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 (as well as `libzonecfg`) is modified to be brand-aware. There is a new option to the 'create' command that allows a zone to be created with a specific brand:

```
   zonecfg:myzone> create -B lx
```

Once a zone has been assigned a brand, that brand cannot be changed or removed.

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, configure, and boot the zone. It also identifies the name of the kernel module (if any) that provides kernel-level functionality for the brand. 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">
        <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>
        <modname>lx_brand</modname>
        <platform name="platform.xml" />
</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</>
```

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 substitutions '%z' (for zonename) and '%Z' (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 an XML file that is separate from the main brand config file, `config.xml`. 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 does Solaris-specific mounts (`/proc`, `/system/contract`, etc.) that will need to be per-brand. We might 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 platform configuration file:

```
    <filesystem spec="/proc" dir="/proc" fstype="proc" />
    <filesystem spec="/usr" dir="/native/usr" fstype="lofs" />
```

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" />

        <device match="zero" />
        <device match="null" />

        <device match="tty" />
        <device match="tcp" />
        <device match="tcp6" />
        <device match="udp" />

        <device match="udp6" />

        <!-- 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" />

        <!--
                /dev/console can't be a symlink because this breaks
                login security checks via /dev/securetty
        -->

        <device match="zconsole" name="console" />
```

2.1.4 Booting a Branded Zone

In addition to the install-time command attribute, the configuration file can also provide 'preboot' and 'postboot' scripts to be

run when a zone is booted. This allows a brand to perform any needed setup tasks before and after the virtual platform has been brought online, giving the brand complete leeway in how the zone is brought online.

2.1.5 Spawning init

To satisfy `smf(5)` requirements, `init(1M)` must be spawned from the kernel, so that the kernel can control its restart. We will run the Linux `init(1M)` within the zone, so the kernel must be given the name and location of the brand's `init(1M)` (or equivalent) binary. Since RHAS also places its `init` in /sbin, we can defer the implementation of a brand-specific `init(1M)` location.

Linux does not allow init to be killed, even with SIGKILL. Since the Linux `init` process does not expect to be killed, and since we cannot just ignore SIGKILL, there is a conflict that we must resolve. The best solution is probably to again interpose on the `kill()` system call, and only send good signals to init in the first place. On Linux, `init` will die if it somehow does manage to catch a `SEGV`, leaving the OS in a failure state that requires rebooting to clear.

2.1.6 Shutdown

The standard `sysvinit` Linux init package operates in much the same way as our `init` did prior to the introduction of `smf(5)`. The 'init N' form of the command writes to the init FIFO, which happens to be `/dev/initctl`. We make use of the postboot configuration script to create this link in the global zone before the zone boots up.

The `init` command then does a `kill(1, SIGHUP)` to notify the running init process that there has been a change of runlevel. We will interpose on the `kill(2)` syscall and translate PID 1 into whatever the zone-local PID is.

To actually power off or reboot the system, `init` calls the `reboot()` function with 'magic numbers' defined in `sys/reboot.h`. This function translates directly to the `reboot` system call. This call, and the associated magic numbers, will be translated into an appropriate `uadmin()` call.

The Linux `simpleinit` program works in a nearly identical fashion. The main differences are in how it processes `/etc/rc` scripts.

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;
};
```

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 the future, 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 develop radically new brands will be limited.

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.

2.2.3 Other Interposition Points

Other interposition points will be placed in the code paths for process creation/exit, thread creation/teardown, signal delivery,

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_setregs)(struct uarg *);
        void    (*b_lwp_setrval)(klwp_t *, int, int);
        void    (*b_shmexit)(struct proc *);
        int     (*b_initlwp)(klwp_t *);
        void    (*b_forklwp)(klwp_t *, klwp_t *);
        void    (*b_freelwp)(klwp_t *);
        void    (*b_lwpexit)(klwp_t *);
        int     (*b_exec)(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. Should be renamed b_forkproc().
- `b_proc_exit`: Called at process exit time to free any per-brand state.
- `b_setregs`: Called at `exec()` time to allow a brand to set the initial register state seen by the new process image.
- `b_lwp_setrval`: set the syscall() return value for the newly created lwp before the creating fork() system call returns.
- `b_shmexit`: Called when a shared memory segment is detached from an exiting branded process.
- `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_exec`: Called to load a brand-specific executable.

The x86-specific ops vector is:

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

The first 4 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_fasttrap)(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 10' brand used for basic testing on SPARC platforms.

These ops vectors are sufficient for the initial Linux brand. Adding support for a new Linux distribution based on the 2.4 kernel can probably also be done without modifying this interface. However, it is likely that adding any new brand that is substantially different from the initial Linux brand will require additional interposition points. For example, adding support for a 2.6-based Linux distribution could require modifications. 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. Debian 3.1 also uses this version of glibc, so adding support for that distro to *lx* should be straightforward.

The further removed one gets from that version of libc, the less likely it is that the *lx* brand will be able to support that distro. For example, RHEL 4 is based on glibc 2.4.7. This represents a significant change to glibc, and thus will likely require the creation of a new brand. The new brand will share most of the source code with the *lx* brand, but it will most likely have to be built and packaged as a separate brand.

Finally, it should be remembered 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.

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.

3.3. Execution of Linux Binaries

When executing Linux binaries, we follow the same 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 a non-native ELF binary is execed inside a branded zone, the brand's exec handler is given control. The `lx` brand-specific exec 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_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
```

