Glen or Glenda

Empowering Users and Applications with Private Namespaces

Eric Van Hensbergen
IBM Research
bergevan@us.ibm.com

Abstract

Private name spaces were first introduced into LINUX during the 2.5 kernel series. Their use has been limited due to name space manipulation being considered a privileged operation. Giving users and applications the ability to create private name spaces as well as the ability to mount and bind resources is the key to unlocking the full potential of this technology. There are serious performance, security and stability issues involved with user-controlled dynamic private name spaces in LINUX. This paper proposes mechanisms and policies for maintaining system integrity while unlocking the power of dynamic name spaces for normal users. It discusses relevant potential applications of this technology including its use with FILESYSTEM IN USERSPACE[24], V9FS[8] (the LINUX port of the PLAN 9 resource sharing protocol) and PLAN 9 FROM USER SPACE[4] (the PLAN 9 application suite including user space synthetic file servers ported to UNIX variants).

1 What's in a name?

Names are used in all aspects of computer science[21]. For example, they are used to reference variables in programing languages, index elements in a database, identify machines in a network, and reference resources within a file system. Within each of these categories names are evaluated in a specific context. Program variables have scope, database indexes are evaluated within tables, networks machine names are valid within a particular domain, and file names provide a mapping to underlying resources within a particular *name space*. This paper is primarily concerned with the evaluation and manipulation of names and name space contexts for file systems under the LINUX operating system.

File systems evolved from flat mappings of names to multi-level mappings where each catalog (or directory) provided a context for name resolution. This design was carried further by MULTICS[1] with deep hierarchies of directories including the concept of links between directories within the hierarchy[5]. Dennis Ritchie, Rudd Canaday and Ken Thompson built the first UNIX file system based on MULTICS, but with an emphasis on simplicity[22]. All these file systems had a single, global name space.

In the late 1980s, Thompson joined with Rob Pike and others in designing the PLAN 9 operating system. Their intent was to explore potential solutions to some of the shortcomings of UNIX in the face of the widespread use of high-speed networks[19]. It was designed from first principles as a seamless distributed system

with integrated secure network resource sharing.

The configuration of an environment to use remote application components or services in place of their local equivalent is achieved with a few simple command line instructions. For the most part, application implementations operate independent of the location of their actual resources. PLAN 9 achieves these goals through a simple well-defined interface to services, a simple protocol for accessing both local and remote resources, and through dynamic, stackable, per-process private name spaces which can be manipulated by any user or application.

On the other hand, the LINUX file system name space has traditionally been a global flat name space much like the original UNIX operating system. In November of 2000, Alexander Viro proposed implementing PLAN 9 style per-process name space bindings[28], and in late February 2001 released a patch[2] against the 2.4 kernel. This code was later adopted into the mainline kernel in 2.5. This support, which is described in more detail in section 4, established an infrastructure for private name spaces but restricted the creation and manipulation of name spaces as privileged.

This paper presents the case for making name space operations available to common users and applications while extending the existing LINUX dynamic name space support to have the power and flexibility of PLAN 9 name spaces. Section 2 describes the design, implementation and advantages of the PLAN 9 distributed system. Example applications of this technology are discussed in Section 3. The existing LINUX support is described in more detail in Section 4. Perceived barriers and solutions to extended LINUX name space support are covered in Section 5. Section 6 overviews related work and recent proposals as alternatives to our approach and Section 7 summarizes our conclusions and recommendations.

2 Background: Plan 9

In PLAN 9, all system resources and interfaces are represented as files. UNIX pioneered the concept of treating devices as files, providing a simple, clear interface to system hardware. In the 8th edition, this methodology was taken further through the introduction of the /proc synthetic file system to manage user processes[10]. Synthetic file systems are comprised of elements with no physical storage, that is to say the files represented are not present as files on any disk. Instead, operations on the file communicate directly with the sub-system or application providing the service. LINUX contains multiple examples of synthetic file systems representing devices (DEVFS), process control (PROCFS), and interfaces to system services and data structures (SYSFS).

