Lexical File Names in Plan 9
or
Getting Dot-Dot Right

Rob Pike
`rob@plan9.bell-labs.com`

Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

Symbolic links make the Unix file system non-hierarchical, resulting in multiple valid path names for a given file. This ambiguity is a source of confusion, especially since some shells work overtime to present a consistent view from programs such as `pwd`, while other programs and the kernel itself do nothing about the problem.

Plan 9 has no symbolic links but it does have other mechanisms that produce the same difficulty. Moreover, Plan 9 is founded on the ability to control a program's environment by manipulating its name space. Ambiguous names muddle the result of operations such as copying a name space across the network.

To address these problems, the Plan 9 kernel has been modified to maintain an accurate path name for every active file (open file, working directory, mount table entry) in the system. The definition of 'accurate' is that the path name for a file is guaranteed to be the rooted, absolute name the program used to acquire it. These names are maintained by an efficient method that combines lexical processing—such as evaluating `..` by just removing the last path name element of a directory—with local operations within the file system to maintain a consistently, easily understood view of the name system. Ambiguous situations are resolved by examining the lexically maintained names themselves.

A new kernel call, `fd2path`, returns the file name associated with an open file, permitting the use of reliable names to improve system services ranging from `pwd` to debugging. Although this work was done in Plan 9, Unix systems could also benefit from the addition of a method to recover the accurate name of an open file or the current directory.

Motivation

Consider the following unedited transcript of a session running the Bourne shell on a modern Unix system:

```
% echo $HOME
/home/rob
% cd $HOME
% pwd
/n/bopp/v7/rob
% cd /home/rob
% cd /home/ken
% cd ../rob
../rob: bad directory
%
```

(The same output results from running `tcsh`; we'll discuss `ksh` in a moment.) To a neophyte being schooled in the delights of a hierarchical file name space, this behavior must be baffling. It is, of course, the consequence of a series of symbolic links intended to give users the illusion they share a disk, when in fact their files are scattered over several devices:

```
% ls -ld /home/rob /home/ken
lrwxr-xr-x  1 root  sys   14 Dec 26  1998 /home/ken -> /n/bopp/v6/ken
lrwxr-xr-x  1 root  sys   14 Dec 23  1998 /home/rob -> /n/bopp/v7/rob
%
```

The introduction of symbolic links has changed the Unix file system from a true hierarchy into a directed graph, rendering `..` ambiguous and sowing confusion.

Unix popularized hierarchical naming, but the introduction of symbolic links made its naming irregular. Worse, the `pwd` command, through the underlying `getwd` library routine, uses a tricky, expensive algorithm that often delivers the wrong answer. Starting from the current directory, `getwd` opens the parent, `..`, and searches it for an entry whose i-number matches the current directory; the matching entry is the final path element of the ultimate result. Applying this process iteratively, `getwd` works back towards the root. Since `getwd` knows nothing about symbolic links, it will recover surprising names for directories reached by them, as illustrated by the example; the backward paths `getwd` traverses will not backtrack across the links.

Partly for efficiency and partly to make `cd` and `pwd` more predictable, the Korn shell `ksh` [Korn94] implements `pwd` as a builtin. (The `cd` command must be a builtin in any shell, since the current directory is unique to each process.) `Ksh` maintains its own private view of the file system to try to disguise symbolic links; in particular, `cd` and `pwd` involve some lexical processing (somewhat like the `cleanname` function discussed later in this paper), augmented by heuristics such as examining the environment for names like `$HOME` and `$PWD` to assist initialization of the state of the private view. [Korn00]

This transcript begins with a Bourne shell running:

```
% cd /home/rob
% pwd
/n/bopp/v7/rob
% ksh
$ pwd
/home/rob
$
```

This result is encouraging. Another example, again starting from a Bourne shell:

```
% cd /home/rob
% cd ../ken
../ken: bad directory
% ksh
$ pwd
/home/rob
$ cd ../ken
$ pwd
/home/ken
$
```

By doing extra work, the Korn shell is providing more sensible behavior, but it is easy to defeat:

```
% cd /home/rob
% pwd
/n/bopp/v7/rob
% cd bin
% pwd
/n/bopp/v7/rob/bin
% ksh
$ pwd
/n/bopp/v7/rob/bin
$ exit
% cd /home/ken
% pwd
/n/bopp/v6/ken
% ksh
$ pwd
/n/bopp/v6/ken
$
```

In these examples, `ksh`'s built-in `pwd` failed to produce the results (`/home/rob/bin` and `/home/ken`) that the previous example might have led us to expect. The Korn shell is hiding the problem, not solving it, and in fact is not even hiding it very well.

