore `SIGPWR`.)

It is also important to note that when signals are not translated, BrandZ relies upon code interposing upon the `wait(2)` system call to translate signals to their proper values for any Linux threads retrieving the status of others. So, while the Solaris signal number for a particular signal is set in the data structures for a process (and would be returned as the result of, for example, `WTERMSIG()`), the BrandZ interposition upon `wait(2)` is responsible for translating the value `WTERMSIG()`, and would return from a Solaris signal number to the appropriate Linux value.

3.6.1.1 Signals to init(1M)

Linux does not allow `kill(2)` to send any signals to `init(1M)` for which a signal handler has not been registered (including `SIGSTOP` and `SIGKILL`, as by definition handlers **cannot** be registered for those signals.) We solve this by having lx_kill() make the same check and only allow a signal to be sent to the Linux `init` process if a handler for the signal has already been registered. The Linux man pages are silent on whether the Linux *kernel* can send `init` an unhandled signal, so we do not perform any particular checks to prevent the Solaris kernel from doing so.

3.6.2 Returning From Signals

The process of returning to an interrupted thread of execution from a user signal handler is entirely different between Solaris and Linux. While Solaris generally expects to set the context to the interrupted one on a normal return from a signal handler, in the normal case Linux instead sets the return address from the signal handler to point to code that calls one of two specific Linux system calls, `sigreturn(2)` or `rt_sigreturn(2)`. Thus, when a Linux signal handler completes execution, instead of returning through what would, in Solaris' libc be a call to `setcontext(2)`, the `sigreturn(2)` or `rt_sigreturn(2)` Linux system calls are responsible for accomplishing much the same thing.

This trampoline code (for a call to `sigreturn(2)`) looks like this:

```
pop   %eax
mov   LX_SYS_sigreturn, %eax
int   $0x80
```

such that when the Linux user signal handler is eventually called, the stack looks like this:

Pointer to *sigreturn* trampoline code
Linux signal number
Pointer to Linux siginfo_t
Pointer to Linux ucontext_t
Linux ucontext_t
Linux fpstate
Linux siginfo_t

BrandZ takes the approach of intercepting the Linux `sigreturn(2)` (or `rt_sigreturn(2)`) system call in order to turn it into a return through the libc call stack that Solaris expects. This is done by the `lx_sigreturn()` or `lx_rt_sigreturn()` routines, which remove the Linux signal frame from the stack and pass the resulting stack pointer to another routine, `lx_sigreturn_tolibc()`, which makes libc believe the user signal handler it had called returned.

When control then returns to libc's `call_user_handler()` routine, a `setcontext(2)` will be done that (in most cases) returns the thread executing the code back to the location originally interrupted by receipt of the signal.

One final complication in this process is the restoration of the %gs segment register when returning from a user signal handler. Prior to BrandZ, Solaris' libc forced the value of %gs to a known value when calling `setcontext()` to return to an interrupted thread from a user signal handler (as libc uses %gs internally as a pointer to curthread, it is a way of ensuring a good "known value" for curthread.)

Since BrandZ requires that setcontext() restore a Linux value for %gs when returning from a Linux signal handler, we made this forced restoration optional on a per-process basis. This was accomplished by means of a new private routine to libc:

```
void set_setcontext_enforcement(int on)
```

By default, the "curthread pointer" value enforcement is enabled. When this routine is called with an argument of '0', the mechanism is disabled for this process.

Shutting off this mechanism will not have any correctness or security implications. Writing to the %gs segment register is not a privileged operation and as such %gs can be set to any value at any time by user code. The only drawback to disabling the mechanism is that if a bad value is set for %gs, the broken application will likely segmentation fault deep within libc.

3.7 Device and `ioctl()` support

3.7.1 Determining Which Devices to Support

Our investigation showed that he following devices are the minimum set required to support Linux branded zones:

```
/dev/null
/dev/zero
/dev/ptmx
/dev/pts/*
/dev/tty
/dev/console
/dev/random
/dev/urandom
/dev/fd/*
```

The following devices were considered, but aren't actually necessary for Linux branded zones.

`/dev/ttyd?` - These are serial port devices.

`/dev/pty[pqr]?` and `/dev/tty[pqr]?` - These are old style terminal devices provided for compatibility purposes. They currently do not exist in native non-global zones. The Unix98 specification replaced these devices with `/dev/ptmx` and `/dev/pts/*`, which have been used as the standard terminal devices for Solaris and Linux for a long time. While these devices do still exist on Red Hat 2.4 systems, they have officially been unsupported since Linux 2.1.115. An inspection of a Linux 2.4 system didn't reveal any applications that were actually using these devices.

Networking Devices

Native Solaris non-global zones have a network interface that is visible (reported via ifconfig), but there are no actual network device nodes accessible via `/dev`. Certain higher level network protocol devices are accessible in native zones:

```
/dev/arp, /dev/icmp, /dev/tcp, /dev/tcp6, /dev/udp, /dev/udp6
/dev/ticlts, /dev/ticots, /dev/ticotsord
```

Notably missing from the list above is:

```
/dev/ip
```

Looking at a native Linux 2.4 system, we see that the following network devices exist:

```
/dev/inet/egp, /dev/inet/ggp, /dev/inet/icmp, /dev/inet/idp
/dev/inet/ip, /dev/inet/ipip, /dev/inet/pup, /dev/inet/rawip
/dev/inet/tcp, /dev/inet/udp
```

`Documentation/devices.txt` describes all of these as 'iBCS-2 compatibility devices', so we will not be supporting them.

Additionally, Linux does not create `/dev` entries for networking devices. Network interface names are mapped to kernel modules via aliases defined in `/etc/modules.conf`. Interface plumbing (via ifconfig) is all done via ioctls to sockets. Reporting status of interfaces (via ifconfig) is done either by socket ioctls or by accessing files in /proc/net/. Finally, network accesses (telnet and ping) are all done via socket calls.

