Rapid location of mount points

JONATHAN M. SMITH

Computer Science Department, Columbia University
New York, New York 10027

SUMMARY

''Mount points'' allow more storage to be grafted into tree-structured hierarchical file systems. Administrative tasks use their locations, which are tabulated in a file. In our System V UNIX environment, this file was occasionally removed. *Getmnt* **was written to recover the information.**

Getmnt **has had three significant versions. The original version (***getmnt1***) was a highly optimized naive tree traversal.** *Getmnt2* **improved the real time performance by a mean factor of 7 by pruning unnecessary branches from the traversal.** *Getmnt3* **doubled** *getmnt2's* **speed, with a change from depth-first to breadth-first search. On our development system,** *getmnt1* **required 647.6 seconds to run, while** *getmnt3* **required 42.53 seconds.**

KEY WORDS Mount Point File System Search Performance

INTRODUCTION

The UNIX® file system[1-3] manages collections of data called *files*. Data and a file control block called an *inode* comprise a UNIX file. *Inodes* are stored in a list maintained per device. A pointer to the list entry, the *i-number*, identifies the file. *Directories* are lists of <*name, i-number*> pairs. Directories are files, and can be components of another directory. A rooted tree structure constrains the resulting ''graph''. A file is uniquely named by the *path* from the *root* to the file in the directory tree, called the *path name*. Name components are separated by the distinguished value ''/''. Referring to Figure 1, the path name of the file ''core'' is ''/u2/smith/core''. The *current directory* is a directory-valued variable. It is an implied prefix, used to abbreviate path names. Directories have pointers to their containing directory to ease traversal; the *root* contains itself.

This scheme is extensible across multiple media through *mount points*. A *mount point* is a named directory used to graft subtrees onto the tree. A mount point in a path name causes interpretation of the portion after the mount point name to occur within the context of the subtree rooted at that mount point. References to the containing directory in the root of the subtree must resolve to the mount point.

The UNIX kernel does not store the path name passed to system calls. For *open(2),*

® UNIX is a registered trademark of AT&T. DEC, Digital, RA81, and VAX are trademarks of Digital Equipment Corporation.

-- --

this absence prevents easy duplication of the MULTICS[4] *hcs_$fs_get_path_name()* call, which returned the segment name of the segment descriptor argument. For *chdir(2), pwd(1),* which finds a path from the root to ‘‘**.**’’*, is roughly equivalent to converting the descriptor to a name. For *mount(2),* an attempt to remedy the lack of pathnames is made by maintaining a file, */etc/mnttab* on System V, containing a mapping between mount point names and device names. Administrative commands such as *mount(1)* and *df(1)* use */etc/mnttab.*

The Problem

In principle, locating the *mount points* is not hard. They must be distinguished in some fashion. The difficulty is in determining the ‘‘path named’’ location of the mount points in a file system tree. The solution could be as simple as examining system data structures, either directly or with a system call, if the operating system stored the path name. The UNIX operating system kernel does not store path names associated with the mount points. A file containing the names of mount points and their associated disk devices is maintained so that administrative functions such as unmounting file systems can be performed, Administrative accidents or errant programs can incorrectly update the file. Such errors are infrequent on any given system, but large numbers of systems increase the frequency of such errors. Several approaches to corrupt or deleted files exist:

- Maintain redundant copies of the file. All programs writing to the file must adhere to this backup maintenance strategy.
- Maintain mount-point names in the kernel for recovery purposes. Path names are not long in practice; the total storage required for null-terminated path name strings on one large system with 38 mount points is 225 characters, less than a system buffer. Thus, storing these names should require a manageable amount of kernel storage.
- Restoring the file from alternate sources of information. For example, the set of commands invoked on system startup could be examined for *mount(1)* commands. This approach does not account for activity after startup. Another possibility is the use of some utility to examine the kernel-maintained data structures. This examination is difficult, and the data may be misleading. File system names stored with the volume may not be accurate, as when a backup copy of a mounted file system is examined.

* By convention, ‘‘**.**’’ is synonymous with the current directory, and ‘‘**..**’’ with its parent.

