An Efficient File Hierarchy Walker

Glenn S. Fowler
David G. Korn
K.-Phong Vo

AT&T Labs - Research
Murray Hill, NJ 07974

ABSTRACT

This paper presents an interface specification and an efficient implementation of a general purpose library routine, *ftwalk*, to traverse a UNIX* file system hierarchy. A number of standard file system utilities, e.g., **find**, **ls**, **rm**, and others have been reimplemented using *ftwalk*. The total source code size is 30% smaller and the efficiency of all commands improves. More importantly, these commands now handle the file system search in a uniform, robust, and secure manner.

New tools have been built with *ftwalk*. A file system perusing tool, **tw**, will be described. **tw** subsumes the functionality of **find** and **xargs**. Further, it provides a powerful expression language with a syntax similar to C. For typical applications in which commands are executed on generated file names, **tw** is 5 to 10 times faster than **find**. The combination of a powerful language and performance efficiency in **tw** should reduce the practice of adding directory recursion to commands.

1. Introduction

Many standard file system utilities such as **ls**, **rm**, **find** and others either require or provide options to traverse file hierarchies exhaustively. Given that traversing a file hierarchy is a common operation, it is desirable to have a standard subroutine implementation for commands that require this functionality. There have been attempts at providing such a routine[ftw][Lin]. In particular, System V and 9th Edition C libraries provide different implementations of a routine, *ftw*, designed for this purpose. Surprisingly, no attempt has been made to use the routine in implementing the standard commands (except for 9th Edition **du**). As a result, each command has its own file system walker with a separate set of quirks and problems. As examples, the 4.3BSD **ls** command cycles forever if there is a loop in the file hierarchy caused by symbolic links, the **rm** command cannot remove a file hierarchy a few hundred levels deep, and the **find** command cannot search a file hierarchy deeper than the number of available file descriptors. Inertia is a likely cause for not using a common file system walker in standard commands. However, on closer inspection, we also found that current implementations of *ftw* are either too inefficient for general use or simply inadequate in the interface to support the variety of requirements from different commands. The interface inadequacy is further stressed when we want to build commands that are portable across different UNIX flavors such as System V and 4.3BSD that have wildly different file system structures. For these reasons, we decided to design and implement a new file system walking routine. It seems reasonable that such a routine should exhibit the following qualities:

- *Generality*: The routine must be general enough to support all current standard file system utilities and other tools that we can think of.
- *Efficiency*: The routine must perform about as well as its hand-coded counterparts in current commands.

* UNIX is a registered trademark of AT&T.

- *Finiteness*: The routine should terminate even in the event of cycles caused by directory links and/or symbolic links. Such cycles must be detected and avoided.
- *Robustness*: The routine should work regardless of the depth and extent of the search. For example, the depth of a file hierarchy should not in itself be a limiting factor. John Linderman pointed out to us that this is an important security consideration in preventing hostile users from creating deep files that may be hidden from current file system utilities.
- *Portability*: The routine should work on all systems that we have access to, e.g., System V, BSD, HP/UX, 9th Edition. In addition, it should be conformant to POSIX 1003.1[IEEE].

We have designed and built a file walking routine *ftwalk* that satisfies the above quality requirements. The extensive interface features of *ftwalk* were carefully designed so that all standard utilities (**cp**, **du**, **find**, **ls** and **rm**) could be rebuilt using *ftwalk*. We have, in fact, rebuilt these commands. Inheriting the same qualities in *ftwalk*, the new commands handle file system traversal in a consistent and robust manner. In addition, their efficiency is at least as good or much better than their earlier counterparts. The generality of *ftwalk* also enables the construction of new tools. An example is **tw**, a new command that both generalizes and subsumes the functionality of **find** and **xargs**. A novel feature of **tw** is a powerful expression language with a C-like syntax. For recursive execution of commands on a file hierarchy, **tw** is typically between 5 to 10 times faster than **find**. Comparing with commands such as **chmod** and **chgrp**, **tw** comes to within 20% in system time and about the same on CPU time.

The rest of the paper is organized as follows. Section 2 describes *ftwalk*. Section 3 discusses issues and problems in reimplementing standard utilities. Section 4 describes **tw**. Finally, in the conclusion, we indicate some future directions.

2. The *ftwalk* Routine

2.1 Interface Specification

The interface of *ftwalk* contains many options designed for two purposes: *generality* and *efficiency*. Generality means that the interface must provide enough abstractions to support all commands that we know of or want to build. Often, a general abstraction can prevent efficient implementations of particular commands. In such cases, there are features allowing an application to ''control'' an abstraction to achieve the desired level of efficiency. For example, by default, *ftwalk* gets file status of all objects as soon as they are found. This enables the walker to determine terminal objects such as files or unreadable directories and process them immediately. However, getting status information of objects is an expensive operation. Thus, an option `FTW_DELAY` is provided to control this feature for a command such as **ls** that may not need such information for directory children (e.g., in a plain **ls** call).