This indicates that initial Linux zones networking support has no actual device requirements, but it does require ioctl translations. (This issue is addressed later in this document.) Given the lack of device-specific configuration, the current native zones network interface can be leveraged in Linux branded zones.

3.7.2 Major/Minor Device Number Mapping

Linux has explicitly hardcoded knowledge about how major and minor device numbers map to drivers and paths. (See `Documentation/devices.txt` in the Linux source.) But on Solaris, major device number to driver mapping is dynamically defined via `/etc/name_to_major`. Minor device number name space is managed by individual drivers.

Also, there is not a 1:1 major/minor device number mapping between Linux and Solaris devices nodes and emulation of the functionality provided by a given Linux driver might involve using multiple Solaris drivers. For example, in Linux, both `/dev/null` and `/dev/random` are implemented using the same driver, so both of these devices have the same major number. In Solaris, these devices are implemented with separate drivers, each with its own major number.

Major/minor device numbers are exposed to Linux applications in multiple places. Some examples are:

- the `stat(2)` / `statvfs(2)` family of system calls
- the filesystem name space via `/dev/pts/*`
- `/proc/<pid>/stat`
- certain ioctls (specifically `LX_TIOCGPTN`)

One important question to answer is, is this device number translation actually necessary? Are major and minor device numbers actually consumed by Linux applications, and if so, how? As it turns out, the answer to this question is unfortunately yes. `glibc's ptsname()` does "sanity checks" of pts paths to makes sure they have expected dev_t values. These "sanity checks" make assumptions about expected major *and* minor device values.

Therefore, we will be required to provide a device number mapping mechanism. For the required devices listed earlier, this gives us:

Device	Solaris Driver	Linux Major/Minor
/dev/null	mm	1 / 3
/dev/zero	mm	1 / 5
/dev/random	random	1 / 8
/dev/urandom	random	1 / 9
/dev/tty	sy	5 / 0
/dev/console	zcons	5 / 1
/dev/ptmx	ptm	5 / 2 [1]
/dev/pts/*	pts	136 / * [2]

`[1]` `ptm` is a clone device, so this translation is tricky. Basically, the `/dev/ptmx` node in a native zone points to the clone device, but when an open is done on this device, the vnode that is returned corresponds to a `ptm` node (and not a clone node). This means that on a Solaris system, a stat of `/dev/ptmx` will return different `dev_t` values than an `fstat` (2) of an fd that was created by opening /`dev/ptmx`. On Linux, both of these operations need to return the same result. So once again, we are mapping multiple major/minor Solaris device numbers to a single Linux device number.

`[2]` For `pts` devices, there will be no translation done for device minor node numbers.

Linux device numbers are currently hard coded into dev_t translators. It was suggested that we move this mapping into an external XML file, to simplify the task of adding a new translator. This change would not significantly reduce the effort required, since the dev_t translators are only a small portion of the updates required to support a new device in a Linux zone. In general to add a new devices into a Linux zone the following things need to be done:

- dev_t translators need to be added for stat(2) calls.

- ioctl translators need to be added for any ioctls supported on the devices.

- mappings need to be added for the devices to rename them from standard Solaris names into Linux device names.

Since device names and semantics are different between Solaris and Linux, by we will not support adding devices Linux zones via the `zonecfg(1M)` device resource. (ie, `zonecfg(1M)` will prevent an administrator from adding "device" resources to a Linux zone.)

For the most part, applications are not likely to be sensitive to these device numbering issues. Searching Linux distro source rpms for references to st_rdev reveals lots of other references to hard coded device number references, but most of these references are in code that we would not expect to support in a Linux zone: utilities to manage cds, dvds, raid arrays, and so on.

3.7.3 Ioctl Translation Support

Ioctl Support - General Issues

A quick investigation shows that most of the required ioctl support isn't actually for devices at all. Most the necessary ioctls are for non-device filesystem nodes. Here's a list of the most obviously needed ioctls, broken up into categories of files that support these ioctls:

```
1) All file ioctls (regular files, streams devices, sockets, fifos):
    FIONREAD, FIONBIO

2) Streams file ioctls (Streams device, sockets, fifos):
    TCSBRK, TCXONC, TCFLSH, TIOCEXCL, TIOCNXCL, TIOCSPGRP,
    TIOCSTI, TIOCSWINSZ, TIOCMBIS, TIOCMBIC, TIOCMSET,
    TIOCSETD, FIOASYNC, FIOSETOWN, TCSETS, TCSETSW,
    TCSETSF, TCSETA, TCSETAW, TCSETAF, TIOCGPGRP, TCSBRKP

3) Socket ioctls:
    FIOGETOWN, SIOCSPGRP, SIOCGPGRP, SIOCATMARK,
    SIOCGIFFLAGS, SIOCGIFADDR, SIOCGIFDSTADDR,
    SIOCGIFBRDADDR, SIOCGIFNETMASK, SIOCGIFMETRIC,
    SIOCGIFMTU, SIOCGIFCONF, SIOCGIFNUM

4) Streams device - ptm
    TIOCGWINSZ, UNLKPT, LX_TIOCGPTN

5) Streams /w ttcompat module - pts
    TIOCGETD

6) Streams /w ldterm, ptem or ttcompat module - pts
    TCGETS

7) Streams /w ldterm or ptem module - pts
    TCGETA

8) Streams device - pts
    LX_TIOCSCTTY, LX_TIOCNOTTY
```

Most of these ioctls are streams ioctls, and since FIFOs and sockets are implemented via streams in Solaris, any FIFO or socket supports most of these ioctls. Of the 45 ioctls listed above, only 8 are actually device-specific ioctls.