The first two are impractical, as they require either command or kernel changes. Such changes are discouraged; experience has shown that the changes necessary for new releases of the system software become unmanageable. Without these changes, the first two methods are either not attractive or not robust. *Setmnt(1)* can create a new */etc/mnttab* if auxiliary information is available.

Related Work

This work consists of both a result, *getmnt,* and a methodology. Since the algorithms are problem-specific, there is little prior work related to the result. Hence, relevant comparisons must be drawn based on the methodology used for performance analysis.

Profiling has been used[5-7] for an analysis of system events. The analysis was applied to improving system performance. These previous studies focus on operating system performance. Disk performance[8] has been studied using trace-gathered measurements of driver activity, but no performance improvements based on these measurements were reported. Similar techniques[9] were used to improve the performance of a commonly used network news management utility by a factor of 19; the focus was divided between the results and the methods used to achieve them. While the goals of the programs are unrelated, a similar methodology was used in the development of *getmnt,* that of profile-based performance analysis to refine a program. *Getmnt's* performance increases were achieved in two major jumps rather than in many small steps. The increases were achieved through algorithmic changes rather than implementation changes; many of the optimizations [9] had been applied in the construction of *getmnt1*.

The techniques used for *getmnt* may be useful in other hierarchical file system searches where mount points are significant, such as finding the machines in a network file system tree.

GETMNT1

The UNIX system call *stat(2)* returns, among other data, the device, identified by a number, that its file name argument resides on. Since the device on which a file is mounted changes for files beneath a mount point, they are found in a file tree traversal. On UNIX, device access is done with *special files,* which are entries in the file system name space kept by convention in the directory ‘‘/dev’’. These special files also contain a device datum in their inode entries, so that we can create a mapping between names and devices. This mapping is unfortunately not 1-to-1: ‘‘swap’’ typically shares a number.

Using these facts, *getmnt1* was written. *Getmnt1* builds a table of special file names, subject to certain naming constraints, indexed by the device id. It then recursively descends the directory tree starting from the root. Whenever the device id of a file differs from that of its "parent", the mount point name and the associated special file name are output. The recursive algorithm is:

```
get_mount_points( dir )

while( dir not empty )
{
   get next element;
   if( dev(dir) != dev(element) )
      mount_point( dir );
   if( is_dir(element) )
      get_mount_points( element );
}
```

The first draft in the development of *getmnt1* was slow, much like a ‘‘`find / -depth`’’ would be. The following optimizations were applied to yield acceptable performance:

— Relative path names and *chdir(2)* were used to cache *namei()* results. As pointed out in a UNIX system performance study[5] the *namei()* procedure used by *stat()* and *chdir()* is very expensive.

— *Read(2)* calls on directories were buffered in a 512 byte buffer.

— Traversal depth was bounded, e.g., to four directories deep.

Getmnt1 was used from late 1983 until the summer of 1987.

GETMNT2

Getmnt1's static and empirically-determined depth bound was unattractive. Worse, its approach did not scale well. The amount of disk storage associated with each processor had increased over time. For example, consider system **E,** a DEC™ 8650 processor. It has 38 file systems distributed over 12 RA81 drives, comprising 5.4 gigabytes of disk storage. *Getmnt1* took tens of minutes of real time to run on **E** and similarly configured systems. Since *getmnt* is used frequently as an administrative tool this performance was unacceptable. The analysis began with a profiled version of *getmnt1*. The *stat()* system call consumed about 47 percent of the execution time, with 29626 calls at 0.88 milliseconds per call; 14 percent by *read()* with 19460 calls at 0.41 msec/call; 14 percent by *chdir()* with 16108 calls at 0.46 msec/call; and 7 percent by *open()* with 8054 calls at

0.48 msec/call. The code was optimized to reduce the *cost* of system calls so that the problem clearly lay with the *number* being issued. It was not obvious that this number could be reduced*. Small changes made to the program resulted in equally small performance improvements.