A deeper question is whether the shell should even be trying to make `pwd` and `cd` do a better job. If it does, then the `getwd` library call and every program that uses it will behave differently from the shell, a situation that is sure to confuse. Moreover, the ability to change directory to `../ken` with the Korn shell's `cd` command but not with the `chdir` system call is a symptom of a diseased system, not a healthy shell.

The operating system should provide names that work and make sense. Symbolic links, though, are here to stay, so we need a way to provide sensible, unambiguous names in the face of a non-hierarchical name space. This paper shows how the challenge was met on Plan 9, an operating system with Unix-like naming.

Names in Plan 9

Except for some details involved with bootstrapping, file names in Plan 9 have the same syntax as in Unix. Plan 9 has no symbolic links, but its name space construction operators, `bind` and `mount`, make it possible to build the same sort of non-hierarchical structures created by symbolically linking directories on Unix.

Plan 9's `mount` system call takes a file descriptor and attaches to the local name space the file system service it represents:

```
mount(fd, "/dir", flags)
```

Here `fd` is a file descriptor to a communications port such as a pipe or network connection; at the other end of the port is a service, such as file server, that talks 9P, the Plan 9 file system protocol. After the call succeeds, the root directory of the service will be visible at the *mount point* /dir, much as with the mount call of Unix. The flag argument specifies the nature of the attachment: MREPL says that the contents of the root directory (appear to) replace the current contents of /dir; MAFTER says that the current contents of dir remain visible, with the mounted directory's contents appearing *after* any existing files; and MBEFORE says that the contents remain visible, with the mounted directory's contents appearing *before* any existing files. These multicomponent directories are called *union directories* and are somewhat different from union directories in 4.4BSD-Lite [PeMc95], because only the top-level directory itself is unioned, not its descendents, recursively. (Plan 9's union directories are used differently from 4.4BSD-Lite's, as will become apparent.)

For example, to bootstrap a diskless computer the system builds a local name space containing only the root directory, /, then uses the network to open a connection to the main file server. It then executes

```
mount(rootfd, "/", MREPL);
```

After this call, the entire file server's tree is visible, starting from the root of the local machine.

While mount connects a new service to the local name space, bind rearranges the existing name space:

```
bind("tofile", "fromfile", flags)
```

causes subsequent mention of the fromfile (which may be a plain file or a directory) to behave as though tofile had been mentioned instead, somewhat like a symbolic link. (Note, however, that the arguments are in the opposite order compared to ln −s). The flags argument is the same as with mount.

As an example, a sequence something like the following is done at bootstrap time to assemble, under the single directory /bin, all of the binaries suitable for this architecture, represented by (say) the string sparc:

```
bind("/sparc/bin", "/bin", MREPL);
bind("/usr/rob/sparc/bin", "/bin", MAFTER);
```

This sequence of binds causes /bin to contain first the standard binaries, then the contents of rob's private SPARC binaries. The ability to build such union directories obviates the need for a shell $PATH variable while providing opportunities for managing heterogeneity. If the system were a Power PC, the same sequence would be run with power textually substituted for sparc to place the Power PC binaries in /bin rather than the SPARC binaries.

Trouble is already brewing. After these bindings are set up, where does

```
% cd /bin
% cd ..
```

set the current working directory, to / or /sparc or /usr/rob/sparc? We will return to this issue.