PLAN 9 took the file system metaphor further, using file operations as the simple, well-defined interface to all system and application services. The design was based on the knowledge that any programmer knows how to interact with files. Interfaces to all kernel subsystems from the networking stack to the graphics frame buffer are represented within synthetic file systems. User-space applications and services export their own synthetic file systems in much the same way as the kernel interfaces. Common services such as domain name service (DNS), authentication databases, and window management are all provided as file systems. End-user applications such as editors and e-mail systems export file system interfaces as a means for data exchange and control. The benefits and details of this approach are covered in great detail in the existing PLAN 9 papers[18] and will be covered to a lesser extent by application examples in section 3.

9P[15] represents the abstract interface used to access resources under PLAN 9. It is somewhat analogous to the VFS layer in LINUX[11].

In PLAN 9, the same protocol operations are used to access both local and remote resources, making the transition from local resources to cluster resources to grid resources completely transparent from an implementation standpoint. Authentication is built into the protocol and was extended in its INFERNO[20] derivative Styx[14] to include various forms of encryption and digesting.

It is important to understand that all 9P operations can be associated with different active semantics in synthetic file systems. Traversal of a directory hierarchy can allocate resources or set locks. Reading or writing data to a file interface can initiate actions on the server. The dynamic nature of these semantics makes caching dangerous and in-order synchronous execution of file system operations a must.

The 9P protocol itself requires only a reliable, in-order transport mechanism to function. It is commonly used on top of TCP/IP[16], but has also been used over RUDP[13], PPP[23], and over raw reliable mechanisms such as the PCI bus, serial port connections, and shared memory. The IL protocol was designed specifically to provide 9P with a reliable, in order transport on top of an IP stack without the overhead of TCP[17].

The final key design point of PLAN 9 is the organization of all local and remote resources into a dynamic private name space. Manipulating an element's location within a name space can be used to configure which services to use, interpose stackable layers onto service interfaces, and create restricted "sandbox" environments. Under PLAN 9 and INFERNO, the name space of each process is unique and can be manipulated by ordinary users through mount and bind system calls.

Mount operations allow a client to attach new interfaces and resources which can be provided by the operating system, a synthetic file server, or from a remote server. Bind commands allow reorganization of the existing name space, allowing certain services to be bound to well-known locations. Bind operations can substitute one resource for another, for example, binding a remote device over a local one. Binding can also be used to create stackable layers by interposing one interface over another. Such interposer interfaces are particularly useful for debugging and statistics gathering.

The default mount and bind behavior is to replace the mount-point. However, PLAN 9 also allows multiple directories to be stacked at a single point in the name space, creating a *union* directory. Within such a directory, each component is searched to resolve name lookups. Flags to the mount and bind operations determine the position of a particular component in the stack. A special flag determines whether or not file creation is allowed within a particular component.

By default, processes inherit an initial name space from their parent, but changes made to the child's name space are not reflected in the parent's. This allows each process to have a context-specific name space. The PLAN 9 fork system call may be called with several flags allowing for the creation of processes with shared name spaces, blank name spaces, and restricted name spaces where no new file systems can be mounted. PLAN 9 also provides library functions (and associated system calls) for creating a new name space without creating a process and for constructing a name space based on a file describing mount sources, destinations, and options.

3 Applications

There are many areas where the pervasive use of private dynamic name spaces under PLAN 9

can be applied in similar ways under LINUX. Many of these are detailed in the foundational PLAN 9 papers [19, 18] as well as the PLAN 9 manual pages[15]. We will step through a subset of these applications and provide some additional potential applications in the LINUX environment.

Under PLAN 9, one of the more straightforward uses of dynamic name space is to bind resources into well-known locations. For example, instead of using a PATH environment variable, various executables are bound into a single `/bin` union directory. PLAN 9 clusters use a single file server providing resources for multiple architectures. Typical startup profiles bind the right set of binaries to the `/bin` directory. For example, if you logged in on an x86 host, the binaries from `/386/bin` would be bound to `/bin`, while on PPC `/power/bin` would be bound over bin. Then the user's private binary directory is bound on top of the system binaries. This has a side benefit of searching the various directories in a single lookup system call versus individually walking to elements in the path list from the shell.