This indicates that doing ioctl translations via layered drivers is not the best approach, since this would only address a minor subset of the total ioctls that need to be supported. Because supporting non-device ioctls will require the creation of a non-layered driver ioctl translation mechanism, it seems more appropriate to handle device ioctls via this same mechanism as well.

With this in mind, it's more interesting if the categories above are renamed based in terms of their vnode v_type and v_rdev values. If we do this, we get:

```
1) VREG, VFIFO, VSOCK, VCHR[ptm, pts, sy, zcons]
2) VFIFO, VSOCK, VCHR[ptm, pts, sy, zcons]
```

```
3) VSOCK
4) VCHR[ptm]
5, 6, 7, 8) VCHR[pts]
```

Supporting ioctls on these vnodes will require a switch table. In addition to the ioctl number, the translation mechanism must look at the type of the file descriptor an ioctl is targeting to determine what translation needs to be done. Hence, the translation layer will need to looking at the v_type and the major portion of v_dev associated with the target file descriptor. These fields are easily accessible from the kernel and are also available via st_mode and st_rdev from fstat(2). So this translation could occur either in the kernel or in userland.

One tricky part about this mapping is that we don't want to hard code the the major Solaris driver number into any translation code, since these number are allocated dynamically via `/etc/name_to_major` in the global zone. Therefore, device ioctl translators will be bound to specific Solaris drivers by their driver name. When an application attempts to perform an ioctl to a driver, the translation code will resolve the driver name to driver major number mapping. This translation code will not have any impact on how devices are managed in the global zone.

Ioctl support - nits

There are other more minor issues surrounding ioctl support that are worth mentioning.

Ioctl cmd symbols that represent the same ioctl command on Solaris and Linux can have different underlying symbol values. For example, TCSBRK on Solaris is 0x5405, while on Linux it's 0x5409. Any translation layers will have to be aware of the Linux ioctl values and translate them into Solaris ioctl values.

BrandZ will take an "opt-in" approach to ioctls. Only those ioctls that we have explicitly added support for will be executed. All others will return EINVAL. The alternative approach, simply passing the unrecognized ioctls through to Solaris proper, is risky for the following reasons:

- Nondeterministic behavior: Since ioctl cmd values can be different from Solaris, and devices on Solaris and Linux can support different behaviors, simply passing on unknown ioctls could result in nondeterministic device and/or application behavior.
- Inadvertent breakage and maintainability issues: If an application depends on certain ioctls that are not explicitly listed as supported in brand-specific code, then there is an increased chance that a developer might attempt to change the implementation of one of those ioctls without knowing that the ioctl is also being consumed by a brand, and thereby inadvertently break the brand. If all ioctls supported by a brand are explicitly listed in the brand support code, then when developers search for consumers of a given ioctl, they will see that the ioctl is exported for application consumption in a brand environment.

3.7.4 Device Minor Nodes and Path Management

Multiple zones (both native and non-native) will be sharing devices with the global zone and with each other. Therefor, we must ensure that this device sharing doesn't generate any security problems where one zone could affect another because of device sharing. Here's a look at the device nodes that will be present in zones and what types of risks/issues these devices present.

```
Device          Notes
------          -----
/dev/null       read-write, doesn't have any consumer state
/dev/zero       read-write, doesn't have any consumer state

/dev/random     Writable only by root, protected from root
```

```
                    zone writes via the 'sys_devices' privilege.
/dev/urandom        Writable only by root, protected from root
                    zone writes via the 'sys_devices' privilege.

/dev/tty            Pass through device, doesn't have any consumer state

/dev/ptmx           Cloning device. Each open results in a unique
                    minor node, so a minor node can only exist
                    in one zone at any given time.

/dev/pts/*          All minor nodes are visible in all zones.
                    Currently, this driver has been made zone
                    aware to prevent multiple zones from
                    accessing the same minor nodes concurrently.

/dev/console        Minor nodes of this device should not be
                    shared across different zones. Each
                    instance of this device in a zone should
                    have a unique minor number.   This is the
                    current behavior for native zones.
```

The other important aspect of device paths is how brand/zone-specific device paths are mapped into branded zones. Here is an example of some the global brand/zone-specific device paths and their path mappings as seen from within linux branded zones.

```
Global zone device path                 Linux zone device path
-----------------------                 ----------------------
/dev/zcons/<zone_name>/zoneconsole      /dev/console
/dev/brand/<brand_name>/ptmx            /dev/ptmx
/dev/brand/<brand_name>/dsp             /dev/dsp
/dev/brand/<brand_name>/mixer           /dev/mixer
```

3.7.5 Device-Specific Issues

3.7.5.1 /dev/console

Each branded zone will need its own console device, just like native zones today. Whenever possible, a brand should leverage the zcons driver and use it as the `/dev/console` device in a non-native zones. We have done this in the *lx* brand.

3.7.5.2 /dev/ptmx and /dev/pts

These devices need special management. When Linux applications access these devices, we need to do two things.

1. After an open of `/dev/ptmx`, we need to ensure that the associated `/dev/pts/*` device link exists (since it can be created asynchronously to ptmx opens), and that it has its ownership changed to match that of the process that opened this instance of `/dev/ptmx`.
2. We need to ensure that after any pts device is opened by a Linux application, the following modules get pushed onto its stream: `ptem, ldterm, ttcompat, and ldlinux`

In Solaris, issue 1 above is addressed by libc`ptsname() with the help of an `su` binary. Issue 2 above is addressed by the client applications that are allocating terminals (for example, this is done in `` xterm`spawn() ``.)

For BrandZ, we will address both issues by replacing the ptm driver with a self-cloning, layered driver in

Linux branded zones. Upon open, this layered driver will:

- Open an instance of the real /dev/ptmx.
- Wait for the corresponding /dev/pts/* node to be created.
- Set the permissions on the corresponding /dev/pts/* node.
- Set up the auto push mechanism (via kstr_autopush() to automatically push the required strmods onto the corresponding /dev/pts/* node when it is actually opened by a Linux application.