Figure-1 shows the function prototypes of *ftwalk* and the user-supplied functions *userf* and *comparf* and the basic data structure `struct FTW`. In a call to *ftwalk*, a depth-first search starts at the directory `path`. Though there are many different graph search techniques[AHU], depth-first search is the common search method in all the commands that we studied. In the case of commands such as **rm** and **du**, a postorder depth-first search is, in fact, the method of choice. This is because the descendants of a directory must be processed before the directory itself can be processed.

The user-supplied function, *userf*, is called at visits of objects. It has a single argument, a pointer to a `struct FTW` structure that stores information on the object being visited. *userf* is called exactly once on a terminal object such as a file or a directory causing cycles, For a normal directory, *userf* may be called once or twice in preorder and/or postorder. As discussed below, it is possible to prune a search after a preorder visit or to restart a search after a postorder visit by setting appropriate values in the `ftw->status` field in *userf*. *userf* should normally return `0`. A non-zero return value from *userf* causes *ftwalk* to return immediately with the same value.

The function, *comparf*, if supplied, is used to order the original paths (see `FTW_MULTIPLE` below) and to order the children of other encountered directories. Note that such an ordering directs the search. This is important for commands such as **ls** that need to traverse a file hierarchy in some specific order. *Comparf* is called with two arguments, `ftw1` and `ftw2`, that point to the `struct FTW` structures of the objects being

```
int ftwalk(char *path, int (*userf)(), int flags, int (*comparf)());
int userf(struct FTW *ftw);
int comparf(struct FTW *ftw1, struct FTW *ftw2);

struct FTW
{
      struct FTW        *link;
      struct FTW        *parent;
      void              *local;
      struct stat       *statb;
      char              *path;
      short             pathlen;
      unsigned short    info;
      unsigned short    status;
      short             level;
      short             namelen;
      char              name[];
};
```

Figure 1. *Ftwalk* and related functions and data structures

compared. *Comparf* should return a negative, zero, or positive integer to indicate whether `ftw1` is considered smaller, equal or larger than `ftw2`.

The argument `flags` of *ftwalk* is a bit vector. Currently, the following bits are supported:

`FTW_DOT`: *ftwalk* will not attempt to change directory (*chdir*) while recursively searching directories. Thus, throughout the search, the current directory (dot) stays the same. `FTW_DOT` should not be used often since changing directory during a search is a major efficiency gain in system time. `FTW_DOT` is needed for applications that try to avoid undesirable file system side effects (e.g., core dumps) in unexpected places. Generally, changing directory is only an option, not a requirement. A successful search depends only on appropriate read permission, not search permission.

`FTW_CHILDREN`: Preorder visits of directories are implied even if `FTW_POST` (below) is turned on. In such a visit, the `ftw->link` field contains a linked list of all children objects of the directory being visited. This option is useful for commands (e.g., **ls**) that need the children list for local processing purposes (e.g., computing multi-column format).

`FTW_DELAY`: When `FTW_CHILDREN` is turned on, the fields `ftw->statb` (`struct stat`) of children objects remain undefined until these objects are visited. This helps a command such as **ls** to avoid expensive computations when only the names of the children of a directory are needed.

`FTW_MULTIPLE`: The argument `path` of *ftwalk* is really of type `char**` and points to a null-terminated array of path names. *Ftwalk* will search all hierarchies rooted at these paths.

`FTW_PHYSICAL`: Symbolic links will not automatically be followed. Each symbolic link object will initially be treated as a terminal object. The user function *userf* may ask *ftwalk* to follow the link by setting the field `ftw->status` to `FTW_FOLLOW` (below).

`FTW_POST`: For non-terminal directories, *userf* is called only on postorder visits. Note that if `FTW_CHILDREN` is turned on, preorder calls to *userf* will always be performed.

`FTW_TWICE`: For non-terminal directories, *userf* is called twice in both preorder and postorder.

The use of some of the fields in the structure `struct FTW` have been mentioned. Below are detailed descriptions of these fields.

`link`: This field is used in two different ways. For a directory that causes cycles, `link` points to the object on the current search path that is identical to this directory. For any other directory, in an `FTW_CHILDREN` preorder visit, `link` points to the list of children of the directory.

`parent`: This field points to the parent directory of the object being visited. Thus, ancestors of an object can be traced via the `parent` pointers. For convenience, a parent structure is also provided for level 0 objects. It contains the same file status information as that of its child.

`local`: This field is provided to users to store any information local to the object being visited. It is initialized to `NULL` by *ftwalk*. Note that `local` is wide enough to store pointers. It is the user's responsibility to free heap allocated data stored in the `local` fields. This can be done, for example, in a postorder visit.