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 3.8 respectively.

Once the support library has finished initialization, it fixes up the 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

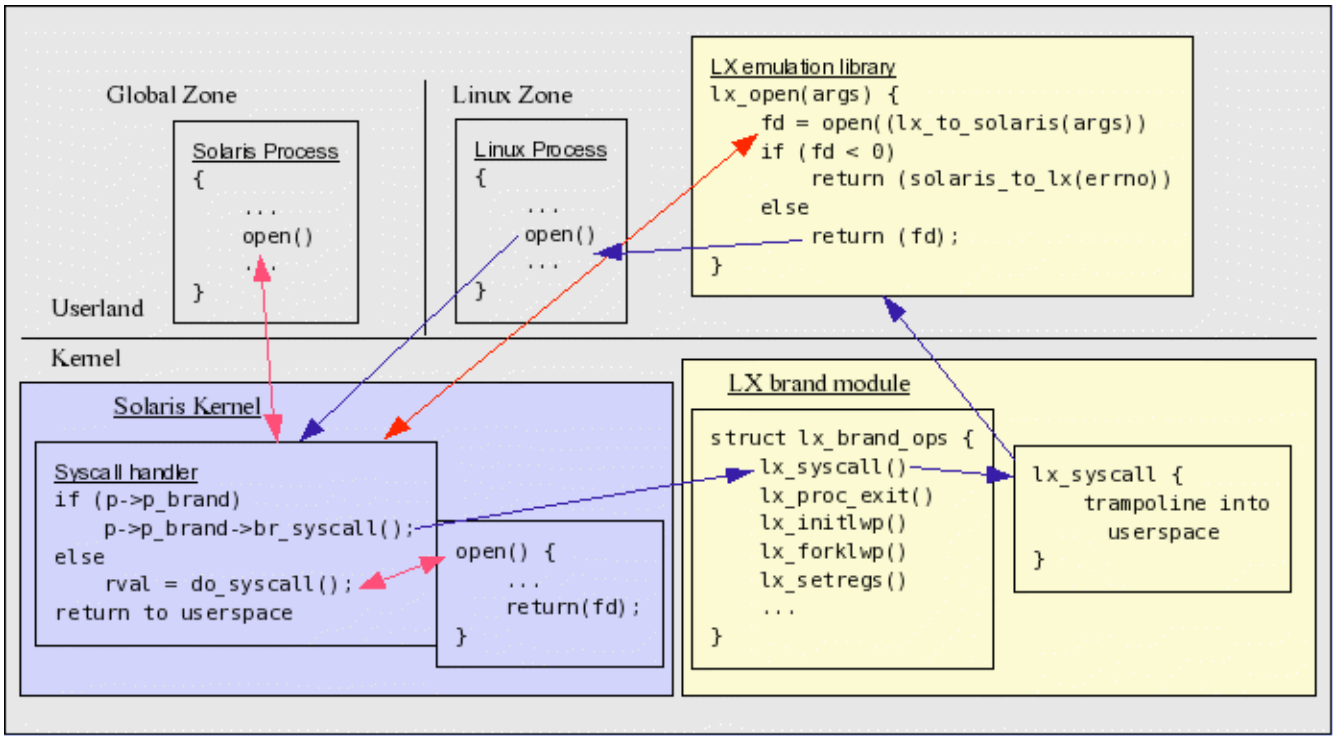
model, the trap handler in the kernel is nothing more than a trampoline, redirecting the execution of the system call to a library in userspace. The library performs any necessary data mangling and then calls the proper native system call. Ideally, this method would require almost no code in the kernel 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:



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().