Upon final close of a Linux ptm node, this driver will remove the auto push configuration it created and close the underlying /dev/ptmx node that it opened.

3.7.5.3 /dev/fd/*

The entries in /dev/fd are not actually devices. The entries in /dev/fd/ allow a process access to its own file descriptors via another namespace. Thus, opens of entries in this directory map to re-opens of the corresponding file descriptor in the current process.

In Solaris /dev/fd is implemented via a filesystem. readdir(3C)s of /dev/fd might not return an accurate reflection of the current file descriptor state of a process, but opens of specific entries in the directory will succeed if that file descriptor is valid for the process performing the open.

In Linux, /dev/fd is implemented as a symbolic link to /proc/self/fd. This /proc filesystem directory is similar to the Solaris /proc/<pid>/fd directory in that it contains an accurate representation of a processes current file descriptor state. But aside from just providing access to the processes current file descriptors, on Linux the files in this directory are actually symbolic links to the underlying files referenced by the processes file descriptors. This is similar to the functionality in Solaris provided by /proc/<pid>/paths.

The most common uses for /dev/fd entries are for suid shell script and as parameters to commands that don't natively support I/O to stdin/stdout. Given these use cases it seems that a simple mount of the existing Solaris /dev/fd filesystem in the Linux zone should be sufficient for compatibility purposes.

3.7.5.4 Audio devices

Linux has two different audio subsystems: OSS and ALSA. To determine which audio subsystem to support we identified some common/popular applications that utilize audio and checked which subsystem they use. We found:

```
OSS only:
      skype, real/helix media player, flash, quake, sox

OSS or ALSA:
      xmms (selectable via plugins)
```

Our survey identified no popular applications that require ALSA, so we will only be supporting OSS audio.

Audio device access on Linux and Solaris is done via reads, writes, and ioctls to different devices.

```
OSS devices:
      /dev/dsp, /dev/mixer
      /dev/dsp[0-9]+, /dev/mixer[0-9]+

Solaris devices:
      /dev/audio, /dev/audioctl
      /dev/sound/[0-9]+, /dev/sound/[0-9]+ctl
```

Unfortunately, we can't simply map the Solaris /dev/audio and /dev/audioctl devices to /

`dev/dsp` and `/dev/mixer` devices in a Linux and expect the ioctl translator to handle everything else for us. Some of the reason for this are:

- The admin/user may not always want a Linux branded zone to have access to system audio devices.
- There may be multiple audio devices on a system each of which may support only input, only output, or both input and output. In Solaris a user can specify which audio device an application should access by providing a `/dev/sound/*` path to the desired device. But in the Linux zone the admin might want the Linux audio device to map to separate Solaris audio devices for input and/or output.
- Linux ioctl translation is done using dev_t major values. On Solaris opening `/dev/audio` will result in accessubg different device drivers based of what the underlying audio hardware is, and these different drivers may have different dev_t values. Hence, if audio devices were directly imported the dev_t translator would need to have knowledge of every potential audio device driver on the system, and as new audio drivers are added to the system this translator would need to be made aware of them as well.
- In Linux audio devices are character devices and support mmap operations. On Solaris audio devices are streams based and do not support mmap operations.

To deal with these problems the following components are provided:

- A way for the user to enable audio support in a zone via zonecfg.

  The user enables audio via zonecfg boolean attribute called "audio". (The absence of this attribute implies a value of false.) Adding this resource to a zone via zonecfg looks like this:

  ```
  --
  zonecfg:centos> add attr
  zonecfg:centos:attr> set name="audio"
  zonecfg:centos:attr> set type=boolean
  zonecfg:centos:attr> set value=true
  zonecfg:centos:attr> end
  zonecfg:centos> commit
  zonecfg:centos> exit
  --
  ```

- A new layered driver to act as a Linux audio device.

  This device will always be mapped into the zone. Linux can change the ownership of this device nodes as it sees fit.

  Since this layered driver will be accessing Solaris devices from within the kernel there will be no problems with device ownership. The Solaris audio devices will continue to be owned by whoever is logged in on the console but a Linux zone will also be able to access the device whenever necessary.

  The Solaris audio architecture includes an integrated audio output stream mixer so that multiple programs can open the audio device and output data at the same time. Unfortunately it's not possible to virtualize all audio features in this same manner. For instance there can only be one active audio recording stream on a given audio device at a time. Also it would be difficult to virtualize global mixer controls. Hence, by allowing a zone shared access to audio devices owned by the console user, it would be possible for zone users and the console user to compete for any non-virtualized audio resources. These limitations seem acceptable if it's assumed that the user who owns the console is the same user who is running Linux audio applications. We assume that this will probably be the common case for most audio enabled Linux zones. If these limitations are not acceptable then an admin always has the option of updating `/etc/logindevperm` to deny console users access to audio devices. (thereby giving a zone exclusive access to the device.)

This device is implemented as a character driver (instead of as a streams driver.) This allows it to more easily support Linux memory mapped audio device access. Theoretically, mmap device access could be simulated in the ioctl translation layer but this would add quite a bit of extra complexity to the translation system. It would require that the ioctl translation layer start to maintain state across multiple ioctl operations, something it currently does not do. It would also require user land helper threads and support for handling and redirecting signals that might get delivered to those threads.

- Provide hooks into the zone state transition brand callback mechanism to propagate zone audio device configuration to the Linux layered audio device.

  This is done via a program that opens the layered Linux audio device when the zone is booted, and uses ioctls to configure the device. When a zone is halted this same program is used to notify the driver that any previous configuration for a given zone is no longer valid.