Another use of stackable binds in PLAN 9 is within a development environment. You can bind directories (or even individual files) with your changes over read-only versions of a larger hierarchy. You can even recursively bind a blank hierarchy over the read-only hierarchy to deposit compiled object files and executables. The PLAN 9 development environment at Bell Labs has a single source tree which people bind their working directories and private object/executable trees over. Once they are satisfied with their changes, they can push them from the local versions to the core directories. Using similar techniques developers can also keep distinct groups of changes separated without having to maintain copies of the entire tree.

Crafting custom name spaces is also a good way to provide tighter security controls for services. Daemons exporting network services can be locked into a very restrictive name space, thus helping to protect system integrity even if the daemon itself becomes compromised. Similarly, users accessing data from other domains over mounted file systems don't run as much risk of other users gaining access if they mount the resources in a private name space. Users can craft custom sandboxes for untrusted applications to help protect against potential malicious software and applets.

As mentioned earlier, PLAN 9 combines dynamic name space with a remote resource sharing protocol to enable transparent distributed resource utilization. Remote resources are bound into the local name space as appropriate and applications run completely oblivious to what resources they are actually using. A straightforward example of this is mounting a networked stereo component's audio device from across the room instead of using your workstation's sound card before starting an audio jukebox application. A more practical example is mounting the external network protocol stack of a firewall device before running a web browser client. Since the external network protocol stack is only mounted for the particular browser client session, other services running in separate sessions with separate name spaces (and protocol stacks) are safe from external access. The PLAN 9 paradigm of mounting any distributed resource and transparently replacing local resources (or providing a network resource when a local resource isn't available) provides an interesting model for implementing grid and utility based computing.

Another example of this is the PLAN 9 cpu(1) command which is used to connect from a terminal to a cluster compute node. Note that this command doesn't operate like ssh or telnet. The cpu(1) command will export to the server the current name space of the process from which it was executed on the client. Server side

scripts take care of binding the correct architectural binaries for the cpu server over those of the client terminal. Interactive I/O between the cpu and the client is actually performed by the cpu server mounting the client's keyboard, mouse, and display devices into the session's private name space and binding those resources over its own. Custom profiles can be used to limit the resources exported to the cpu server, or to add resources such as local audio devices or protocol stacks. It represents a more elegant approach to the problems of grid, cluster, and utility-based computing providing a mechanism for the seamless integration and organization of local resources with those spread across the network.

Similar approaches can be provided to virtualization and para-virtualization environments. At the moment, the LINUX kernel is plagued by a plethora of "virtual" device drivers supporting various devices for various virtualized environments. Separate gateway devices are supported for Xen[7], VMware[30], IBM Hypervisors[3], User Moder Linux[9], and others. Additionally, each of these virtualization engines requires separate gateways for each class of device. The PLAN 9 paradigm provides a unified, simple, and secure method for supporting these various virtual architectures and their device, file system, and communication needs. Dynamic private name spaces enable a natural environment for sub-dividing and organizing resources for partitioned environments. Application file servers or generic plug-in kernel modules provide a variety of services including copy-on-write file systems, copy-on-write devices, multiplexed network connections, and command and control structures. IBM Research is currently investigating using V9FS together with private name spaces and application synthetic file servers to provide just such an approach for partitioned scale-out clusters executing high-performance computing applications.

4 Linux Name Spaces

The private name space support added in the 2.5 kernel revolved around the addition of a `CLONE_NEWNS` flag to the LINUX clone(2) system call. The clone(2) system call allows the creation of new threads which share a certain amount of context with the parent process. The flags to clone specify the degree of sharing which is desired and include the ability to share file descriptor tables, signal handlers, memory space, and file system name space. The current default behavior is for processes and threads to start with a shared copy of the global name space.

When the `CLONE_NEWNS` flag is specified, the child thread is started with a copy of the name space hierarchy. Within this thread context, modifications to either the parent or child's name space are not reflected in the other. In other words, when a new name space is requested during thread creation, file servers mounted by the child process will not be visible in the parent's name space. The converse is also true. In this way, a thread's name space operations can be isolated from the rest of the system. The use of the `CLONE_NEWNS` flag is protected by the `CAP_SYS_ADMIN` capability, making its use available only to privileged users such as root.

