Performance Analysis and Improvement in UNIX File System Tree Traversal.

Jonathan M. Smith

Computer Science Department
Columbia University
New York, New York 10027
and
Bell Communications Research†
3 Corporate Place
Piscataway, New Jersey 08854

Technical Report CUCS-323-88

ABSTRACT

A utility program has been developed to aid UNIX system administrators in obtaining information about mounted file systems. The program gathers the information by a traversal of the accessible nodes in the file hierarchy; without kernel-recorded path names, this is the only way to dynamically determine mount-point names.

The program has had three significant versions, the second and third of which were driven by performance requirements rather than functional requirements. The original version showed a factor of 7 improvement over the performance of a naive tree traversal. The second iteration showed a factor of 10 improvement over the previous version by using extra information about the structure of the file system tree to prune unnecessary branches from the traversal. The third iteration showed another factor of 3 improvement by changing the search strategy. The performance improvements depend on an analysis described in this report.

Since the program's main task is traversal of a UNIX file system tree, our experience can be generalized to other such searches.

Performance Analysis and Improvement in UNIX File System Tree Traversal.

Jonathan M. Smith

Computer Science Department
Columbia University
New York, New York 10027
and
Bell Communications Research†
3 Corporate Place
Piscataway, New Jersey 08854

Technical Report CUCS-323-88

1. Introduction

The UNIX® kernel[8, 9] does not store the path name passed to system calls. In the instance of *open()*, this prevents one from easily duplicating the functionality of a useful MULTICS[6] call such as *hcs_$fs_get_path_name()*, which returned the segment name of the segment descriptor argument. In the instance of *chdir()*, a rough equivalence of inverting the name to descriptor conversion is achieved by */bin/pwd*. In the instance of *mount()*, an attempt to remedy the lack of pathnames is made by maintaining a file[1] containing a mapping between mount point names and device names. This is used by administrative commands such as *mount(1)* and *df(1)*.

In general, the path name inversion process is impossible without kernel changes[1] such as storing pathnames or allowing a directory change to be accomplished with *<device,i-number>* arguments rather than a name[2]. We have discovered that restoring the information in the file associated with the mounted file systems is tractable; this discovery was driven by occasional, but not infrequent, obliteration or corruption of this information. This report describes a set of three algorithms in historical order and the performance analysis which drove development after the first version. The average improvements[3] of the second version over the first, as measured by *timex(1)*, were: real time, 10.5 times faster; user time, 14.3 times faster; and system time, 5.2 times faster. The average improvements for the third version over those of the second version were: real time, 3.7 times faster; user time, 3.7 times faster; and system time, 3.1 times faster.

Thus, on average, a user can expect to see a 40-fold improvement in execution time over the original algorithm.

1.1. Related Work

McKusick, *et al.*[5] and Ousterhout, *et al.*[7] have measured performance by the method we present, that of profiling and using the profile to aid an analysis of the system events. Those studies both focus on the operating system, with McKusick's analysis being iterated for further performance improvements. For disk performance, Johnson, Smith, and Wilson[3] provide trace-gathered measurements of driver activity, but no performance improvements based on their measurements were reported. The most closely related work is that of Collyer and

---

† Jonathan M. Smith is a Ph.D. candidate at Columbia University, on educational leave of absence from Bell Communications Research, Inc. (Bellcore), where some development and all testing was done.

® UNIX is a registered trademark of AT&T.

1.) */etc/mnttab* on System V.

2.) This was done by Bellcore Lab 25832 as a method of detecting directory "snooping"; the algorithm took a process identifier, found that process's working directory, changed to it with the *<device,i-number>* call, and executed */bin/pwd*.

3.) These figures were obtained by summing the ratios of improvement for each system, as presented in Figures 7 and 10, and then dividing by the number of systems.

Spencer[2] , who improved the performance of a commonly used news management utility by a factor of 19. We use a similar methodology on our system administration utility, that of applying *prof(1)* to drive a performance analysis. Our performance increases were achieved in two major jumps rather than in many small steps, as the increases were achieved through a change of method rather than changes in the implementation of an existing method; many of Collyer and Spencer's optimizations had been performed on our initial program version. We share the same basic views on performance improvement: *care about performance* and *understand the problem.* Our results for this program can be used in other hierarchical file system searches, such as finding the machines in a network file system tree.

2. Initial Solution

2.1. The Problem

The essence of the problem is that the path names associated with the mount points are not stored by the kernel, and thus they are neither protected from writing nor accessible using a system call. As any file, the file containing the mount point names can be written to by processes with uid==0. This can happen by accident or by an errant program; while infrequent on any given system, if many systems are managed it can happen often. There are several possible solutions to the problem of a corrupt or obliterated file. These are:

- Maintain copies of the file in case one becomes damaged. This would require kernel support since programs cannot be forced to adhere to the backup-maintenance property without it.
- Maintain mount-point names in the kernel for recovery purposes. Since the path names are not long (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) they could easily be stored by the kernel.
- Restoring the file from other alternate sources of information. For example, the set of commands invoked on system startup could be examined to see which file systems are mounted, but this does not account for activity after startup. Another possibility is the use of some utility to examine the kernel-maintained mount table. This is difficult, and the file system names obtained from the superblock may be misleading, particularly if a backup copy of some mounted file system is mounted.