This model provides each Linux zone with one audio device. There are no inherent limits in this approach that would prevent future support for multiple virtual audio devices in a linux zone. The `zonecfg` attribute configuration mechanism could be extended to allow the the administrator to specify what the mappings for additional audio devices should be. The audio layered driver could also be enhanced to export multiple audio and mixer minor nodes which would all be imported into any linux zone on the system.

3.7.5.4.1 Supporting Multiple Audio Devices

By default, when a Linux audio applications attempts to open `/dev/dsp` this access is mapped to `/dev/audio` under Solaris., and access to `/dev/mixer` is mapped to `/dev/audioctl`.

On a system with multiple audio devices, the admin can control which Solaris devices a Linux zone can access by setting two additional zonecfg attributes:

```
audio_inputdev = none | [0-9]+
audio_outputdev = none | [0-9]+
```

f `audio_inputdev` is set to none, then audio input is disabled. If `audio_inputdev` is set to an integer, any Linux application's open `/dev/dsp` for input is mapped to `/dev/sound/<audio_inputdev attribute value>`. The same behavior applies to `audio_outputdev` for Linux audio output accesses.

If `audio_inputdev` or `audio_outputdev` exist but the audio attribute is missing (or set to false) audio will not be enabled for the zone.

3.7.6 Networking

All network plumbing is done by the global zone – just as network plumbing is done today for native zones.) The Linux zone administrator should not configure the Linux zone to attempt to plumb network interfaces. Any attempt to plumb network interfaces or change network interface settings from within the Linux zone will fail. In most cases the failure will manifest as an `ioctl(2)` operation failingwith `ENOTSUP`.

Any networking tools that utilize normal tcp/udp socket connections (*e.g.,* `ftp, telnet, ssh`, etc.) should work. Any tools that require raw socket access (*e.g.,* `traceroute, nmap`) will fail with `ENOTSUP`. Utilities that query network interface properties (*e.g.,* `ifconfig`) will work, although attempts to change network configuration will fail.

3.7.7 Future Directions

Linux is moving away from `devfs` and torwards `udev`. `udev` consists of a userland daemon that consumes device hotplug events from the `sysfs` filesystem. (The `sysfs` filesystem seems similar to the

Solaris devfs filesystem found in /devices which exports a snapshot of all the physical and virtual devices currently accessible to the system.) udev has a rules database that is uses to translate sysfs device hotplug events nto mknods of devices in /dev. (This is very similar to how the devfsadmd link generators create /dev links on Solaris.) udev only exists on Linux 2.5 and later and currently the BrandZ *lx* brand only supports Linux 2.4.

If we attempt to support Linux 2.5 and later we will need to address this issue. The most likely way to handle this will be to simply disable udev. This approach should work for two reasons. First we're already supplying a fully populated /dev filesystem within the Linux zone so we don't need udev to create one for us. Second we're currently not supporting any hot-plugable devices within a Linux zone so there will be no hotplug events for which we need udev to generate /dev entries.

3.8 NFS Support

The BrandZ *lx* brand will have NFSv3/2 client support. NFSv4 is not supported in an *lx* zone, as the Linux distros we are targetting do not support NFSv4. As with any other non-global zone, *lx* zones cannot be NFS servers.

3.8.1 NFSv3/2 Client Support

NFS client support consists of two major components:

- A kernel filesystem module that can "mount" nfs shares from a server and service accesses to that filesystem via VOP_ interfaces
- lockd and statd rpc services to handle nfs locking requests.

On Solaris lockd is a userland daemon with a significant kernel component: klmmod. Most of the lockd functionality is actually implemented in the kernel in klmmod. The kernel lockd component also uses private undocumented interface (NSM_ADDR_PROGRAM) to communicate with statd.

On Linux lockd is actually entirely contained within the kernel. When the kernel starts up the lockd services, it creates a fake process that is visible via the ps command but lacks most normal /proc style entries.

Given the how closely integrated the separate components of the NFS client are on Solaris, and given that that most of the NFS client on Linux is in the kernel and there for not usable by the *lx* brand, the approach taken to support the NFS client in the *lx* brand was to simply run the Solaris NFS client within the *lx* zone.

Adding support for using the all Solaris NFS client components in a zone involved modifications in BrandZ, the *lx* brand, and base Solaris. Some of these areas and the modifications that were required are described below.

3.8.1.1 mount(2) support

The first component of support NFS client operations in an *lx* branded zone is translating Linux NFS mount system call requests into Solaris mount system call requests. This requires translating the arguments and options strings normally passed to the Linux into formats that the Solaris kernel is expecting.

3.8.1.2 Starting and stopping of the lockd and statd daemons

The versions of RedHat/CentOS Linux that the *lx* brand supports have a rc.d startup script that is used to start, stop, and check the status of statd and lockd. To support the execution of the Solaris versions of lockd and statd in a zone, after install we replace the Linux lockd and statd binaries with symlinks to scripts in /native/usr/lib/lx/brand that start the Solaris versions of the two daemons. The startup script has also been modified to successfully allow for the startup, stopping, and querying of status for lockd and statd. This approach preserves the existing techniques for administering NFS locking

under RedHat/CentOS in the *lx* brand.

`lockd` and `statd` are also run under different user and group ids under Linux than Solaris. The startup wrapper scripts pass command line options to lockd and statd to indicate what user and group should be used during normal operations.

3.8.1.3 Running native processes in a branded zone

Normally all processes in a non-native zone are branded non-native process. NFS client support is the only exception to this rule since it introduces the execution of two native Solaris processes into a non-native zone. To support the basic execution of a native Solaris process in the following steps were taken.

3.8.1.3.1 Brand aware exec path

The internal kernel exec path path (starting at `exec_common()`) was updated to have a new brand operation flag. The possible values for this flag are: clear (no special brand operation), brand-exec (indicates that the current process should become branded if the exec operation succeeds), and native-exec (indicates that the current process should become a native process if the exec operation succeeds.)