`statb`: This field stores the `struct stat` buffer containing all file system information available on the object. Note that if the option `FTW_DELAY` is turned on, such information is undefined when the object is on the children list of a directory being visited in preorder (`FTW_CHILDREN`).

`path` and `pathlen`: These fields contain the path name and the path name length of the object being visited. The `path` buffer is guaranteed to be large enough so that the name of any child object of the current object can be appended.

`info`: The type of the object being visited. Following are the types:

`FTW_NS`:	The type of this object is unknown because *stat* or *lstat* failed.
`FTW_F`:	This object is a file.
`FTW_SL`:	This object is a symbolic link.
`FTW_D`:	This object is a directory being visited in preorder.
`FTW_DC`:	This object is a directory that causes cycles.
`FTW_DNR`:	This object is a directory that is not readable.
`FTW_DNX`:	This object is a directory that is readable but not searchable.
`FTW_DP`:	This object is a directory that is being visited in postorder.

`status`: This field is used in two different ways. In the call to *userf*, it is set to either `FTW_NAME` or `FTW_PATH` to indicate whether the user function should use the base `name` or the `path` name to get file system status of the object. This is important for correct processing when the option `FTW_DOT` is off. If `FTW_DOT` is on, the value of `status` is always `FTW_PATH`. In return, `userf` can set `status` to one of the below values:

`FTW_AGAIN`: If this is a postorder visit, descend the hierarchy rooted at this object again.

`FTW_NOPOST`: This is usually set in a preorder visit of an object. It suppresses any postorder visit to this object.

`FTW_FOLLOW`: If this is a symbolic link, follow it.

`FTW_SKIP`: Prune the search at this object. Thus, descendants of this object will not be visited.

`FTW_STAT`: This value can be assigned to the `status` fields of children of the current object when `FTW_DELAY` and `FTW_CHILDREN` were turned on. It means that the `statb` structure of the respective child has been filled by *userf*.

`level`: This field contains the depth from the root object of the object being visited.

`name` and `namelen`: These fields contain the base name and name length of the current object.

2.2 The *ftwalk* Search Algorithm

ftwalk employs a non-recursive depth-first search to walk file hierarchies. Using a non-recursive version of depth-first search allows us to handle various exceptions in a clean manner. In a recursive implementation, an exception may necessitate the unraveling of several recursion layers with extra book-keeping. The non-recursive algorithm also simplifies optimizations such as reducing the number of back-up *chdir* calls by eliminating the need to use static variables to retain states across recursive calls.

```
 0.   struct FTW *todo, *ftw, *first, *last, *f;
 1.   todo = top level path;
 2.   while(todo)
 3.   {
 4.           ftw = todo;
 5.           if(preorder processing is needed for ftw)
 6.                   userf(ftw);
 7.           if(ftw is a directory that does not cause cycles)
 8.           {
 9.                   first = last = NULL;
10.                   for(each child f of ftw)
11.                   {
12.                           if(first == NULL)
13.                                   first = last = f;
14.                           else  last = last->link = f;
15.                   }
16.                   if(last)
17.                   {
18.                           last->link = todo;
19.                           todo = first;
20.                   }
21.           }
22.           while(todo && todo == ftw)
23.           {
24.                   if(postorder processing is needed for ftw)
25.                           userf(ftw);
26.                   ftw = ftw->parent;
27.                   todo = todo->link;
28.           }
29.   }
```

Figure 2. A non-recursive file system depth-first search

Ignoring detailed interface features such as turning off FTW_DOT and sorting objects, Figure-2 shows a skeleton of the non-recursive depth-first search algorithm in a quasi-C syntax. Note that internal to the algorithm, the `link` field of `struct FTW` is used to maintain the list of elements being searched. The following discussion is based on the line numbers in Figure-2.

`0-1`: These statements declare relevant variables and initialize the `todo` list with the root path of the hierarchy to be searched.

`2-29`: This is the main loop of the algorithm. The variable `todo` always points to the element to be processed next. The `todo` list is basically a stack in which the last element put on it is the next element to be processed.

`4-6`: The variable `ftw` is set to the element to be processed. If a preorder processing of `ftw` is required, it is done unless `ftw` is a directory and the option `FTW_CHILDREN` is turned on. In that case, the code on lines 5 and 6 for preorder processing is actually done after line 18 following the construction of the children list. If `FTW_DOT` is off, the current directory is set to `ftw` before its children are read. This speeds up various system calls such as *stat* or *open*.

`9-20`: The list of children of `ftw` is constructed. Since all children of a directory are read, the algorithm only needs one open file pointer at any given time. For efficiency, if *comparf* is given, the children of a directory are sorted using a stable insertion sort based on a self-adjusting binary tree[ST]. A similar tree is also used for efficient checking of cycles.