There are some important differences between binds and symbolic links. First, symbolic links are a static part of the file system, while Plan 9 bindings are created at run time, are stored in the kernel, and endure only as long as the system maintains them; they are temporary. Since they are known to the kernel but not the file system, they must be set up each time the kernel boots or a user logs in; permanent bindings are created by editing system initialization scripts and user profiles rather than by building them in the file system itself.

The Plan 9 kernel records what bindings are active for a process, whereas symbolic links, being held on the Unix file server, may strike whenever the process evaluates a file name. Also, symbolic links apply to all processes that evaluate the affected file, whereas `bind` has a local scope, applying only to the process that executes it and possibly some of its peers, as discussed in the next section. Symbolic links cannot construct the sort of `/bin` directory built here; it is possible to have multiple directories point to `/bin` but not the other way around.

Finally, symbolic links are symbolic, like macros: they evaluate the associated names each time they are accessed. Bindings, on the other hand, are evaluated only once, when the bind is executed; after the binding is set up, the kernel associates the underlying files, rather than their names. In fact, the kernel's representation of a bind is identical to its representation of a mount; in effect, a bind is a mount of the `tofile` upon the `fromfile`. The binds and mounts coexist in a single *mount table*, the subject of the next section.

The Mount Table

Unix has a single global mount table for all processes in the system, but Plan 9's mount tables are local to each process. By default it is inherited when a process forks, so mounts and binds made by one process affect the other, but a process may instead inherit a copy, so modifications it makes will be invisible to other processes. The convention is that related processes, such as processes running in a single window, share a mount table, while sets of processes in different windows have distinct mount tables. In practice, the name spaces of the two windows will appear largely the same, but the possibility for different processes to see different files (hence services) under the same name is fundamental to the system, affecting the design of key programs such as the window system [Pike91].

The Plan 9 mount table is little more than an ordered list of pairs, mapping the `fromfiles` to the `tofiles`. For mounts, the `tofile` will be an item called a `Channel`, similar to a Unix `vnode`, pointing to the root of the file service, while for a bind it will be the `Channel` pointing to the `tofile` mentioned in the `bind` call. In both cases, the `fromfile` entry in the table will be a `Channel` pointing to the `fromfile` itself.

The evaluation of a file name proceeds as follows. If the name begins with a slash, start with the `Channel` for the root; otherwise start with the `Channel` for the current directory of the process. For each path element in the name, such as `usr` in `/usr/rob`, try to 'walk' the `Channel` to that element [Pike93]. If the walk succeeds, look to see if the resulting `Channel` is the same as any `fromfile` in the mount table, and if so, replace it by the corresponding `tofile`. Advance to the next element and continue.

There are a couple of nuances. If the directory being walked is a union directory, the walk is attempted in the elements of the union, in order, until a walk succeeds. If none succeed, the operation fails. Also, when the destination of a walk is a directory for a purpose such as the `chdir` system call or the `fromfile` in a `bind`, once the final walk of the sequence has completed the operation stops; the final check through the mount table is not done. Among other things, this simplifies the management of union directories; for example, subsequent `bind` calls will append to the union associated with the underlying `fromfile` instead of what is bound upon it.

A Definition of Dot–Dot

The ability to construct union directories and other intricate naming structures introduces some thorny problems: as with symbolic links, the name space is no longer hierarchical, files and directories can have multiple names, and the meaning of `..`, the parent directory, can be ambiguous.

The meaning of `..` is straightforward if the directory is in a locally hierarchical part of the name space, but if we ask what `..` should identify when the current directory is a mount point or union directory or multiply symlinked spot (which we will henceforth call just a mount point, for brevity), there is no obvious answer. Name spaces have been part of Plan 9 from the beginning, but the definition of `..` has changed several times as we grappled with this issue. In fact, several attempts to clarify the meaning of `..` by clever coding resulted in definitions that could charitably be summarized as 'what the implementation gives.'

Frustrated by this situation, and eager to have better-defined names for some of the applications described later in this paper, we recently proposed the following definition for `..`:

> The parent of a directory *X*, *X*/`..`, is the same directory that would obtain if we instead accessed the directory named by stripping away the last path name element of *X*.