LINUX name spaces are currently manipulated by two system calls: `mount(2)` and `umount(2)`. The `mount(2)` system call attaches a file system to a mount-point within the current name space and the `umount(2)` system call detaches it. More recently in the 2.4 kernel series, the `MS_BIND` flag was added to allow an existing file or directory subtree to be visible at other mount-points in the current name space. Both system calls are only valid for users with `CAP_SYS_ADMIN` capability, and so are predominately used only by root. The table of mount points in a thread's

current name space can be viewed by looking at the `/proc/xxx/mounts` file.

Users may be granted the ability to mount and unmount file systems through the mount(1) application and certain flags in the fstab(5) configuration file. This support requires that the mount application be configured with set-uid privileges and that the exact mount source and destination be specified in the fstab(5). Certain network file systems (such as SMBFS, CIFS, and V9FS) which have a more user-centric paradigm circumvent this by having their own set-uid mount utilities: smbmnt(8), cifs.mount(8), and 9fs(1). More recently, there has been increased interest in user-space file servers such as FILESYSTEM IN USERSPACE (FUSE)[24] with its own set-uid mount application fusermount(8).

The proliferation of these various set-uid applications that circumvent the kernel protection mechanisms indicates the need to re-evaluate the existing restrictions so that a more practical set of policies can be put in place within the kernel. Users should be able to mount file systems when and where appropriate. Private name spaces seem to be a natural fit for preventing global name space pollution with individual user mount and bind activities. They also provide a certain degree of isolation from user mounted synthetic file systems, providing an additional degree of protection to system demons and other users who might otherwise unwittingly access a malicious user-level file server.

Private name space support in LINUX is under utilized primarily due to the classification of name space operations as privileged. It is further crippled by the lack of stackable name space semantics and application file servers. Unlocking applications and environments such as those described in Section 3 by removing some of the restrictions enforced by the LINUX kernel would create a much more elegant and powerful computing environment. Additionally, providing a more flexible, yet consistent set of kernel enforced policies would be far superior to the wide range of semantics currently enforced by file system specific set-uid mount applications.

5 Barriers and Solutions

PLAN 9 is not LINUX, and LINUX is not PLAN 9. There are significant security model and file system paradigm differences between the two systems. Concerns related to these differences have been broken down into four major categories: concerns with user name space manipulation, problems with users being able to mount arbitrary file systems, potential problems with user file systems, and problems with allowing users to create their own private name spaces.

5.1 Binding Concerns

The `mount(1)` command specified with the `-bind` option, hereafter referred to as a bind operation, is an incredibly useful tool even in a shared global name space. When combined with the notion of private name spaces, it allows users and applications to craft custom environments in which to work. However, the ability to dynamically bind directories and/or files over one another creates several security concerns that revolve around the ability to transparently replace system configuration and common data with potentially compromised versions.

PLAN 9 places no restrictions on bind operations. Users are free to bind over any system directory or file regardless of access permissions—binding writable layers over otherwise read-only directories can be one of the more useful operations. However, PLAN 9's

authentication and system configuration mechanisms are constructed in such a way as to not rely on accessing files when running under user contexts. In other words, authentication and configuration are system services which are started at boot (or reside on different servers), and so aren't affected by user manipulations of their private name spaces.

Under LINUX, system services are constructed differently and there is still heavy reliance on well-known files which are accessed throughout user sessions. Examples include such sensitive files as `/etc/passwd` and `/etc/fstab`.

Similar concerns apply to certain system directories which multiple users may have write access to, such as `/tmp` or `/usr/tmp`. If users are able to bind over these public directories under the global name space, they could potentially compromise the data of another user who inadvertently used a bound `/tmp` instead of the system `/tmp`.

These problems can be addressed with a simple policy of only allowing a user to bind over a directory they have explicit write access to. This solves the problem of system configuration files, but doesn't cover globally writable spaces such as `/usr/tmp`. A simple solution to protecting such shared spaces is to only allow user initiated binds within private name spaces. A slightly more complicated form of protection is based on the assumption that such public spaces have the *sticky bit* set in the directory permissions.

When used within directory permissions, the sticky bit specifies that files or subdirectories can only be renamed or deleted by their original owner, the owner of the directory, or a privileged process. This prevents users from deleting or otherwise interfering with each other's files in shared public spaces. A simple policy to extend this protection to user name space manipulation is to return a permissions error when a normal user attempts to bind over a directory in which the sticky bit is set.