The first two are unacceptable, as they require kernel modifications. We've found that the number of kernel changes necessary for each new release becomes unmanageable, and we always miss one. Without such changes, the first two methods are not robust. *Setmnt(1m)* can be used to write a new copy of the file if we can gather the information for the third approach using some method other than those suggested.

2.2. Getmntl as a solution

A UNIX system call, **stat(2)**, returns, among other data, the device a file name argument resides on. Since the device on which a file is mounted changes for files beneath a mount point[4], we can determine mount points by doing a tree traversal. The block special files also contain a device datum in their inode entries, so that we can create a mapping between names and block devices (this is unfortunately not 1-to-1: "swap" is typically the same as some other block device).

Using these facts, a program called *getmntl* was written. The program builds a table of block 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", we output the mount point name and the associated block special file name. The algorithm is tersely described in Figure 1.

The first draft was abominally slow[5]; the following optimizations were applied to yield acceptable performance:

4.) Directory entries can also be scanned for i-number == 2; see Thompson.[9]

5.) Consider the roughly equivalent "`find / -depth`"; this required 5527.92 seconds of real time, 15.91 seconds of user time, and 266.46 seconds of system time as measured by *timex* on system S16 (names have been changed to protect the innocent), described further elsewhere.

```
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 );
}
```

Figure 1: Getmntl Algorithm

1.) Relative path names and *chdir()* were used to effect caching of *namei()* results. As pointed out by McKusick, *et al.*[5] the *namei()* procedure used by *stat()* and *chdir()* is very expensive.

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

3.) The depth of the traversal was bounded, e.g., to four directories deep.

This version served from late 1983 until the summer of 1987.

3. Performance: Problems, Analysis, Improvement

Unfortunately, even with optimizations, *getmntl's* approach requires a significant amount of time. It's unfortunate because the traversal is the only correct way to obtain the needed path name data. However, as our laboratory has acquired larger and larger processors, the attached disk resources have grown as well. For example, system S16, a DEC™ 8650 processor, has 38 file systems mounted on 12 RA81 drives, comprising 5.4 gigabytes of disk storage. As there is more work, it takes more time; *getmntl* took tens of minutes of real time to run on this and similarly configured systems. Since the command is used frequently as an administrative tool and as a part of other administration packages, this was unacceptable. An analysis was begun to see what performance improvements could be made; this began with a profiled version of the code, which yielded the results shown in Figure 2 when run on S16.

%Time	Seconds	Cumsecs	#Calls	msec/call	Name
47.4	26.02	26.02	29626	0.8782	_stat
14.4	7.92	33.93	19460	0.4068	_read
13.6	7.45	41.38	16108	0.4625	_chdir
7.0	3.83	45.22	8054	0.476	_open
5.1	2.82	48.03	117807	0.0239	_get_mnt_pts
3.3	1.82	49.85	236012	0.0077	_strcat
2.4	1.32	51.17	117806	0.0112	_strncpy
1.8	1.00	52.17	8055	0.124	_close
1.6	0.90	53.07	118223	0.0076	_strcpy
1.3	0.70	53.77	117806	0.0059	_strcmp

Figure 2: Top 10 lines of profile for getmntl

This data was informative: as discussed above, 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[6]. Some small changes were made to the program, which resulted in equally small increases in performance. These were:

1.) Directories had been read in 512 bytes at a time (this was an old block size) and the program logic had

* DEC, Digital, RA81, and VAX are trademarks of Digital Equipment Corporation.

6.) *Stat()* system calls issued in the course of 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.

used *lseek()* 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 directory reads were parameterized to reflect the file system block size, thus achieving a halving of the number of *read()* calls issued. "." and ".." were read in and explicitly skipped over rather than using *lseek()*.

2.) The program read information from */dev* and its subdirectories one directory at a time. This was changed so that the logic described in the previous item was used. The hashing scheme used in device name lookup was briefly examined at this point; experiments showed that it provides randomly distributed short lists of items to search, as it should.

3.) *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. This was changed so that the routine is never called when the depth limit would be exceeded. This exemplifies a rule for trees which broaden rapidly: when you can, examine from the top.

These changes gave improvements of a few percent, not the necessary order of magnitude. The remainder of this section describes a method for achieving performance improvements of (possibly) several orders of magnitude; one system has shown an almost 40 to 1 speedup.

The major improvement was due to derived data about characteristics of the file tree which can speed our traversal. The extra information only improves performance; in the worst case the performance reverts to that of the old algorithm. However, as is shown in the execution profile of Figure 3, taken from system **S16**, the performance improves significantly in some cases. In particular, we have reduced the number of expensive system calls.