`22-28`: The stack is popped. This corresponds to subroutine returns in a recursive implementation of depth-first search. If post-order processing of an element being popped from the stack is required, the user function *userf* is called. Though not shown in the code, The number of popped levels is also recorded to optimize the number of *chdir* calls to back up the current directory.

2.3 An Example

In the appendix, we show the code of a user function, *draw*, to generate the description of a file hierarchy in the DAG language[GNV]. Figure-3 shows a generated DAG drawing of a small hierarchy. In the drawing, boxes indicate directories, ellipses files, and diamonds symbolic links. Dotted edges represent either symbolic links or hard links.

Figure 3. A file hierarchy

Following is the code fragment to invoke *ftwalk* to walk and draw a given file hierarchy:

```
printf(".GS\n");
ftwalk(path,draw,FTW_PHYSICAL,NULL);
printf(".GE\n");
```

The two `printf` statements generate the prologue and epilogue of a DAG description. The root of the hierarchy to be drawn is given in `path`. `FTW_PHYSICAL` is turned on to detect symbolic links. Since the relative order of directory entries is not important, `comparf` is set to `NULL`.

3. Reimplemented Standard Commands

A good way to test the applicability of *ftwalk* is to reimplement standard utilities that peruse file systems. This allows performance, feature and coding style comparisons. Further motivating factors are to enforce consistency and to enhance robustness across these commands:

Logical vs. Physical: A logical search automatically follows symbolic links to their physical counterparts while a physical search stops at symbolic links. With *ftwalk*, both types of search can be provided as command options without too much programming effort.

Cycle Detection: Cycles caused by both symbolic and hard links on directories are automatically detected. This allows commands to default to the logical search mode without fear of non-termination, a major problem on systems with symbolic links.

Robustness: A good deal of effort went into *ftwalk* to remove artificial constraints. As an example, *ftwalk* can handle directory structures of arbitrary depths (up to virtual or hard memory limits). Current implementations of important standard utilities such as **find** and **rm**, in fact, fail on deep directory structures.

An informal survey of BSD, System V and 9th Edition machines produced the following commands that do some form of file hierarchy traversal: **chgrp**, **chmod**, **cp**, **diff**, **du**, **find**, **ls**, **rm**, and **tar**. Instead of blindly doing wholesale changes to these commands, we study their interfaces and the interrelationships in their use. This allows us to reimplement and extend certain commands selectively while leaving others untouched. We discuss below the issues and problems in rebuilding these commands.

chgrp and **chmod**: The **–R** (recursive) option added to the BSD versions of **chgrp** and **chmod** can be simulated by using these commands in conjunction with **find** as in the following shell[BK] statements:

```
find dir -exec chmod permissions "{}" ";"
chmod permissions $(find dir -print)
```

The first statement always works but it is slow, since a separate **chmod** process is created for each generated file name. The second command is faster but it does not always work since it may exceed the argument limit to the system call *exec*. To avoid the temptation of adding `-R` to all our favorite commands (‘‘`grep -R pattern .`’’ might be nice), we changed the `-exec` primitive of **find** to accept + (plus) as a command terminator in addition to the usual **;** (semicolon). The + instructs **find** to execute the command a minimal number of times in the manner of **xargs** (see also **tw** below).

cp: The `-r` option allows copying directory hierarchies (9th Edition, provides **reccp** to accomplish the same thing). This operation is better accomplished by the POSIX ‘‘**pax –rw**’’ command[IEEE] (similar to ‘‘**cpio –p**’’) since **pax** provides better control over the attributes of the copied files. With this in mind, **cp** was originally redone to execute ‘‘**pax –rw**’’ for the recursive case. However, a recoding of **cp** using *ftwalk* is more efficient even in the non-recursive case by taking advantage of the information provided in the `struct FTW` structure of *ftwalk*.

diff: This command has an option to compare two directory hierarchies. This exposes a limitation in a function interface such as *ftwalk* that cannot allow simultaneous explorations of two or more different hierarchies. We shall come back to this issue in the conclusion.

du: This command is simplest of all the commands to reimplement. In reimplementing it, we found the need to have a place to store summaries of information of descendants of directories on a search path. The `ftw.local` field of `struct FTW` fulfills this need. The simplicity of **du** is one of the guiding factors in the design of the **tw** expression evaluator described below.

find: This is an indispensible command but it commits a terrible sin of robustness: it will not search beyond a depth higher than the number of available file descriptors (about 20 for our system). This means that files in relatively shallow hierarchies may never be found. Further, the expression interface of **find** is messy, making any extension difficult. The new command `tw` described in Section 4 corrects these problems.

ls: The column output mode (`-C` option) of this command requires that all children of a directory are available when it is visited in preorder. The option `FTW_CHILDREN` of *ftwalk* solves this problem. In addition, providing a sort function, *comparf*, to *ftwalk* efficiently solves the problem of sorting directory entries.