While limiting binds to sticky-bit directories is reasonable enough, it is an unnecessary restriction. The use of private name spaces solves several security concerns with user-modifications to name space, and does so without overly limiting the user's ability to mount over these shared spaces. Another benefit of requiring user binds to be within a private name space is that it prevents such binds from polluting the global system name space.

5.2 Mounting Concerns

Another set of concerns has to do with allowing users to mount new file systems into a name space. As discussed previously, this is something currently accomplished through set-uid mount applications which check the user's permissions versus particular policies. A more global policy would give administrators more consistent control over users and help eliminate the potential problems caused by the use of set-uid applications

One of the primary problems with giving users the ability to mount arbitrary file systems is the concern that they may mount a file system with set-uid scripts allowing them to gain access to privileged accounts (i.e., root). It is relatively trivial for a user to construct a file system image, floppy, or CD-ROM on a personal machine with set-uid shells. If they were allowed to mount these on an otherwise secure system, they could instantly compromise it. The existing mount applications circumvent such a vulnerability by providing a NOSUID flag which disables interpretation of set-uid and set-gid permission flags. A similar

mechanism enforced as the default for all user-mounts would provide a certain level of protection against such an attack.

Another possible attack vector would be the image being mounted. Most file systems are written on the assumption that the backing store is somewhat secure and reputable. LINUX kernel community members have expressed concern that disk images could be constructed specifically to crash or corrupt certain file systems, so as to disable or disrupt system activity. This is particularly difficult to protect against, but not all file systems are vulnerable to such attacks. In particular, network file systems are written defensively to prevent such corruption from affecting the rest of the system. Such defensively written file systems could be marked with an additional file system flag marking them as safe for users to mount.

Each mounted file system uses a certain amount of system resources. Unlocking the ability to mount a new file system also unlocks the ability for the user to abuse the system resources by mounting new file systems until all system memory is expended. This sort of activity is easily controlled with per-user mount limits maintained using the kernel resource limit system with a policy set by the system administrator.

A slightly different form of resource abuse mentioned earlier is name space pollution. If users are granted the ability to mount and bind a large number of file systems, the resulting name space pollution could prove to be distracting, if not damaging to performance. Enforcing a policy in which users are only able to mount new file systems within a private name space easily contains such pollution to the user's session. Additionally, the current name space garbage collection will take care of conveniently unmounting file servers and recovering resources when the session associated with the private name space closes.

5.3 User File System Concerns

A driving motivation behind providing users the ability to mount and bind file systems is the increase in popularity of user-space file servers. These predominantly synthetic file systems are enabled through a number of different packages including V9FS and more predominantly FUSE. These packages export VFS interfaces or equivalent APIs to user space, allowing applications to act as file servers. Practical uses for such file servers include the exporting of archive file contents as mountable name spaces, adding cryptographic layers, and mapping of network transports such as ftp to synthetic file hierarchies.

Since they are implemented as user applications, these synthetic file servers pose an even greater danger to system integrity by allowing users to implement arbitrary semantics for operations. These implementations can easily provide corrupt data to system calls or block system call resolution indefinitely, bringing the entire system to a grinding halt. Because of this, application file servers have fallen under harsh criticism from the LINUX kernel community. However their many practical uses makes the engineering of a safe and reliable mechanism allowing their use in a LINUX environment highly desirable.

Many of the prior solutions mentioned can be used to limit the damage done by a malicious user-space file servers. Private name spaces can protect system daemons and other users from stumbling into a synthetic file system trap. Restrictions preventing set-uid and set-gid interpretation within user mounts can prevent malicious users from using application file servers to gain access to privileged accounts or information.

A different sort of permissions problem is also introduced by application file servers. Typi-

cally, the file servers are started by a certain user and information within the file system is accessed under that user's authority, potentially in a different authentication domain. For example, if a user mounts a ftpfs or an sshfs by logging into a remote server domain, they are potentially exposing the data from that domain to other users and administrators on the local domain. As this is undesirable, it is important that other users (besides the initiator) do not obtain direct access to mounted file systems. While there are several ways of approaching this (including overloaded permissions checks that deny access to anyone but the mounter), private name spaces seem to handle this nicely without changing other system semantics.