For example, if we are in the directory `/a/b/c` and `chdir` to `..`, the result is *exactly* as if we had executed a `chdir` to `/a/b`.

This definition is easy to understand and seems natural. It is, however, a purely *lexical* definition that flatly ignores evaluated file names, mount tables, and other kernel-resident data structures. Our challenge is to implement it efficiently. One obvious (and correct) implementation is to rewrite path names lexically to fold out `..`, and then evaluate the file name forward from the root, but this is expensive and unappealing. We want to be able to use local operations to evaluate file names, but maintain the global, lexical definition of dot-dot. It isn't too hard.

The Implementation

To operate lexically on file names, we associate a name with each open file in the kernel, that is, with each `Channel` data structure. The first step is therefore to store a `char*` with each `Channel` in the system, called its `Cname`, that records the *absolute* rooted file name for the `Channel`. `Cnames` are stored as full text strings, shared copy-on-write for efficiency. The task is to maintain each `Cname` as an accurate absolute name using only local operations.

When a file is opened, the file name argument in the `open` (or `chdir` or `bind` or ...) call is recorded in the `Cname` of the resulting `Channel`. When the file name begins with a slash, the name is stored as is, subject to a cleanup pass described in the next section. Otherwise, it is a local name, and the file name must be made absolute by prefixing it with the `Cname` of the current directory, followed by a slash. For example, if we are in `/home/rob` and `chdir` to `bin`, the `Cname` of the resulting `Channel` will be the string `/home/rob/bin`.

This assumes, of course, that the local file name contains no `..` elements. If it does, instead of storing for example `/home/rob/..` we delete the last element of the existing name and set the `Cname` to `/home`. To maintain the lexical naming property we must guarantee that the resulting `Cname`, if it were to be evaluated, would yield the identical directory to the one we actually do get by the local `..` operation.

If the current directory is not a mount point, it is easy to maintain the lexical property. If it is a mount point, though, it is still possible to maintain it on Plan 9 because the mount table, a kernel-resident data structure, contains all the information about the non-hierarchical connectivity of the name space. (On Unix, by contrast, symbolic links are stored on the file server rather than in the kernel.) Moreover, the presence of a full file name for each `Channel` in the mount table provides the information necessary to resolve ambiguities.

The mount table is examined in the `from`→`to` direction when evaluating a name, but `..` points backwards in the hierarchy, so to evaluate `..` the table must be examined in the `to`→`from` direction. (''How did we get here?'')

The value of `..` is ambiguous when there are multiple bindings (mount points) that point to the directories involved in the evaluation of `..`. For example, return to our original script with `/n/bopp/v6` (containing a home directory for `ken`) and `/n/bopp/v7` (containing a home directory for `rob`) unioned into `/home`. This is represented by two entries in the mount table, `from=/home`, `to=/n/bopp/v6` and `from=/home`, `to=/n/bopp/v7`. If we have set our current directory to `/home/rob` (which has landed us in the physical location `/n/bopp/v7/rob`) our current directory is not a mount point but its parent is. The value of `..` is ambiguous: it could be `/home`, `/n/bopp/v7`, or maybe even `/n/bopp/v6`, and the ambiguity is caused by two `tofiles` bound to the same `fromfile`. By our definition, if we now evaluate `..`, we should acquire the directory `/home`; otherwise `../ken` could not possibly result in `ken`'s home directory, which it should. On the other hand, if we had originally gone to `/n/bopp/v7/rob`, the name `../ken` should *not* evaluate to `ken`'s home directory because there is no directory `/n/bopp/v7/ken` (`ken`'s home directory is on `v6`). The problem is that by using local file operations, it is impossible to distinguish these cases: regardless of whether we got here using the name `/home/rob` or `/n/bopp/v7/rob`, the resulting directory is the same. Moreover, the mount table does not itself have enough information to disambiguate: when we do a local operation to evaluate `..` and land in `/n/bopp/v7`, we discover that the directory is a `tofile` in the mount table; should we step back through the table to `/home` or not?