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. When all four flags are clear then the clone is equivalent to a fork, and when they are all set then 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.

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.

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 call will be implemented in libc, where it wil be available to any application.

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

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_t`s, 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.

Conversely, 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, `setsigaction()`:

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

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.

3.6.2 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 pushes actual code to call a specific Linux system call, `sigreturn(2)`, onto the signal handler's stack. Thus, when a Linux signal handler completes execution, instead of returning through what would in libc be a call to `setcontext(2)`, the `sigreturn(2)` Linux system call is responsible for accomplishing much the same thing.

This trampoline code looks something 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 trampoline code
Linux signal number
Pointer to Linux siginfo_t
Pointer to Linux ucontext_t
Linux ucontext_t
Linux fpstate
Linux siginfo_t
Trampoline code to call the lx_sigreturn() syscall

BrandZ takes the approach of intercepting the Linux `sigreturn(2)` system call in order to turn it into the return through the libc call stack that Solaris expects. This is done by the `lx_sigreturn()` and `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. The proper value is saved when the thread context is originally saved, but prior to BrandZ, code existed in libc to **force** the value to that used by libc before the `setcontext()` is called within `call_user_handler()`.

For BrandZ, the code that did so has been removed. While perhaps making faults due to bad user context values for %gs harder to debug (as such bad values will now make applications appear to fault deep within Solaris' libc), the versatility to properly restore custom %gs values seems worth the trade-off.

3.7. Device and `ioctl()` support

3.7.1 Determining Which Devices to Support

An initial inspection of the devices indicates that the following devices will probably be 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 probably 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. It's probably safe to assume that most Linux applications have long ago been ported to use pts/ptmx terminal devices and we shouldn't bother supporting these old devices unless we actually discover an application that needs them. A quick look around a Linux 2.4 system didn't reveal any applications that were actually using these devices. Also, these have officially been unsupported since Linux 2.1.115. (However, it should be noted that these devices do still exist on Red Hat 2.4 systems.)

3.7.1.1 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
```

But if we look in `Documentation/devices.txt` we see that all these devices described as 'iBCS-2 compatibility devices'. Hence, we probably don't need to support 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) seems to all be done via ioctls to sockets. Reporting status of interfaces (via ifconfig) seems to be done by socket ioctls and accessing files in /proc/net/. Finally, network accesses (telnet and ping) appear to all be done via socket calls.

This seems to indicate 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 support should probably 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`)

Any device number mapping mechanism will have to be accessible to all BrandZ emulation code that provides these interfaces to Linux applications. This could be difficult since `/proc` emulation will probably have to be done in the kernel while we may wish to do ioctl emulation in userland. In the end, this mapping mechanism will probably have to be in the kernel and we might have to provide interfaces to query current mappings to userland emulation code.

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.

So unfortunately, 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]

Notes:
[1] ptm is a clone device, so this translation is tricky.
    Basically, the /dev/ptmx node in a native zone actually
    points to the clone device.  But when an open is done
    on this device, the vnode that is returned actually 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 should probably be no translation done
    for device minor node numbers.
```

3.7.3 Ioctl Translation Support

3.7.3.1 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.

What this seems to indicate is that doing ioctl translations via layered drivers might not be 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 may be 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 probably involve 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.

The only tricky part about this determination is that we probably don't want to hard code the the major Solaris device number into any code since this number is assigned via `/etc/name_to_major` in the global zone. Therefor, devices should probably be specified by their Solaris driver names. Then, when the first instance of a brand is initialized, this name can be translated into a major device number.

3.7.3.2 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.

One final note: 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       write-only, doesn't have any consumer state
/dev/zero       read-only, 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 seen from the global zone. Currently, the only brand/zone-specific device that exists is the zcons console device. Here are examples of zcons device paths that could exist in the global zone (with two native zones booted):

```
/dev/zcons/<zone1_name>/masterconsole ->

        /devices/pseudo/zconsnex@1/zcons@0:masterconsole

/dev/zcons/<zone1_name>/zoneconsole ->

        /devices/pseudo/zconsnex@1/zcons@0:zoneconsole

/dev/zcons/<zone2_name>/masterconsole ->
        /devices/pseudo/zconsnex@1/zcons@1:masterconsole

/dev/zcons/<zone2_name>/zoneconsole ->
        /devices/pseudo/zconsnex@1/zcons@1:zoneconsole
```

These device paths are very zcons centric and don't really extend well if we attempt to introduce any new brand-specific devices. If we decide to add any new brand-specific devices, then these paths should probably be changed. For instance, if we add a Linux brand-specific driver that layers on top of `/dev/ptmx`, then the following device paths might make more sense:

```
        /dev/zones/<brand>/<zone_name>/masterconsole ->
                /devices/pseudo/zonesnex@1/zcons@0:masterconsole

        /dev/zones/<brand>/<zone_name>/zoneconsole ->

                /devices/pseudo/zonesnex@1/zcons@0:zoneconsole

        /dev/zones/<brand>/ptmx ->

                /devices/pseudo/ptm_linux@0:ptmx
```

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

For BrandZ, there two possible approaches for solving these problems.