5.4 Private Name Space Concerns

While they are limited, several barriers do exist to user creation and use of private name spaces. One objection to allowing users to create their own private name spaces is the existence of a vulnerability in the `chroot(1)` infrastructure in the presence of such private name spaces. The `chroot(1)` command is used to establish a new root for a particular user's name space. However, if a private name space is created with the `CLONE_NS` flag, the new thread is allowed to traverse out of the chroot "jail" simply using the dot-dot traversal. This appears to be more of a bug than a feature and should be easy to defend against by never allowing a user to traverse out of the root of their current name space.

The same resource concerns that apply to user mounts also apply to private name spaces. However, since the user can have no more private name spaces than processes, there is a preexisting constraint. Additionally, due to the copy semantics present in the existing LINUX name space infrastructure, the user will be charged for every mount he inherits when creating a private name space. If these two limitations are deemed insufficient, an additional per-user limit can be established for private name spaces.

More prevalent among these perceived problems is the change in basic paradigm. No longer can the same file system environment be expected from every session on a particular system. In fact, depending on the extent to which private name spaces are used there may even be different file system views in different windows on the same session. The plurality of name spaces across processes and sessions provides a great deal of flexibility in construction of private environments, but is quite a departure from expected behavior.

The ability to maintain a certain degree of traditional semantics is desirable during a transition in paradigms. Further, having to mount core resources for each session is rather tedious and undesirable. To a certain extent this can be mitigated by more advanced inheritance techniques within the private name spaces—allowing changes in parents to be propagated to children but not vice versa. This is further discussed in the Related Work section regarding Alexander Viro's shared subtrees proposal.

Another possibility is a per-session name space created when a user logs into the system. This provides a single name space for that session separate from the global name space insulating user modifications from the unsuspecting. However, in simpler embodiments it doesn't provide the per-user name space semantics some desire (ie. the name space wouldn't actually bridge two different SSH sessions). One possibility here is to tightly bind creation and adoption of the per-user name space to the login process (potentially as part of the PAM infrastructure). Another possibility would be to use the name space description present in the

`/proc/xxxx/mounts` synthetic file to create a duplicate name space in different process groups. This would work well for network file systems, binds, and V9FS but may not work well for certain user file servers such as FUSE.

V9FS enables multi-session user file servers without problems as it separates mount-point from the file system semantics. In other words, when you run a V9FS application file server, it creates a mount point which could be used by several different clients to mount the resulting file system. Besides giving the ability to share the resulting file system between user sessions, this technique potentially allows other users to access the mount-point. User credentials are part of the V9FS mount protocol, so each user is authenticated on the file system based on their own credentials instead of the credentials of the user who initially started the file server application.

6 Related Work

There are several historical as well as ongoing attempts to provide more dynamic name space operations in LINUX and/or open up those operations to end-users and not just privileged administrators. There are also several outstanding request-for-comments on extensions to the existing name space support.

The original V9FS project had tried to integrate private name space support into the file system and remote-resource sharing [12]. While this worked in practice, Alexander Viro's release of private name space support within the LINUX kernel suspended work on the V9FS private name space implementation.

As a follow-up to Viro's initial name space support, he released a shared sub-tree request-for-comments[29] detailing specific policies for propagating name space changes from parent to children. This provides a more convenient form of inheritance allowing name space changes in parents to also take effect in children with private name spaces.

Miklos Szeredi, the project leader of FUSE has proposed several patches related to opening up and expanding name space support. Among these were an altered permission semantics[25] to prevent users other than the mounting user from accessing FUSE mounts. After this met from some resistance from the LINUX kernel community, Miklos proposed an invisible mount patch[26] which tries to protect other users from potentially malicious mounts by hiding them from other users without the use of private name spaces. A separate patch[27] attempted to unlock mount privileges by enforcing a static policy on user-mounts including some of the protections we have described previously (only writable directories can be mounted over, only safe file systems can be mounted, and set-uid/set-gid permissions are disabled). To date, none of these patches have been incorporated into the mainline, but most of these events are happening concurrently with the writing and revision of this paper.