The solution comes from the `Cnames` themselves. Whether to step back through the mount point `from=/home`, `to=/n/bopp/v7` when evaluating `..` in `rob`'s directory is trivially resolved by asking the question, Does the `Cname` for the directory begin `/home`? If it does, then the path that was evaluated to get us to the current directory must have gone through this mount point, and we should back up through it to evaluate `..`; if not, then this mount table entry is irrelevant.

More precisely, both *before* and *after* each `..` element in the path name is evaluated, if the directory is a `tofile` in the mount table, the corresponding `fromfile` is taken instead, provided the `Cname` of the corresponding `fromfile` is the prefix of the `Cname` of the original directory. Since we always know the full name of the directory we are evaluating, we can always compare it against all the entries in the mount table that point to it, thereby resolving ambiguous situations and maintaining the lexical property of `..`. This check also guarantees we don't follow a misleading mount point, such as the entry pointing to `/home` when we are really in `/n/bopp/v7/rob`. Keeping the full names with the `Channels` makes it easy to use the mount table to decide how we got here and, therefore, how to get back.

In summary, the algorithm is as follows. Use the usual file system operations to walk to `..`; call the resulting directory *d*. Lexically remove the last element of the initial file name. Examine all entries in the mount table whose `tofile` is *d* and whose `fromfile` has a `Cname` identical to the truncated name. If one exists, that `fromfile` is the correct result; by construction, it also has the right `Cname`. In our example, evaluating `..` in `/home/rob` (really `/n/bopp/v7/rob`) will set *d* to `/n/bopp/v7`; that is a `tofile` whose `fromfile` is `/home`. Removing the `/rob` from the original `Cname`, we find the name `/home`, which matches that of the `fromfile`, so the result is the `fromfile`, `/home`.

Since this implementation uses only local operations to maintain its names, it is possible to confuse it by external changes to the file system. Deleting or renaming directories and files that are part of a `Cname`, or modifying the mount table, can introduce errors. With more implementation work, such mistakes could probably be caught, but in a networked environment, with machines sharing a remote file server, renamings and deletions made by one machine may go unnoticed by others. These problems, however, are minor, uncommon and, most important, easy to understand. The method maintains the lexical property of file names unless an external agent changes the name surreptitiously; within a stable file system, it is always maintained and `pwd` is always right.

To recapitulate, maintaining the `Channel`'s absolute file names lexically and using the names to disambiguate the mount table entries when evaluating `..` at a mount point combine to maintain the lexical definition of `..` efficiently.

Cleaning names

The lexical processing can generate names that are messy or redundant, ones with extra slashes or embedded `../` or `./` elements and other extraneous artifacts. As part of the kernel's implementation, we wrote a procedure, `cleanname`, that rewrites a name in place to canonicalize its appearance. The procedure is useful enough that it is now part of the Plan 9 C library and is employed by many programs to make sure they always present clean file names.

`Cleanname` is analogous to the URL-cleaning rules defined in RFC 1808 [Field95], although the rules are slightly different. `Cleanname` iteratively does the following until no further processing can be done:

1. Reduce multiple slashes to a single slash.
2. Eliminate `.` path name elements (the current directory).
3. Eliminate `..` path name elements (the parent directory) and the non-`.` non-`..`, element that precedes them.
4. Eliminate `..` elements that begin a rooted path, that is, replace `/..` by `/` at the beginning of a path.
5. Leave intact `..` elements that begin a non-rooted path.

If the result of this process is a null string, `cleanname` returns the string `"."`, representing the current directory.

The fd2path system call

Plan 9 has a new system call, `fd2path`, to enable programs to extract the `Cname` associated with an open file descriptor. It takes three arguments: a file descriptor, a buffer, and the size of the buffer:

```
int fd2path(int fd, char *buf, int nbuf)
```

It returns an error if the file descriptor is invalid; otherwise it fills the buffer with the name associated with `fd`. (If the name is too long, it is truncated; perhaps this condition should also draw an error.) The `fd2path` system call is very cheap, since all it does is copy the `Cname` string to user space.