— Directories had been read using an old block size of 512 bytes, and the program logic had used *lseek(2)* to skip past the ‘‘**.**’’ and ‘‘**..**’’ entries in a directory before reading began, thus misaligning the blocks read with respect to the blocks on disk. The buffer size was adjusted to the file system block size, halving the number of *read()* calls.

— The program read information from */dev* and its subdirectories one directory entry at a time. Buffering was applied, as before. Experimental data showed that the simplistic hashing scheme used in device name lookup was effective in generating short, well-distributed lists.

— *Get_mnt_pts(),* the file tree walking routine, and the source of most string manipulation calls, checked whether it had exceeded the depth limit specified. Checking before the routine was called eliminated many calls. This pre-checking exemplifies a rule for bushy trees: when you can, examine from the top.

These changes gave improvements of a few percent, not the desired order of magnitude. The next section describes our first method for achieving major performance improvements, ‘‘Leaf Pruning’’.

Leaf Pruning

The system calls in the profile results, e.g., *stat(), read(), chdir()* and *open(),* are used as part of a search process. The number of system calls is reduced if the search is more efficient. One way of making a search more efficient is a better criterion for stopping the search.

Extra information can give us a better stopping criterion. In the example file tree structure of Figure 1, mount points are marked with a parenthesized ‘‘device number’’. *Getmnt1* would traverse the illustrated tree to the depth bound of 4.

The extra information about the organization of the tree is obtained as follows:

— The system mount table structure is read from */dev/kmem,* a file system name-space entry for the kernel memory. A flag associated with each device is initially marked

* *Stat()* system calls issued during a file tree walk are used to gather information, e.g., directory status and device number. *Chdir()* is used in tree traversal. The *open(), read(),* and *close()* system calls are used to gather information from directories.

`UNREFERENCED`.

— The system mount table is examined, and the inode table entry is obtained for each mounted-on file system. This data forms a table of *<file system device #, mounted-on file system device #>* pairs.

Table 1. Device numbers with parents

device number	parent device
0	0
10	0
23	0
12	10
31	10
33	0

— The flags of devices in the right-hand column of the table are marked `INTERNAL`, as *mounted-on* file systems are *parents* in the tree structure. Devices present in the left-hand column but not in the right-hand column are marked `LEAF`. The special devices are thus partitioned into `UNREFERENCED`, `LEAF`, and `INTERNAL`. Thus, a membership test can determine the type of a given file system.

This information can be used to ‘‘prune’’ nodes from the search for mount points; no mount points can be found beneath a leaf node. For example, referring again to Figure 1, the search beneath */u1*, */usr/src*, */usr/spool/news*, and */u2* is unnecessary because they are leaf nodes. *Getmnt1* would search directories such as */u2/smith* and */u1/jms*. The algorithm for `get_mount_points( dir )` is:

```
while( dir not empty )
{
   get next element;
   if( dev(dir) != dev(element) )
      mount_point( element );
   if( is_dir(element) && !leaf(dev(element)) )
      get_mount_points( element );
}
```

Discussion

The extra information is used as a ‘‘hint’’, as an old */etc/mnttab* could be, thus it only improves performance; in the worst case the performance reverts to that of the old algorithm. However, a profile illustrates the performance can improve significantly. *Stat()* now consumes about 81 percent of the execution time, with 8100 calls at 0.95 msec/call; *chdir()* 4 percent, with 556 calls at 0.69 msec/call; *get_mnt_pts(),* a local function, about 3 percent, with 6749 calls at 0.04 msec/call; and *read()* about 3 percent, with 751 calls at 0.36 msec/call.

This technique is particularly effective where many leaf file systems are mounted at or near the root of the tree structure. The algorithm discovers the leaves almost immediately, and thus dispenses with their subtrees. The effectiveness of the technique is sensitive to the shape of the tree, but is remarkably effective in practice. Performance results were computed by gathering data for 17 systems. All command executions were timed with *timex(1),* which provides three statistics: *real* time, the amount of wall clock time used during the execution; *user* time, the amount of time the application program spent in control of the CPU; and *system* time, the amount of time the operating system spent in control of the CPU for the application. Timesharing of the system with other applications and waiting for I/O to complete account for *real-(user+system).* The data was gathered when the systems were in ‘‘single-user’’ mode to remove the effects of timesharing; the system buffer cache was pre-flushed. The ratio time(getmnt1)/time(getmnt2) is used to measure the improvement; this ratio allows comparison between unlike systems. Statistics* for these ratios are given for each relevant variety of time in Table 2. Column AVG1 contains the computed mean of the run times, in seconds, for *getmnt1*.