rm: This command varies the most among different implementations, none of which deal with hard link directories properly. Although hard links to directories are rare, a standard command in `/bin` should be able to handle them. There are two different system calls to remove file entries, *rmdir* and *unlink*. Roughly, *unlink* simply removes an entry while *rmdir* removes a directory entry and the implied ‘‘hard links’’ **.** (dot) and **..** (dot-dot). In the presence of hard links, care must be taken as to which of these system call to use. Our reimplemented **rm** solves this problem. Using *ftwalk* also solves a robustness problem with previous implementations where deep hierarchies cannot be removed.

tar: On our system, **tar** is replaced by a local implementation of the POSIX command **pax**, which uses *ftwalk* for file system traversal.

Table-1 and Table-2 summarize comparative performance statistics for the commands: **cp**, **du**, **find**, **ls** and **rm**. Table-1 gives statistics for the old commands while Table-2 gives statistics for the reimplemented commands. The statistics include numbers of non-commented source lines (NCSL), and CPU times, System times, and their total. The times are measured in seconds and result from running the programs on `/usr/src` on our local system which is a VAX 8650 running 4.3BSD. This directory system has 201 directories and a total of 3480 entries. To reduce statistical fluctuations, each timing result is an average of three runs at night when the machine is largely idle.

c c c c c
c r r r r.

Program	NCSL	CPU	System	CPU+Sys
cp	211	1.72	37.06	38.78
du	152	.47	3.50	3.97
find	1053	.29	3.65	3.94
ls	616	1.60	5.29	6.89
rm	218	.91	17.90	18.81
Total	2250	4.99	67.40	72.39

Table 1. Statistics of old commands

c c c c c
c r r r r.

Program	NCSL	CPU	System	CPU+Sys
cp	208	1.02	36.27	37.29
du	124	.33	3.46	3.79
find	676	.48	3.44	3.92
ls	499	.75	3.80	4.55
rm	184	.62	12.28	12.90
Total	1691	3.20	59.25	62.45

Table 2. Statistics of reimplemented commands

As Table-1 and Table-2 show, overall, there is a reduction of about 30% in source code size. Not all of the code reduction is due to using *ftwalk* since the commands have been rewritten extensively. The combined time performance for all commands improves about 15%. Major performance gains are in **ls** and **rm**. Both of these commands benefit from *ftwalk*'s efficient utilization of system calls. The more than 100% improvement in CPU time for **ls** is due to *ftwalk*'s use of an efficient search tree for sorting directory children.

4. The tw Command

The **find** command is a popular tool for recursive processing of objects in file hierarchies. However, **find** has several drawbacks that limit its use. The most visible problems are **find**'s unconventional expression syntax and its use of shell meta-characters as arguments which necessitates awkward quoting. Worse than **find**'s awkward interface are useful options whose semantics cause extreme inefficiency. For example, the `−exec` option requires a separate invocation of the specified program for each generated file name.

To alleviate the efficiency problem in **find**, System V provides **xargs**, a command which reads file names, one per line, and applies a given command to as many files at one time as possible. As an example, below are two shell command lines that perform more or less the same function though the second one is more efficient on a large directory structure.

```
find path -exec program "{}" ";"
find path -print | xargs program
```

A problem with using **xargs** in this manner is that file names that contain the `\n` (new-line) character cannot be handled. This is a potential security problem as the operating system does not preclude the creation of such files.

Aside from **find** and **xargs**, a popular approach in BSD is to add a recursive option to individual commands. This requires modification of every command of interest and lacks the useful pattern matching mechanism of **find** for selective command execution.

Inadequacies in current approaches expose the need for a new language and tool to interface to the file system structure that is: flexible to use, efficient, and easily extensible to match frequent changes in file system interface structures. In this spirit, we created a new command, **tw**, whose interface language is C-like and provides extensive access to file information. Like **find**, **tw** recursively traverses a file hierarchy and performs actions on visited files. Like **xargs**, **tw** only executes a given command a minimal number of times. The rest of this section describes **tw** and gives examples of how to use it.

4.1 Interface Description

Without arguments, **tw** is equivalent to ‘‘`find . -print`’’. When given a command, **tw** executes the command with the generated file names as arguments. The given command may be executed more than once depending on the limit of the *exec* system call. Similar to **xargs**, **tw** can also read file names from the standard input if its first argument is given as `-`. As an example, below are four commands that performs equivalent functions:

```
tw chmod o-w
chmod -R o-w
find . -print | xargs chmod o-w
find . -print | tw - chmod o-w
```

Conditional C-style expressions may be specified with the `-e` option. The expression operands are the fields of the `struct FTW` and `struct stat` structures (the prefix ‘‘`st_`’’ is deleted) of the object being visited. Here is a command to list files with *group* or *other* write permissions:

```
tw -e "mode & 022"
```