The Plan 9 implementation of `getwd` uses `fd2path` rather than the tricky algorithm necessary in Unix:

```
char*
getwd(char *buf, int nbuf)
{
	int n, fd;

	fd = open(".", OREAD);
	if(fd < 0)
		return NULL;
	n = fd2path(fd, buf, nbuf);
	close(fd);
	if(n < 0)
		return NULL;
	return buf;
}
```

(The Unix specification of `getwd` does not include a count argument.) This version of `getwd` is not only straightforward, it is very efficient, reducing the performance advantage of a built-in pwd command while guaranteeing that all commands, not just pwd, see sensible directory names.

Here is a routine that prints the file name associated with each of its open file descriptors; it is useful for tracking down file descriptors left open by network listeners, text editors that spawn commands, and the like:

```
void
openfiles(void)
{
        int i;
        char buf[256];

        for(i=0; i<NFD; i++)
                if(fd2path(i, buf, sizeof buf) >= 0)
                        print("%d: %s\n", i, buf);
}
```

Uses of good names

Although pwd was the motivation for getting names right, good file names are useful in many contexts and have become a key part of the Plan 9 programming environment. The compilers record in the symbol table the full name of the source file, which makes it easy to track down the source of buggy, old software and also permits the implementation of a program, src, to automate tracking it down. Given the name of a program, src reads its symbol table, extracts the file information, and triggers the editor to open a window on the program's source for its main routine. No guesswork, no heuristics.

The openfiles routine was the inspiration for a new file in the /proc file system [Kill84]. For process *n*, the file /proc/*n*/fd is a list of all its open files, including its working directory, with associated information including its open status, I/O offset, unique id (analogous to i-number) and file name. Here is the contents of the fd file for a process in the window system on the machine being used to write this paper:

```
% cat /proc/125099/fd
/usr/rob
  0 r  M 5141 00000001.00000000        0 /mnt/term/dev/cons
  1 w  M 5141 00000001.00000000       51 /mnt/term/dev/cons
  2 w  M 5141 00000001.00000000       51 /mnt/term/dev/cons
  3 r  M 5141 0000000b.00000000     1166 /dev/snarf
  4 rw M 5141 0ffffffc.00000000      288 /dev/draw/new
  5 rw M 5141 00000036.00000000  4266337 /dev/draw/3/data
  6 r  M 5141 00000037.00000000        0 /dev/draw/3/refresh
  7 r  c    0 00000004.00000000  6199848 /dev/bintime
%
```

(The Linux implementation of /proc provides a related service by giving a directory in which each file-descriptor-numbered file is a symbolic link to the file itself.) When debugging errant systems software, such information can be valuable.

Another motivation for getting names right was the need to extract from the system an accurate description of the mount table, so that a process's name space could be recreated on another machine, in order to move (or simulate) a computing environment across the network. One program that does this is Plan 9's cpu command, which recreates the local name space on a remote machine, typically a large fast multiprocessor. Without accurate names, it was impossible to do the job right; now /proc provides a description of the name space of each process, /proc/*n*/ns:

```
% cat /proc/125099/ns
bind  / /
mount −aC #s/boot /
bind  #c /dev
bind  #d /fd
bind −c #e /env
bind  #p /proc
bind −c #s /srv
bind  /386/bin /bin
bind −a /rc/bin /bin
bind  /net /net
bind −a #l /net
mount −a #s/cs /net
mount −a #s/dns /net
bind −a #D /net
mount −c #s/boot /n/emelie
bind −c /n/emelie/mail /mail
mount −c /net/il/134/data /mnt/term
bind −a /usr/rob/bin/rc /bin
bind −a /usr/rob/bin/386 /bin
mount  #s/boot /n/emelieother other
bind −c /n/emelieother/rob /tmp
mount  #s/boot /n/dump dump
bind  /mnt/term/dev/cons /dev/cons
...
cd /usr/rob
%
```