One of the responses to the FUSE patches was the assertion that the job may have been better done in user-space by an extended form of the mount(1) application. The advantage to using a user-space policy solution is a much wider and dynamic set of policies than would be desirable to incorporate directly into the kernel. Such an application would have set-uid style permissions, which several in the community have criticized as undesirable. An alternative to this approach would be to use up-calls from the kernel to a user-space policy daemon.

Another outcome of the FUSE discussion was a patch[6] providing an unshare system call which could be used to create private name spaces in a pre-existing thread. In other words,

this would allow a thread to request a private name space without having been spawned with one, making the creation of private name spaces more accessible. The unshare patch also provides similar facilities for controlling other resources originally only available via flags during the clone system call.

The file system translator project (FiST)[33] takes a different approach, offering users the ability to add incremental features to existing file systems. It provides a set of templates and a toolkit which allow for relatively easy creation of kernel file system modules which sit atop pre-existing conventional file systems. The resulting modules have to be installed and mounted by a privileged user. Instead of relying on set-uid helper applications, FiST allows use of "private" instances of the file system through a special ioctl attach command and per-user sub-hierarchies. Several example file system layers are provided with the standard FiST distribution including cryptographic layers and access control list enforcement layers.

One of the more interesting FiST file system layers is UNIONFS[32][31]. It provides a fan-out file system which goes beyond the relatively simple semantics of PLAN 9's union directories by providing additional flexibility and granular control of specific components. There is also support for rudimentary sandboxing without the use of private name spaces.

Among the additional features of UNIONFS is recursive unification allowing deep binds of directories. In PLAN 9 and the existing LINUX name space implementations, a bind only affects a single file or directory. The recursive unification feature of UNIONFS allows entire hierarchies to be bound. This is particularly useful in the context of copy-on-write file system semantics. While such functionality can be provided with scripts under PLAN 9 and LINUX, the UNIONFS approach would seem to provide a more efficient and scalable solution.

7 Conclusions

Opening up name space operations to common users will enable better working environments and transparent cluster computing. Users should be granted the permission to establish private name spaces through flags provided to the clone(2) system call or using the newly proposed unshare system call. Once isolated in a private name space, normal users should be granted the ability to mount new resources and organize existing resources in ways they see fit. A simple set of system-wide restrictions on these activities will prevent malicious users from obtaining privileged access, disrupting system operation, or compromising protected data. Adding stackable file name spaces into the kernel file system interfaces would further extend these benefits.

8 Acknowledgements

Between the time this paper was proposed and published, much debate has occurred on the LINUX kernel mailing list and the LINUX file systems developers mailing list. I've incorporated a great deal of that discussion and commentary into this document and many of the ideas represented here come from that community.

I'd like to thank Alexander Viro for laying the ground work by adding the initial private name space support to LINUX. I'd also like to thank Miklos Szeredi and the FUSE team for pushing the ideas of unprivileged mounts and user application file servers.

Support for this paper was provided in part by the Defense Advance Research Projects Agency under Contract No. NBCH30390004.

References

[1] F. J. Corbato and V.A. Vyssotsky. Introduction and overview of the multics system. *Joint Computer Conference*, 1965.

[2] Jonathan Corbet. Kernel development. *LWN.NET Weekly News*, 0301, March 2001.

[3] IBM Corp. Virtualization engine. `http://www.ibm.com/`.

[4] Russ Cox. Plan 9 from user space. `http://swtch.com/plan9port`.

[5] R.C. Daley and P.G. Neumann. A general-purpose file system for secondary storage. *Fall Joint Computer Conference*, 1965.

[6] Janak Desai. new system call, unshare. EMAIL, May 2005. `http://marc.theaimsgroup.com/?l=linux-fsdevel&m=111573064706562&w=2`.

[7] B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, I. Pratt, A. Warfield, P. Barham, and R. Neugebauer. Xen and the art of virtualization. In *Proceedings of the ACM Symposium on Operating Systems Principles*, October 2003.

[8] E. Van Hensbergen and R. Minnich. Grave robbers from outer space: Using 9p200 under linux. In *Proceedings of Freenix Annual Conference*, pages 83–94, 2005.

[9] Hans jorg H Oxer, Hans jorg Hoxer, Kerstin Buchacker, and Volkmar Sieh. Implementing a user mode linux with minimal changes from original kernel. *unknown*, 2002.