Some fields allow `"..."` and `'...'` string operands. The first command below is equivalent to the above command but somewhat more readable. The second command lists all objects modified since yesterday morning (most common time and date expressions are allowed).

```
tw -e "mode & 'go+w'"
tw -e "mtime >= 'yesterday 8am'"
```

The `'...'` strings are handy when combined with `"` shell quoting. Strings may be compared, where `==` and `!=` treat the right hand operand as a **ksh**[BK] file match pattern. The following command executes `grep pattern` on a selected collection of source files:

```
tw -e "name == '*.[chly]'" grep pattern
```

For convenience, the type bits of the field `st_mode` in `struct stat` are placed in a separate field. For example, the following lists all subdirectories of the current directory:

```
tw -e "type == DIR"
```

Identifiers may be prefixed by one or more ‘‘`parent.`’’ to reference parent directory status information. The below command prunes all directories that are in a different file system type (e.g., *nfs* vs. *rfs*) than their parent directories:

```
        tw -e "if (fstype != parent.fstype) status = SKIP"
```

Explicit file references can also be done by prefixing an identifier with '`filename`'. Following is an example script that lists all files newer than the file `/etc/backup.time`.

```
        tw -e "mtime > '/etc/backup.time'.mtime"
```

The complete list of predefined identifiers appears below. Most correspond to the `st_` elements of `struct stat`. New fields can be added as `struct stat` grows.

`atime`: Most recent access time. If `atime`, `ctime`, or `mtime` is the left hand side operand of a binary operator, the right hand side operand may be a string interpreted as a date expression.

`blocks`: The number of blocks in the file.

`ctime`: The inode change time.

`dev`: The device number.

`fstype`: The file system type name. The default value is `ufs`.

`gid`: The group id number. If `gid` is the left hand side operand of a binary operator, the right hand side operand may be a string that is interpreted as a group name.

`gidok`: `1` if `gid` is a valid group id in the system database, `0` otherwise.

`ino`: The inode or file serial number.

`level`: The depth of the file relative to the starting directory.

`local`: The `local` field of `struct FTW`. This field may be assigned certain values.

`mode`: The identifier `FMT` is recognized and can be used to mask the file format and permission bits. If `mode` is the left hand side operand of a binary operator, the right hand side operand may be a string that is interpreted as a **chmod** permission expression.

`mtime`: The time at which the object was last modified.

`name`: The file name of the object with no directory prefix.

`nlink`: The number of hard links.

`path`: The full pathname of the current object. Note that this is only defined for the current object. For example, `parent.path` is undefined.

`perm`: The file permission bits of `mode`. If `perm` is the left hand side operand of a binary operator, the right hand side operand may be a string that is interpreted as a **chmod** permission expression.

`rdev`: The device numbers for special files.

`size`: The number of bytes in the file.

`status`: The `status` field of `struct FTW`. This field may be assigned one of: `AGAIN`, `FOLLOW`, `NOPOST`, and `SKIP` which correspond to the *ftwalk* constants prefixed by ‘‘`FTW_`’’. For example, `status=SKIP` may be used to prune subdirectories from the search.

`type`: The file type bits of `mode`. The identifiers `BLK`, `CHR`, `DIR`, `FIFO`, `LNK`, `REG`, and `SOCK` are recognized.

`uid`: The user id number. If `uid` is the left hand side operand of a binary operator, the right hand side operand may be a string that is interpreted as a user name.

`uidok`: `1` if `uid` is a valid user id in the system database, `0` otherwise.

`visit`: A variable (initialized to `0`) that can be updated on each visit to the file. A file is uniquely identified by its `st_dev` and `st_ino` numbers.

Expressions in **tw** are labeled to indicate different types of processing. The default label for an expression is `select`. Possible labels are:

`action`: This expression defines the action to be executed on selected file names (below). The default action is to list the selected file names on the standard output.

`begin`: If defined, this expression is evaluated once before the file system traversal starts.

`end`: If defined, this expression is evaluated once after the file system traversal ends.

`select`: This expression is used to select files to be acted upon. If it evaluates to true (non-zero) then the `action` expression is evaluated. The default value for `select` is `1`.

`sort`: This expression specifies an identifier by which child entries are sorted. The below example lists files sorted by newest to oldest `mtime` (the `!` inverts the sort sense).

```
tw -e "sort: !mtime"
```

Finally, `if-else`, `printf` and variable declarations round out the expression language.