(The # notation identifies raw device drivers so they may be attached to the name space.) The last line of the file gives the working directory of the process. The format of this file is that used by a library routine, `newns`, which reads a textual description like this and reconstructs a name space. Except for the need to quote # characters, the output is also a shell script that invokes the user-level commands `bind` and `mount`, which are just interfaces to the underlying system calls. However, files like `/net/il/134/data` represent network connections; to find out where they point, so that the corresponding calls can be reestablished for another process, they must be examined in more detail using the network device files [PrWi93]. Another program, `ns`, does this; it reads the `/proc/`*n*`/ns` file, decodes the information, and interprets it, translating the network addresses and quoting the names when required:

```
...
mount −a '#s/dns' /net
...
mount −c il!135.104.3.100!12884 /mnt/term
...
```

These tools make it possible to capture an accurate description of a process's name space and recreate it elsewhere. And like the open file descriptor table, they are a boon to debugging; it is always helpful to know exactly what resources a program is using.

Adapting to Unix

This work was done for the Plan 9 operating system, which has the advantage that the non-hierarchical aspects of the name space are all known to the kernel. It should be possible, though, to adapt it to a Unix system. The problem is that Unix has nothing corresponding precisely to a `Channel`, which in Plan 9 represents the unique result of evaluating a name. The `vnode` structure is a shared structure that may represent a file known by several names, while the `file` structure refers only to open files, but for example the current working directory of a process is not open. Possibilities to address this discrepancy include introducing a `Channel`-like structure that connects a name and a `vnode`, or maintaining a separate per-process table that maps names to `vnodes`, disambiguating using the techniques described here. If it could be done the result would be an implementation of `..` that reduces the need for a built-in `pwd` in the shell and offers a consistent, sensible interpretation of the 'parent directory'.

We have not done this adaptation, but we recommend that the Unix community try it.

Conclusions

It should be easy to discover a well-defined, absolute path name for every open file and directory in the system, even in the face of symbolic links and other non-hierarchical elements of the file name space. In earlier versions of Plan 9, and all current versions of Unix, names can instead be inconsistent and confusing.

The Plan 9 operating system now maintains an accurate name for each file, using inexpensive lexical operations coupled with local file system actions. Ambiguities are resolved by examining the names themselves; since they reflect the path that was used to reach the file, they also reflect the path back, permitting a dependable answer to be recovered even when stepping backwards through a multiply-named directory.

Names make sense again: they are sensible and consistent. Now that dependable names are available, system services can depend on them, and recent work in Plan 9 is doing just that. We—the community of Unix and Unix-like systems—should have done this work a long time ago.

Acknowledgements

Phil Winterbottom devised the `ns` command and the `fd` and `ns` files in `/proc`, based on an earlier implementation of path name management that the work in this paper replaces. Russ Cox wrote the final version of `cleanname` and helped debug the code for reversing the mount table. Ken Thompson, Dave Presotto, and Jim McKie offered encouragement and consultation.

References

[Field95] R. Fielding, ''Relative Uniform Resource Locators'', *Network Working Group Request for Comments: 1808*, June, 1995.

[Kill84] T. J. Killian, ''Processes as Files'', *Proceedings of the Summer 1984 USENIX Conference*, Salt Lake City, 1984, pp. 203-207.

[Korn94] David G. Korn, ''ksh: An Extensible High Level Language'', *Proceedings of the USENIX Very High Level Languages Symposium*, Santa Fe, 1994, pp. 129-146.

[Korn00] David G. Korn, personal communication.

[PeMc95] Jan-Simon Pendry and Marshall Kirk McKusick, ''Union Mounts in 4.4BSD-Lite'', *Proceedings of the 1995 USENIX Conference*, New Orleans, 1995.

[Pike91] Rob Pike, ''8½, the Plan 9 Window System'', *Proceedings of the Summer 1991 USENIX Conference*, Nashville, 1991, pp. 257-265.

[Pike93] Rob Pike, Dave Presotto, Ken Thompson, Howard Trickey, and Phil Winterbottom, ''The Use of Name Spaces in Plan 9'', *Operating Systems Review*, **27**, 2, April 1993, pp. 72-76.

[PrWi93] Dave Presotto and Phil Winterbottom, ''The Organization of Networks in Plan 9'', *Proceedings of the Winter 1993 USENIX Conference*, San Diego, 1993, pp. 43-50.