Table 2. Performance Summary, getmnt2/getmnt1

Time	mean	median	max	min	std_dev	AVG1
real	7.1	7.9	13.5	2.7	2.8	276.6
user	8.4	6.5	14.8	3.3	4.1	8.2
sys	3.2	2.3	7.0	1.3	1.8	60.5

GETMNT3

Mount points naturally tend to be located near the root of the tree. A depth-first strategy, even one with leaf-pruning, does not take advantage of this fact. It encouraged a strategy of exploring the top of the tree first, e.g., *breadth-first* search.

* Detailed data is available.[10]

Breadth-first search was implemented in early 1988:

```
List := "/";

while( not( all mount points found ) )
{
   get next directory, dir, from List;
   while( dir not empty )
   {
      get next element for which is_dir(element) == TRUE;

      if( dev(dir) != dev(element) )
         mount_point( element );
      if( leaf(dev(element)) )
         continue;
      append element to List;
   }
}
```

The change in search strategy was made in an attempt to reduce the number of *stat()* calls still further. It was successful, as profile results* demonstrate: *Stat()* now requires 88 percent of the execution time, with 1767 calls at 1.471 msec/call; *match(),* a local pattern-matching function, about 3 percent, with 7680 calls at 0.01 msec/call; and *malloc()* a library function for memory allocation, requires about 2 percent, with 1623 calls at 0.03 msec/call. This reduction in the number of calls should translate into a performance improvement. *Open(), chdir(),* and *read()* have become relatively minor costs. While fewer in number, each *stat()* call has become about 50 percent more costly, due to the greater effectiveness of the UNIX disk buffer cache in depth-first search. The table shows that the performance always improves and improves significantly on average. The magnitude of the improvement is not uniform; the variation is due to differences in tree shape. AVG2 is the computed mean of the times for *getmnt2;* from AVG2 the run times can be estimated using the ratios.

---

* */dev* was removed from the search, as it is traversed when device names, e.g., */dev/dsk/11s0,* are being mapped to device numbers. */dev* is also an unlikely place for file systems to be mounted. This (hack) removed about 600 *stat()* calls.

Table 3. Performance Summary, getmnt3/getmnt2

Time	mean	median	max	min	std_dev	AVG2
real	2.0	1.9	3.0	1.1	0.5	46.3
user	2.8	2.4	5.8	1.3	1.2	1.2
sys	2.4	2.3	4.0	1.3	0.6	29.2

Table 4 relates the performance of *getmnt3* to the performance of *getmnt1*.

Table 4. Performance Summary, getmnt3/getmnt1

Time	mean	median	max	min	std_dev
real	13.6	13.5	24.1	3.6	4.9
user	27.0	24.25	65.3	4.7	20.4
sys	7.4	8.3	15.5	2.7	4.2

To illustrate why the performance increases, consider the tree in Figure 1, and imagine that */usr/spool/news* is not a mount point. A directory *font* with many subdirectories is found previous to *src* in */usr*. Since *src* remains to be found, */usr* is not a leaf and the *font* directory must be searched to the depth bound. Such search can be costly. Breadth-first search postpones such traversals until necessary for correctness.

There were other improvements whose effect was not dramatic:

— *Path bunching*; in an attempt to reduce the cost of *chdir()* calls, the number of calls was traded against a slightly increased complexity for each call. Consider a directory search, shown in a *sh(1)* -like notation:

```
for i in a b c d
do
   cd ${i}
   # work
   cd ..
done
```

Equivalently:

```
cd a
#work
for i in b c d
do
   cd ../${i}
   #work
done
cd ..
```

Note that five calls to *chdir()* do the work of eight. A routine *cheap_cd(from_dir, to_dir)* applies several such heuristics to reduce the cost of directory changes.