The first is to continue with the initially prototyped approach and do both of these things post open in an interception layer. For the ptm device (issue 1 above), this interception layer would have to be in the kernel since it involves changing the ownership of a device and `/dev` is mounted as read only in zones. The post open pts device processing (issue 2 above) could be done in the kernel or in user and. This approach doesn't seem that great since its implementation is spread across both userland and the kernel. It also involves post-processing all opens to determine if additional work is necessary.

A preferred approach is to replace the ptm driver with a layered driver in Linux branded zones. The current ptmx driver could be replaced with a self-cloning layered driver in the Linux zones. Upon open, this layered driver would:

- 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 would 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. The only known consumers of `/dev/fd` entries are suid shell scripts for redirection of file descriptors.

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`.

Since the only known consumers of `/dev/fd` are suid shell scripts, which are a huge security risk and hence a rarity, it seems that a simple mount of the existing Solaris `/dev/fd` filesystem in the Linux zone should be sufficient for compatibility purposes. Of course, it is possible that other Linux applications exist that utilize some of the additional functionality of the Linux `/dev/fd` implementation that isn't available in Solaris. (For example, `/dev/fd` on Linux provides an accurate reflection of the current fd state of the process and allows applications to determine file descriptor to file path mappings.) If such applications are discovered, then we might need to revisit this strategy.

3.7.5.4 audio devices

Linux has two different audio subsystems OSS and ALSA. To determine which audio subsystem to support we need to look at which subsystem is being used by application we wish to support. Randomly selecting some common/popular applications that utilize audio and checking with subsystem they use under Linux reveals the following:

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

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

Luckily there are no ALSA only applications. So if we implement support OSS we immediately enable all the applications above. Hopefully this subset of applications is indicative of other Linux applications that use audio. The rest of this discussion assumes that we're only providing support for 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 always 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.
- By default Solaris audio device ownership is dynamic and controlled via `/etc/logindevperm`. When a user logs into the console the ownership of the audio devices nodes is changed to grant that user access to the devices. I am not sure if/how Linux attempts to manipulate audio device ownership.
- Linux ioctl translation is done based of driver dev_t major values and on Solaris opens of `/dev/audio` will result in opens of different device drivers based of what the underlying audio hardware is. 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 proposed:

- Provide a way for the user to enable audio support in a zone via zonecfg The user could enable audio via zonecfg boolean attribute called "audio". (The absence of this attribute would lead to an assumed value of false.) Adding this resource to a zone via zonecfg would look 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
  --
  ```

  By default when a Linux audio applications attempts to open `/dev/dsp` this access will be mapped to `/dev/audio` under Solaris. (Linux application access to `/dev/mixer` will be mapped to `/dev/audioctl`.)

  To allow an admin to control which Solaris devices a Linux zone can send input/output to we can provide two additional

attributes that have string values:

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

If audio_inputdev is set to none, then audio input will be disabled. If audio_inputdev is set to an integer, then when a Linux application attempts to open `/dev/dsp` for input this access will be mapped to `/dev/sound/<audio_inputdev attribute value>`. The same behavior will apply 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.

Currently there is no mechanism outside of zonecfg itself for verifying zone attributes. So if a user specifies invalid types or values for these attributes there is no way to alert them during the zone configuration stage. (Later when attempting to boot the zone we will have an opportunity to verify these attributes and can fail to boot the zone if they are invalid.)

- Create 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.

  Luckily, 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 matter, 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 physical person as the 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. (there by giving a zone exclusive access to the device.)

  This device would be implemented as a character driver (instead of as a streams driver.) This will allow 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 will be done via a program that will open the layered Linux audio device and via ioctls notify it of the current zone audio configurations when a zone is booted. When a zone is halted this same program can be used to notify the driver

that any previous configuration for a given zone is no longer valid.

3.8 Debugging and Observability

3.8.1 Ptrace

In order to support Linux `strace`, it will be necessary to duplicate the full functionality of the `ptrace` system call. Rather than trying to implement this giant wad into the kernel, it would make more sense to implement it in userland on top of a native `/proc`. Note that this will require mounting a native version within the non-global zone (perhaps as `/proc/native`, though that would pollute the directory namespace), but that shouldn't be a problem. We already have a ptrace compatibility flag in our own /proc implementation which should give us the majority of the semantics (interaction through wait(2)).

The two difficult parts about ptrace are Linux system call tracing and attaching to other processes. In order to do system call tracing, we want to stop the program in userland before the interposition library has done any work. To do this, we will have some per-brand scratch space in the ulwp_t structure, similar to the 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 notice 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 will probably do more harm than good when trying to hide this behavior.

When a debugger attaches to another process using PTRACE_ATTACH, it actually becomes the parent for the target process in all ways except getppid() in the target process 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 make sure to add it to the debugger's brand list.

There are also potential issues around multithreaded apps. We will not be able to support all forms of ptrace. For example, one thread examining another thread in the same processs. Instead, we will implement the most common methods, which is either all threads or a single thread. Some of the subtle issues here still need some investigation.

3.8.2 librtld_db: MDB and Solaris ptools Support

Since the Linux binaries and applications are standard ELF objects, Solaris tools should be 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, etc) use interfaces provided by librtld_db to debug live processes and core files. The librtld_db library understands enough of the internals of the Solaris runtime linker to iterate over the link maps and process the objects it finds mapped into a process address space or core dump. Without any modification, the library 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.

In order to give Solaris tools visibilty into Linux binaries, librtld_db must be modified to understand enough of the Linux linker internals to walk its link maps and return information about the Linux objects mapped into the target (either a core file or a live process).

The elfexec() routine in the kernel saves a pointer to the runtime linker's debug data and makes it available via the AT_SUN_LDDATA aux vector entry. > *From that pointer, librtld_db is able to find the link maps and walk the* linker's internal data structures. For librtld_db to process Linux binariues, the brand's exec routine must save a pointer to the non-native linker's debug data and make it available through the AT_SUN_BRAND_LDDATA aux vector entry.

Preliminary investigation shows that because the link map structures used by the Linux runtime linker are similar enough to those used by the Solaris runtime linker, it should be possible to modify librtld_db to walk the Linux linker's link maps. With that done, higher level tools that talk to librtld_db should be able to get symbols and address information about Linux binaries.

3.8.3 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`.

3.8.4 DTrace

Support will be provided so that the DTrace PID provider can be applied to Linux processes. Preliminary work suggests that this will require only minimal extra effort to do, once the librtld_db support is in place.

We will also insert SDT probes into the *lx* brand library to provide visibility into the actions taken by that library.

Finally, we will instrument the int80 handler, allowing DTrace to trace the Linux system calls issued by the application.

3.9. /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/*	no*	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.

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.c	manages the brandx framework, tracks loaded brands, and so on.
usr/src/uts/common/syscall/brandsys.c	the brandsys() system call

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

For the *lx* brand:

usr/src/pkgdefs/SUNWlx	The package containing all the pieces 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_systrace	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

Userspace components:

usr/lib/brand/lx/lx_brand.so.1	*lx* emulation library
usr/lib/brand/lx/lx_librtld_db.so.1	*lx* rtld_db plugin
usr/lib/brand/lx/platform.xml	Definition of the *lx* virtual platform
usr/lib/brand/lx/install	The scripts needed to install supported distros
usr/lib/brand/lx/lx_install	Main *lx* install script
usr/lib/brand/lx/lx_postboot	Postboot script

Kernel components:

kernel/brand/lx_brand	*lx* brand kernel module for 32-bit systems
kernel/brand/amd64/lx_brand	*lx* brand kernel module for 64-bit systems
kernel/drv/lx_systrace	*lx* syscall provider for 32-bit systems
kernel/drv/amd64/lx_systrace	*lx* syscall provider for 64-bit systems
kernel/fs/lx_proc	*lx* /proc filesystem for 32-bit systems
kernel/fs/amd64/lx_proc	*lx* /proc filesystem for 64-bit systems
kernel/strmod/ldlinux	*lx* ldterm for 32-bit systems
kernel/strmod/amd64/ldlinux	*lx* ldterm for 64-bit systems

All of these components are installed in the global zone.

The /usr tree is loopback mounted into the Linux zone, so the brand library is available to Linux applications.

Note: We are still investigating exactly which components to include in the non-global zone. In the final product, it is likely the we will only make the required libraries and /proc available in the non-global zone.