4.2 Examples and Discussions

Figure-4 shows a **tw** script to list the name and inode number of each file and to list the total number of files encountered. Note that the variable `count` is automatically initialized to `0`.

```
tw -e "
      int count;
action:
      count++;
      printf('name=%s inode=%08ld\n', name, ino);
end:
      printf('%d file%s\n', count, count == 1 ? '' : 's');
"
```

Figure 4. A **tw** script to list file names and inode numbers

Figure-5 shows a script to emulate the **du** command. Here, the `-p` option specifies a postorder traversal and the `-P` option specifies a physical (don't follow symbolic links) traversal. Note how the `local` field is used to accumulate the total number of blocks contained in a directory subtree and the `visit` field check is used to ensure that hard links are counted only once. This script is about 40% slower than the standard **du** command, where 33% of the difference is in CPU time. This is reasonable since the `action` expression must be evaluated interpretively for each file in the search.

```
tw -pP -e "
action:
      if (visit++ == 0)
      {
            parent.local += local + blocks;
            if (type == DIR)
                  printf('%d\t%s\n', local + blocks, path);
      }
"
```

Figure 5. A **tw** script to emulate **du**

To measure system time overhead caused by the *fork* and *exec* system calls in **tw** and combinations of **find** and **xargs**, we ran tests on `/usr/src`, a large hierarchy, using the below three commands:

```
    tw -d /usr/src null
    find /usr/src -exec null ";"
    find /usr/src -print | xargs null
```

Here, **null** is a command that does nothing. Using **null** instead of another standard command such as **wc** or **ls** allows us to measure only the system time overhead caused by *fork* and *exec* and not the cost of ‘‘real’’ work being performed by these commands. Table-3 shows the comparative timing results. The **tw** version is about 10 times faster than the **find** version and about 6.5 times faster than the combination of **find** and **xargs**. The relatively bad timing result for the **find** and **xargs** combination is largely due to the poor implementation of **xargs**.

c c c c
c r r r.

Program	CPU	System	CPU+Sys
tw	.55	5.45	6.00
find	1.46	55.81	57.27
find | xargs	5.39	34.44	39.83

Table 3. Statistics for **tw**, **find** and **xargs**

5. Conclusions

We presented a new file hierarchy traversal routine, *ftwalk*, and its use in reimplementing certain standard file system commands and in implementing new commands. The applicability of *ftwalk* in a wide variety of programs shows the generality of its interface. The total source code size for the reimplemented commands reduces by 30% despite the addition of new options in some cases. Timing results show that these commands perform at least as well as before. In cases such as **ls** or **rm**, performance improves significantly. More importantly, all commands handle the file hierarchy traversal in a consistent and robust manner. Robustness is crucial for security concerns. Certain important commands such as **find** and **rm** previously could not handle deep hierarchies. We have tested our versions of these commands on directory structures deeper than 2000 levels.

New commands such as **pax**, a new POSIX conformant file archiver, and **tw**, a file system walker, have been built based on *ftwalk*. The latter command, **tw**, was described here. It subsumes the functionality of **find** and **xargs**. In addition, **tw** provides a powerful expression language with a syntax that resembles the C language. The expression language for **tw** is designed to be easily extendible to match any future changes in the file system interface structures. The performance of **tw** is many times better than **find** when a given command must be executed against generated file names. For a command such as **chmod** (**chgrp**), `tw chmod` performs nearly as well as its hand-coded counterpart `chmod -R`. The `tw chmod` version is only about 20% slower in system time than the `chmod -R` version while CPU time is the same. The efficiency level and the availablility of a powerful language in **tw** should stop the practice of adding a recursive option to every command in sight.

Finally, one way of looking at *ftwalk* is that it outputs an ordered stream of objects from a file hierarchy. The order is defined by local orderings of children of directories and by the depth-first search. An interesting observation that arises from studying the non-recursive depth-first search algorithm of *ftwalk* is that the algorithmic steps are encoded in the `todo` data structure. This opens the possibility of implementing an interface along the line of *opendir* and *readdir* in which hierarchies are opened and read. The order in which an object is read will be exactly the same as the order that *ftwalk* would produce if it was called on the root of the respective hierarchy. This neatly solves a major problem for programs such as **diff** that need to manipulate multiple hierarchies simultaneously.

References

[AHU] A.V. Aho, J.E. Hopcroft and J.D. Ullman, *The Design and Analysis of Algorithms*, Addison-Wesley, 1974.

[GNV] E.R. Gansner, S.C. North, and K.P. Vo, *DAG, A Program to Draw Directed Graphs*, Soft. Prac. and Exp., 1988.

[BK] M. Bolsky and D.G. Korn, *The KornShell Command and Programming Language*, Prentice Hall, 1988.

[ftw] *ftw*, System V Programmer's Reference Manual and 9th Edition UNIX Time-Sharing System Programmer's Manual.

[IEEE] *The IEEE Standard Portable Operating System Interface for Computer Environments*, IEEE, 1988.

[Lin] J.P. Linderman, *dwalk*, Private communication, 1988.

[ST] D. Sleator and R.E. Tarjan, *Self-Adjusting Binary Trees*, Proc. 15th Ann. ACM Symp. on Found. of Comp., 1985.