[10] T.J. Killian. Processes as files. In *USENIX Summer Conf. Proceedings*, Salt Lake City, UT, June 1984.

[11] Robert Love. *Linux Kernel Development*. Sam's Publishing, 800 E. 96th Street, Indianapolis, Indiana 46240, 2nd edition, August 2003.

[12] Ron Minnich. V9fs: A private name space system for unix and its uses for distributed and cluster computing. In *Conference Francaise sur les Systemes*, June 1999.

[13] C. Partridge and r. Hinden. Reliable data protocol. Internet RFC/STD/FYI/BCP Archives, April 1990.

[14] R. Pike and D. M. Ritchie. The styx architecture for distributed systems. *Bell Labs Technical journal*, 4(2):146–152, April-June 1999.

[15] Rob Pike et al. *Plan 9 Programmer's Manual - Manual Pages*. Vita Nuova Holdings Limited, 3rd edition, 2000.

[16] J. Postel. Transmission control protocol darpa internet program protocol specification. Internet RFC/STD/FYI/BCP Archives, September 1981.

[17] D. Presotto and P. Winterbottom. *The IL Protocol*, volume 2, pages 277–282. AT&T ell Laboratories, Murray Hill, NJ, 1995.

[18] D. Presotto R. Pike et al. The use of name spaces in plan 9. *Operating Systems Review*, 27(2):72–76, April 1993.

[19] K. Thompson R. Pike et al. Plan 9 from bell labs. *Computing Systems*, Vol 8(3):221–254, Summer 1995.

[20] R. Pike S. Dorward et al. The inferno operating system. *Bell Labs Technical Journal*, 2(1):5–18, Winter 1997.

[21] J. H. Saltzer. *Lecture Notes in computer Science, 60, Operating systems - An Advanced Course*, chapter 3.A.: Naming and Binding of Objects, pages 99–208. Springer-Verlag, 1978.

[22] Peter H. Salus. *The Daemon, the GNU, and the Penguin*, chapter 2 & 3: UNIX. Groklaw, 2005.

[23] W. Simpson. The point-to-point protocol (ppp) for the transmission of multi-protocol datagrams over point-to-point links. Internet RFC/STD/FYI/BCP Archives, May 1992.

[24] Miklos Szeredi. Filesystem in userspace. `http://fuse.sourceforge.net`.

[25] Miklos Szeredi. Fuse permission modell. EMAIL, April 2005. `http://marc.theaimsgroup.com/?l=linux-fsdevel&m=111323066112311&w=2`.

[26] Miklos Szeredi. Private mounts. EMAIL, April 2005. `http://marc.theaimsgroup.com/?l=linux-fsdevel&m=111437333932219&w=2`.

[27] Miklos Szeredi. Unpriviledged mount/umount. EMAIL, May 2005. `http://marc.theaimsgroup.com/?l=linux-fsdevel&m=111513156417879&w=2`.

[28] Alexander Viro. Re: File system enhancement handled above the file system level. Email, November 2000.

[29] Alexander Viro. Shared subtrees. EMAIL, January 2005. `http://marc.theaimsgroup.com/?l=linux-fsdevel&m=110565591630267&w=2`.

[30] VMware. Vmware home page. `http://www.vmware.com`.

[31] C. P. Wright et al. Versatility and unix semantics in a fan-out unification file system. Technical Report FSL-04-01B, Computer Science Department, Stony Brook University, October 2004. `http://www.fsl.cs.sunysb.edu/docs/unionfs-tr/unionfs.pdf`.

[32] C.P. Wright and E. Zadok. Unionfs: Bringing file systems together. *Linux Journal*, December 2004.

[33] E. Zadok. Writing stackable file systems. *Linux Journal*, pages 22–25, May 2003.

Proceedings of the Linux Symposium

Volume Two

July 20nd–23th, 2005
Ottawa, Ontario
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc.*
C. Craig Ross, *Linux Symposium*
Stephanie Donovan, *Linux Symposium*

Review Committee

Gerrit Huizenga, *IBM*
Matthew Wilcox, *HP*
Dirk Hohndel, *Intel*
Val Henson, *Sun Microsystems*
Jamal Hadi Salimi, *Znyx*
Matt Domsch, *Dell*
Andrew Hutton, *Steamballoon, Inc.*

Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*