Note, these extended exec flags are not accessible through the normal exec() system call path. The normal exec() system call path always defaults to this flag being clear. To change the branding of a process via exec, a special brand operation (invoked via the `brandsys(2)` system call) is used.

3.8.1.3.2 Chroot

One problem with running native Solaris binaries in a branded zone is that both the native binary and native libraries that they use expect to be able to access native Solaris paths and files that may not exist inside a branded. Rather than implementing a path mapping mechanism to re-direct filesystem accesses for native binaries to paths into /native, during the startup of these daemons we do a chroot("/native"). We've also ensured that there is enough of the native Solaris environment created in /native to allow `lockd` and `statd` run properly.

3.8.1.4 Alowing `lockd` and `statd` to communicate with Linux services/interfaces within the zone

`lockd` and `statd` are fairly self contained but they do require access to certain services for which the native Solaris versions won't be available in a zone. An audit of `lockd` and `statd` reveal that these daemons depend on access to the following services:

- naming services (via libnsl.so)
- syslog (via libc.so)

Normally, these daemons simply access these services via local libraries. These libraries in turn use local files, other libraries, and various network based resources to resolve requests. In a branded zone most of these resources will not be available. For example, we can't expect the Solaris libnsl.so library to know how to parse Linux NIS configuration files.

To handle these requests we need to be able to leverage existing Linux services and interfaces. This requires translating certain Solaris `lockd` and `statd` services requests into Linux service requests, and then translating any results back into a format that Solaris libraries and utilities are expecting. In the *lx* brand we've decided to call this process of translating service requests *thunking*. (akin to a 32-bit OS calling into 16-bit BIOS code.) To service these requests we have created a thunking layer which translates Solaris calls into Linux calls.

This thunking layer works as follows:

1. When `lockd` or `statd` make a request that requires thunking, this request ends up getting directed into a library in the process called lx_thunk.so (the mechanism used to direct requests into this library varies based of the type of request being serviced and is discussed further below).

2. The lx_thunk.so library packs up this request and sends it via a door to child Linux process called lx_thunk.
3. If the lx_thunk process does not exist then the lx_thunk.so library will `fork(2)/exec(2)` it.
4. The lx_thunk process is a one line /bin/sh script that attempts to execute itself and is executed in a Linux shell. When the brand emulation library (lx_brand.so) detects that it is executing as the lx_thunk process and it is attempting to re-exec itself, the library takes over the process and sets itself up as a doors server.
5. When the lx_thunk process receives a door request from lx_thunk.so library in a native process, it unpacks the request and uses a Linux thread to call invoke Linux interfaces to service the request.
6. Once it is done servicing the request it packs up any results and returns them via the same door call that it received the request on.

This thunking layer means that now the *lx* brand is dependent upon Linux interfaces so we need to worry about Linux interfaces changing and breaking the lx_thunk server process. To help avoid this possibility, most the Linux interfaces that we've chosen to use are extremely well known and listed in the glibc ABI. All of the interfaces are used by many applications outside of glibc. Here are the Linux interfaces currently used by the lx_thunk process:

- `gethostbyname_r`
- `gethostbyaddr_r`
- `getservbyname_r`
- `getservbyport_r`
- `openlog`
- `syslog`
- `closelog`
- `__progname`

Also worth mentioning is the means by which service requests that require thunking are directed to lx_thunk.so. To intercept name service requests the *lx* brand is introducing a new libnsl.so plugin name-to-address translation library. Libnsl already supports name-to-address translation plugin libraries that can be specified via netconfig(4). For *lx* branded zones there will be a custom netconfig file installed into /native/etc/netconfig that will instruct libnsl.so to redirect name service lookup requests to a new library called lx_nametoaddr.so. This library will in turn resolve name service requests using private interfaces exported from the thunking library, lx_thunk.so.

3.8.1.5 rpcbind vs portmap

`lockd` and `statd` are both rpc based services. Therefore, when they start up they must register with an rpc portmapper. This registration is done via a standardized rpc protocol. The difficulty with this registration is that there are multiple versions of the protocol.

Initially the protocol was called the portmapper protocol and only supported IP based transports. This is the version of the protocol that RedHat/CentOS support. Solaris long ago upgraded the version of the protocol it supports to be rpcbind. rpcbind supports registrations on non-ip based transports. The problems faced by BrandZ are that all the libnsl.so interfaces designed for talking to a local portmapper assume that the local portmapper supports the rpcbind protocol. In the case of an *lx* branded zone we're actually running the Linux portmapper which doesn't support the rpcbind protocol. To work around this a new command line flag has been added to `lockd` and `statd` to indicate that they should attempt to to use the portmapper protocol instead of the rpcbind protocol for registering rpc services. Also, new private interfaces have been added into libnsl.so to allow it to support communication via the portmapper protocol instead of the rpcbind protocol.

3.8.1.6 Privilege retention for `lockd` and `statd`

On Solaris, `lockd` and `statd` and privilege aware daemons and upon startup they drop all privilege they

won't need for normal execution. When running in a Linux zone, these daemons need additional privileges so that they can `chroot(2)`, `fork(2)`, `exec(2)`, and run the lx_thunk processes. So the lx_thunk.so library which is preloaded into these processes also prevents them from dropping the following privileges:

- `proc_exec`
- `proc_fork`
- `proc_chroot`
- `file_dac_read`
- `file_dac_write`
- `file_dac_search`

3.8.2 Automounters

Linux supports two automounters: "amd" and "automount".

amd is implemented as a userland NFS server. It mounts NFS filesystems on directories where it will provide automount services, and specified itself as the server for these NFS filesystems. To support amd only required adding translation support for all the Linux system call mount options it expects to work.

automount, the more common (and often default) automounter, is substantially more complex than amd. automount relies on a filesystem module called autofs. Upon startup, automount mounts the autofs filesystem onto all automount controlled directories. As an option to the mount command it passes a file descriptor that indicates a pipe will be used to send requests to the automount process. automount listens for requests on this pipe. When it gets a request, it looks up shares via whatever name services are configured, executes `mount(2)` system calls as necessary, and notifies the autofs filesystem that a request has been serviced. The exact semantics of the interfaces between automount and autofs are versioned and appear to differ based of the Linux kernel version. To support automount the *lx* brand will introduce a new filesystem module called `lx_afs`. When the automount process attempts to mount the autofs filesystem we will instead mount the `lx_afs` filesystem which will emulate the behavior one specific version of the autofs filesystem.