Appendix

Below is a user function `draw` that can be used with *ftwalk* to draw file hierarchies.

```
#include    <ftw.h>

draw(ftw)
struct FTW  *ftw;
{
      int         linklev;    /* level of linked object */
      char        *shape,           /* shape of this object */
                  name[256],  /* unique name of this object */
                  link[256],  /* name of its linked object if any */
                  parent[256];      /* name of its parent */
      static int  N = 0;            /* to make unique name */

      /* follow symlink, set local so we know it was a symlink */
      if(ftw->info == FTW_SL)
      {
            ftw->status = FTW_FOLLOW;
            ftw->local = (VOID*) 1;
            return 0;
      }
      /* an object with no status */
      else if(ftw->info == FTW_NS)
      {
            linklev = -1;
            sprintf(name,"ns%d",++N);
      }
      else
      {     /* linkobj gets the level of the object linked to ftw.
               If there is no such object, it returns -1. */
            linklev = linkobj(ftw);

            /* use (dev,ino) to generate a unique name for this object */
            sprintf(name,"n%d_%d_%d",
                  ftw->statb.st_dev,ftw->statb.st_ino,
                  linklev >= 0 ? ++N : 0);
      }

      /* generate the draw node statement */
      shape = ftw->local ? "Diamond" : ftw->info == FTW_F ? "Ellipse" :
            ftw->info == FTW_NS ? "Plaintext" : "Box";
      printf("draw %s as %s label \"%s\";\n",name,shape,ftw->name);

      /* generate the (parent,child) edge statement */
      if(ftw->level > 0)
      {
            sprintf(parent,"n%d_%d_0",
                  ftw->parent->statb.st_dev, ftw->parent->statb.st_ino);
            printf("edge from %s to %s;\n",parent,name);
      }

      /* if this is a link, draw the link edge and prune the search */
      if(linklev >= 0)
      {
            sprintf(link,"n%d_%d_0",
                  ftw->statb.st_dev,ftw->statb.st_ino);
            if(linklev == ftw->level)
                  printf("same rank %s %s;\n",name,link);
            printf("%sedge from %s to %s dotted;\n",
                  ftw->level >= linklev ? "back" : "",name,link);
            ftw->status = FTW_SKIP;
      }

      return 0;
}
```

For comparison, below is a **tw** script that is computationally equivalent to *draw*

```
      tw -P -d $1 -e "
      begin:
            printf('.GS\n');
      end:
            printf('.GE\n');
      action:
            if (type == LNK)
            {
                  status = FOLLOW;
                  local = 1;
            }
            else
            {
                  printf('draw n%d_%ld_%d as ', dev, ino, visit);
                  if(local)
                        printf('Diamond');
                  else if(type == FILE)
                        printf('Ellipse');
                  else if(type == NS)
                        printf('Plaintext');
                  else  printf('Box');
                  printf(' label \"%s\";\n', name);
                  if(level > 0)
                        printf('edge from n%d_%ld_0 to n%d_%ld_%d;\n',
                              parent.dev,parent.ino,dev,ino,visit);
                  if(visit == 0)
                        visit = level+1;
                  else
                  {
                        if(visit == level + 1)
                              printf('same rank n%d_%ld_%d n%d_%ld_0;\n',
                                    dev,ino,dev,ino,visit);
                        printf('%sedge from n%d_%ld_%d to n%d_%ld_0 dotted;\n',
                              (visit >= level + 1) ? '' : 'back',
                              dev,ino,dev,ino,visit);
                        status = SKIP;
                  }
            }
      "
```

Figure-6 shows the generated DAG code corresponding to the hierarchy shown in Figure-3. In DAG, nodes are identified by unique strings and edges are defined by either an `edge` or a `backedge` statement. The `draw` statement defines the shape used to draw a node or collection of nodes and the labels for such nodes if they are different from their names.

```
.GS
draw n2324_141316_0 as Box label "/home/kpv/Src/TEST";
draw n2324_24634_0 as Box label "dir1";
edge from n2324_141316_0 to n2324_24634_0;
draw n2324_24637_0 as Ellipse label "file1";
edge from n2324_24634_0 to n2324_24637_0;
draw n2324_122913_0 as Box label "dir2";
edge from n2324_141316_0 to n2324_122913_0;
draw n2324_24637_1 as Ellipse label "file2";
edge from n2324_122913_0 to n2324_24637_1;
same rank n2324_24637_1 n2324_24637_0;
backedge from n2324_24637_1 to n2324_24637_0 dotted;
draw n2324_141316_2 as Diamond label "dir3";
edge from n2324_122913_0 to n2324_141316_2;
backedge from n2324_141316_2 to n2324_141316_0 dotted;
.GE
```

Figure 6. DAG description of a file hierarchy