— Directories are read with a single system call. A set of routines emulating the 4.2BSD [11] directory access routines was written. When accessed, the entire directory is read into a buffer. Subsequent calls for directory entries are satisfied from this buffer. While space utilization might be a problem in a recursive search, the strategy employed in *getmnt3* completes its examination of a given directory before beginning another. The space allocated to directory buffers is thus proportional to the single largest directory examined.

— The information gathered about the kernel table of mounted file systems was expanded to a complete tree structure. This organization has the advantage that changes in the tree structure can be noted as leaves are detected and removed. Termination of the search is detected by the root node's transition to a leaf, when all its children, and their children, and so on, have been detected.

However, the major gain was effected by the change in search strategy.

Costs

One potential problem was overuse of memory, but the memory utilization of *getmnt3* was roughly comparable, about a factor of 2 greater than *getmnt2.* The major cost of the changes is clearly in code complexity. Efficient breadth-first search required more complex data structures and procedures for traversing the UNIX file tree. The code complexity increased from *getmnt1* to *getmnt2* mainly as a result of the examination of kernel memory. While a single conditional test was added to the main loop of *getmnt1* for *getmnt2,* the cost of implementing the conditional was great.

The knowledge of many system details makes the code harder to understand and is an impediment to portability. The increase in source lines from *getmnt1,* 445, to *getmnt2,* 818, including comments, was about 80 percent; *getmnt3* showed a 60 percent increase, from 818 to 1317 lines. As a result of the changes, the code is less portable, and more effort is required to maintain it for new architectures and UNIX releases. Copies of previous versions are archived for reference by maintainers.

CONCLUSIONS

The performance increase is particularly satisfying on our development system, **E,** as the difference between *getmnt1* and *getmnt3* on this machine was a factor of 15, somewhat greater than the mean. The performance increases have been achieved by reducing the number of system calls. Under a timesharing workload, each system call causes the caller to be rescheduled, possibly resulting in a service delay. Thus, the observed improvements are more dramatic[10] when a system is timesharing.

In locating mount points the items of interest are near the top of the file hierarchy, and information about relative positions was available. We pruned nodes from our search and used breadth-first search to yield a significant performance improvement. The main ideas, of *using extra information* and *adapting the search strategy to the problem*, are applicable to many instances of file system search, and to algorithms in general.

In addition, the methodology may be useful to others; the results are embodied in the current *getmnt.*

ACKNOWLEDGMENTS

Henry Wong, Lorenzo Bonnani and John Ashmead instigated the development of *getmnt2.* Questions from John Ashmead and Gerald Maguire prompted *getmnt3.*

The presentation was improved by constructive criticism and suggestions from reviewers. Jonathan Gross made the presentation of performance data more meaningful. Charles Colbert, Vivian Hsu, Dave Slade and Jack Pucci helped to make new measurements for a revision. Some development and all testing was done at Bell Communications Research, Inc.

REFERENCES

1. D.M. Ritchie and K.L. Thompson, ‘‘The UNIX Operating System,’’ *Communications of the ACM* **17**, pp. 365-375 (July 1974).
2. K.L. Thompson, ‘‘UNIX Implementation,’’ *The Bell System Technical Journal* **57**(6, Part 2), pp. 1931-1946 (July-August 1978).
3. M. J. Bach, *The Design of the UNIX Operating System,* Prentice-Hall (1986).
4. Elliott I. Organick, *The Multics System,* Massachusetts Institute of Technology Press (1972).

5. Samuel J. Leffler, Michael J. Karels, and Marshall Kirk McKusick, ''Measuring and Improving the Performance of 4.2BSD,'' in *Proceedings, Summer 1984 USENIX Technical Conference*, Salt Lake City, Utah (June 12-15, 1984), pp. 237-252.