3.9 Debugging and Observability

3.9.1 Ptrace

In order to support Linux `strace` and `mdb`,, it will be necessary to duplicate almost the full functionality of the `ptrace(2)` system call. Rather than trying to implement this complex interface in the kernel, we will implement it in userland on top of a native `/proc`, which will be mounted at `/native/proc`. It is worth noting that the Solaris `/proc` already has a ptrace compatibility flag, which provides many of the semantics we want (e.g., interaction through wait(2)).

Two difficult parts about `ptrace(2)` are tracing Linux system calls and attaching to existing processes.

To implement system call tracing, we want to stop the program in userland before the interposition library has done any work. To do this, we introduce a per-brand scratch space in the ulwp_t structure, similar to that used by the DTrace PID provider. When we want to trace a system call enter or exit, we set this flag from within the kernel via a brand-specific system call. In the trampoline code, we check the status of this flag and issue another brand-specific system call that will stop us with an appropriately formed stack. Note that it's generally not possible to "hide" the interpositioning library when it comes to signals. Besides trying to figure where we are in the brand library (if we're in it at all), we would probably do more harm than good when trying to hide this behavior.

When a debugger attaches to another process using PTRACE_ATTACH under Linux, it actually becomes the parent for the target process in all ways - except that getppid() still returns the original value. Since this is an implementation detail and an undesirable pollution of the Solaris process model, we will instead add a

per-brand child list that the wait(2) system call will also check. When we attach to a process, we add it to the debugger's brand list.

Finally, there are also significant issues around multithreaded apps. Trying to reconcile the differences between the Linux and Solaris threading models as well as the differences between `ptrace(2)` and `/proc` appears to be a nearly intractable problem. For at least the initial BrandZ release, we do not expect to be able to support `ptrace(2)` for multithreaded applications.

3.9.2 librtld_db: MDB and Solaris ptools Support

Since the Linux binaries and applications are standard ELF objects, Solaris tools are able to process them in essentially the same way that Solaris binaries are processed. The main objective in doing so is to retrieve symbols and thereby aid debugging and observability.

mdb and the ptools (pstack, pmap, etc) use interfaces provided by `librtld_db` to debug live processes and core files. `librtld_db` discovers ELF objects which have been mapped into the target's (a target can be either a live process or a core file) address space and reports these back to the `librtld_db` client. `librtld_db` library understands enough of the internals of the Solaris runtime linker to iterate over the linker's private link maps and process the objects it finds. `librtld_db` allows our tools to debug the Solaris portions of a branded process (such as the brand library, libc, etc.), but they cannot understand any Linux objects that are mapped into our address space, because the Solaris linker only has Solaris objects on its link maps.

In order to give Solaris tools visibility into Linux binaries, a brand helper-library framework is implemented in `librtld_db`. When `librtld_db` is asked to examine a branded target process or core file, it uses the AT_SUN_BRANDNAME aux vector to get the brand name of the process. Once it has the brand name, it `dlopen(3C)`s a shared object from: `/usr/lib/brand/`*`brandname`*`/`*`brandname`*`_librtld_db.so.1`

Brand helpers must have a vector of operations called from critical `librtld_db` hooks (e.g. helper initialization, loaded object iteration, etc).

Once loaded, the helper library is responsible for finding any brand-specific information it needs, such as the brand equivalent of `AT_SUN_LDDATA` (used by `librtld_db` to find the Solaris link maps), and preparing to return details about the objects loaded into the address space by the brand's linker.

When a client of `librtld_db` operating on a branded process, asks to know what objects are loaded in the target, `librtld_db` walks the Solaris link maps and iterates over each object it finds there, handing information about each to the caller. It then calls down into the helper library, which does the same for the brand-specific objects used by the target. In this manner, the client of `librtld_db` does not need any modification to operate on branded processes, nor does it need any special knowledge of brands; all data is passed via established `librtld_db` APIs.

Because mdb and the other ptools are Solaris binaries, they may not run inside a non-Solaris branded zone. They must therefore run from the global zone and may operate on individual processes within the branded zone.

3.9.3 Core Files

Since dumping core is handled by the kernel, it will produce Solaris core files that cannot be understood by the native Linux tools. While it would be possible to provide a brand-specific tool to convert between Solaris core files and Linux core files, it will not be provided as part of the initial release. Unless there is significant customer demand for this tool, it will not likely be included in subsequent releases either. Our preferred method for examining Linux core files is through mdb.

3.9.4 DTrace

With the addition of brand support in librtld_db.so, the DTrace PID provider is able to instrument Linux processes correctly with no additional work.

We have also added an lx-syscall provider, which allows DTrace to trace the Linux system calls issued by the application.

3.10 /proc

The lx brand will deliver a lx_proc kernel module that provides the necessary semantics of a Linux /proc filesystem.

Linux tends to use /proc as a dumping ground for all things system-related, although this is reduced by the introduction of sysfs in the 2.6 kernel. Thus, we will not be able to emulate a large number of elements from within a zone. Examples of unsupported functionality include physical device characteristics, the USB device tree, acccess to kernel memory, etc. Because various commands expect these files to be present, but do not actually act on their contents, a number of these files will exist but otherwise be empty.

We are able to emulate the per-process directories completely. The following table shows the support status of other /proc system files.

File	Supported	Description
`cmdline`	empty	Kernel command line options
`cpuinfo`	empty	Physical chip characteristics
`crypto`	no	Kernel crypto module info
`devices`	empty	Major number mappings
`dma`	empty	???
`driver/*`	no	Per-driver configuration settings
`execdomains`	no	???
`fb`	no	Framebuffer device
`filesystems`	empty	Available kernel filesystems
`fs/*`	no	???
`ide/*`	no	Kernel IDE driver info
`interrupts`	empty	Kernel interrupt table
`iomem`	no	I/O memory regions
`ioports`	empty	I/O port bindings
`irq/*`	no	???
`kcore`	empty	Kernel core image
`kmsg`	empty	Kernel message queue
`ksyms`	no	Kernel symbols
`loadavg`	yes	System load average
`locks`	no	???
`mdstat`	no	???
`meminfo`	yes	Virtual memory information

misc	no	???
modules	no	Random module information
mounts	yes	System mount table
mpt/*	no	???
mtrr	no	???
net/*	empty	Various network configuration
partitions	empty	Parition table
pci	no	PCI device info
scsi/*	no	SCSI device info
slabinfo	no	Kernel Slab allocator stats
stat	yes	General system wide statistics
swaps	no	Swap device information
sys/*	no	???
sysrq-trigger	no	???
sysvipc/*	no	System V IPC statistcis
tty/*	no	???
uptime	yes	System uptime
version	empty	Kernel version

4 Deliverables

4.1 Source delivered into ON

Below is a summary of the new sources being added to ON.

4.1.1 New code for brand support:

usr/src/lib/libbrand	support for reading the new XML files defining brands and virtual platforms
usr/src/lib/brand/native/	virtual platform and template files for non-branded zones
usr/src/uts/common/os/brand.[ch]	manages the brandx framework, tracks loaded brands, and so on.
usr/src/uts/common/syscall/brandsys.c	the brandsys() system call

4.1.2 New directories created as holding areas for per-brand code:

usr/src/lib/brand	for the userspace components of brands
usr/src/uts/common/brand	for platform-independent brand code
usr/src/uts/intel/brand	for Intel-specific brand code
usr/src/uts/sparc/brand	for SPARC-specific brand code

4.1.3 For the Solaris n-1 'test' brand:

usr/src/pkgdefs/SUNWsn1rint	The package containing all the kernel-space pieces of the Solaris n-1 brand
usr/src/pkgdefs/SUNWsn1uint	The package containing all the user-space pieces of the Solaris n-1 brand
usr/src/lib/brand/sn1/sn1_brand	contains the emulation code and the zones integration support.
usr/src/uts/common/brand/sn1	source for kernel-level Solaris n-1 support
usr/src/uts/intel/sn1_brand	where the Solaris n-1 Intel brand module is built
usr/src/uts/sparc/sn1_brand	where the Solaris n-1 SPARC brand module is built

4.1.4 For the lx brand:

usr/src/pkgdefs/SUNWlxr	The package containing the kernel-space components of the lx brand
usr/src/pkgdefs/SUNWlxu	The package containing the user-space components of the lx brand
usr/src/lib/brand/lx/lx_brand	contains the emulation code, as well as the zones integration support. (install scripts and so on)
usr/src/lib/brand/lx/librtld_db	the rtld_db plugin for the lx brand
usr/src/uts/common/brand/lx	source for kernel-level lx support
usr/src/uts/common/brand/lx/procfs	source for the lx /proc
usr/src/uts/common/brand/lx/dtrace	source for the lx syscall provider
usr/src/uts/intel/lx_brand	where the lx brand module is built
usr/src/uts/intel/lx_syscall	where the lx syscall provider is built
usr/src/uts/intel/lx_proc	where the lx /proc filesystem is built

4.2 Components installed with the lx brand

4.2.1 Userspace components:

usr/lib/brand/lx/lx_brand.so.1	lx emulation library
usr/lib/brand/lx/[amd64]/lx_librtld_db.so.1	lx rtld_db plugin
usr/lib/brand/lx/[amd64]/lx_thunk.so	lx thunking support
usr/lib/brand/lx/[amd64]/lx_nametoaddr.so.1	Wrapper for Solaris name services
usr/lib/brand/lx/config.xml	Definition of the lx brand
usr/lib/brand/lx/platform.xml	Definition of the lx virtual platform

usr/lib/brand/lx/SUNWblank.xml	Values for a blank zone configuration
usr/lib/brand/lx/SUNWdefault.xml	Values for a default zone configuration
usr/lib/brand/lx/install	The scripts needed to install supported distros
usr/lib/brand/lx/lx_audio_config	Audio device configuration
usr/lib/brand/lx/lx_lockd	Wrapper for Solaris lockd
usr/lib/brand/lx/lx_statd	Wrapper for Solaris statd
usr/lib/brand/lx/lx_native	Wrapper for native Solaris processes
usr/lib/brand/lx/lx_boot	Boot-time script
usr/lib/brand/lx/lx_halt	Halt-time script
usr/lib/brand/lx/lx_install	Main lx install script
usr/lib/devfsadm/linkmod/SUNW_lx_link_i386.so	Brand-aware devfsadm link module

4.2.2 Kernel components:

kernel/brand/[amd64]/lx_brand	lx brand kernel module
kernel/drv/[amd64]/lx_audio	lx Linux-compatible audio driver
kernel/drv/[amd64]/lx_systrace	lx syscall provider
kernel/drv/[amd64]/lx_ptm	lx Linux-compatible ptm driver
kernel/fs/[amd64]/lx_afs	lx automounter support
kernel/fs/[amd64]/lx_proc	lx /proc filesystem
kernel/strmod/[amd64]/ldlinux	lx Linux-compatible ldterm module