6. Marshall Kirk McKusick, Samuel J. Leffler, Michael J. Karels, and Luis Felipe Cabrera, ''Measuring and Improving the Performance of Berkeley UNIX,'' Technical Report, Computer Systems Research Group, University of California, Berkeley (November 30, 1985).

7. J. Ousterhout, H. Da Costa, D. Harrison, J. Kunze, M. Kupfer, and J. Thompson, ''A Trace-Driven Analysis of the UNIX 4.2 BSD File System,'' in *Proceedings of the Tenth ACM Symposium on Operating Systems Principles (ACM Operating Systems Review)*, Orcas Island, WA (December, 1985).

8. Thomas D. Johnson, Jonathan M. Smith, and Eric S. Wilson, ''Disk Response Time Measurements,'' in *Proceedings, Winter 1987 USENIX Technical Conference*, Washington, DC (January, 1987), pp. 147-162.

9. Geoff Collyer and Henry Spencer, ''News Need Not Be Slow,'' in *Proceedings, Winter 1987 USENIX Technical Conference*, Washington, DC (January, 1987), pp. 181-190.

10. Jonathan M. Smith, ''Performance Analysis and Improvement in UNIX File System Tree Traversal,'' Technical Report CUCS-323-88, Columbia University Computer Science Department (1988).

11. W. Joy, *4.2BSD System Manual*, 1982.

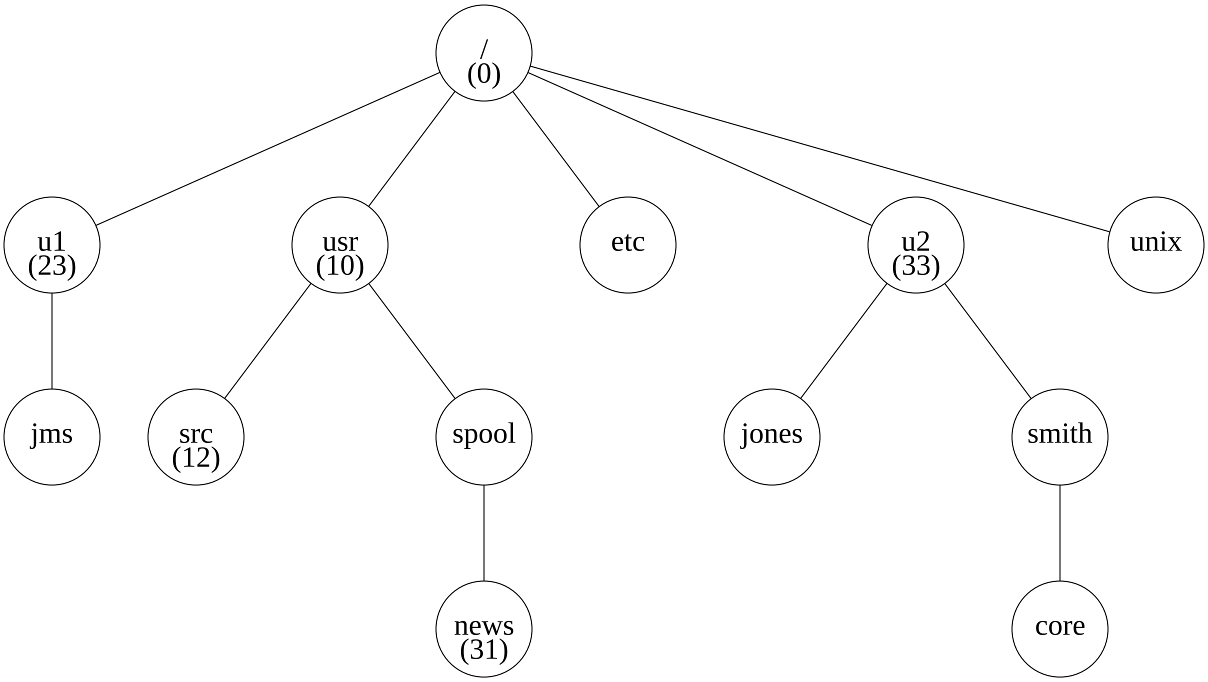

Figure 1. Sample File Tree