%Time	Seconds	Cumsecs	#Calls	msec/call	Name
81.3	7.70	7.70	8100	0.951	_stat
4.0	0.38	8.08	556	0.69	_chdir
3.2	0.30	8.38	6749	0.044	_get_mnt_pts
2.8	0.27	8.65	751	0.36	_read
1.6	0.15	8.80	7169	0.021	_strcpy
1.4	0.13	8.93	8532	0.016	_match
1.2	0.12	9.05	280	0.42	_open
1.1	0.10	9.15			mcount
0.9	0.08	9.23	6750	0.012	_strcmp
0.9	0.08	9.32	6748	0.012	_strncpy

Figure 3: Top 10 lines of profile for 2nd getmnt

3.1. Leaf Pruning

The system calls are used as part of a search process; if we can make the search more efficient, the number of system calls can be reduced. One way of making a search more efficient is to generate a better criterion for stopping the search; this prevents searching nodes that are irrelevant.

We can use information about the structure of the tree of mounted file systems to give us a better stopping criterion. An example file tree structure is shown in Figure 4; mount points are marked with a parenthesized number. *Getmntl* would have traversed the illustrated tree to depth four.

The extra information is obtained in the following manner:

1.) The system mount table structure is read in from */dev/kmem*. A flag associated with each block device is initially marked UNREFERENCED.

2.) The system mount table is examined, as well as the inode table entry of the mounted-on file system for each entry. This allows us to create a table of *<file system device #, mounted-on file system device #>* pairs. For example, letting the parenthesized numbers in Figure 4 be device numbers, we get the table of Figure 5.

3.) The flag of any devices in the right-hand column of the table is marked INTERNAL. Devices present in the left-hand column but not in the right-hand column are marked LEAF. We have thus partitioned

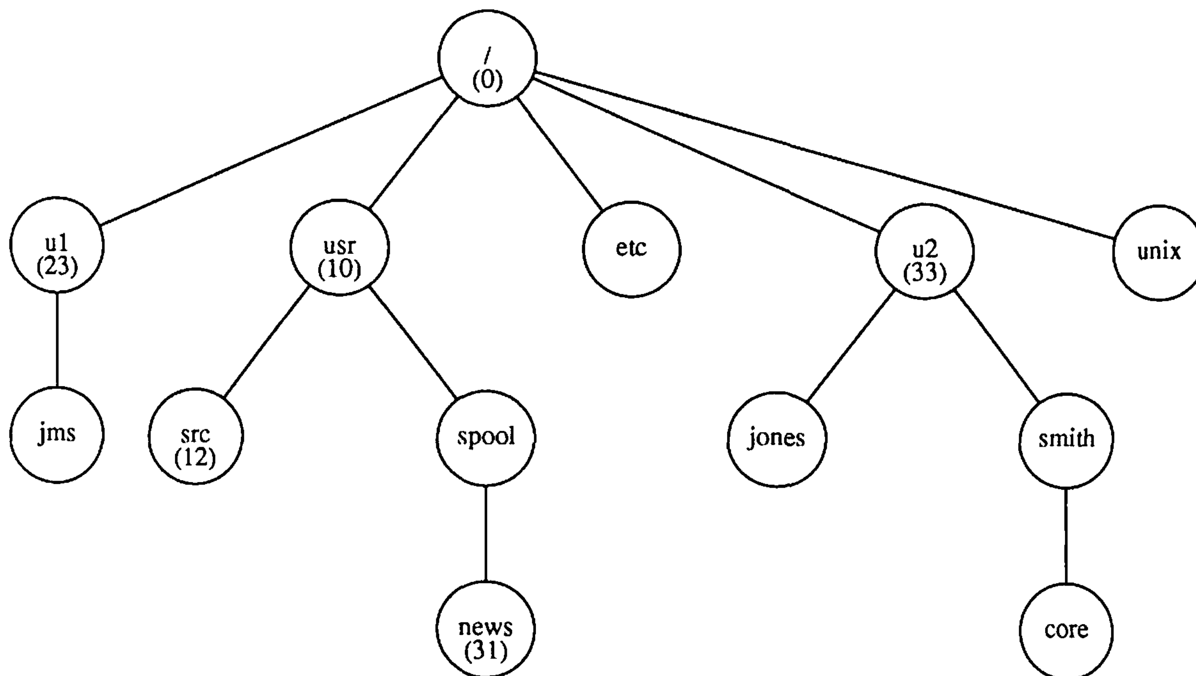

Figure 4: Sample File Tree

dev	mtd_on
0	0
10	0
23	0
12	10
31	10
33	0

Figure 5: Table of <F.S. dev #, mounted-on F.S. dev #> pairs

the block-special devices into three sets based on the information from the mount table: UNREFERENCED, LEAF, and INTERNAL. Thus, a membership test is sufficient to determine the type of a given file system.

This information can be used to optimize the tree traversal. If we are looking for mount points, we can stop a tree traversal as soon as the file system we are in is a leaf node; no other mount points will be found beneath this point in the tree. For example, referring again to Figure 4, we can avoid the search beneath */u1*, */usr/src*, */usr/spool/news*, and */u2* because they are leaf nodes. This is in contrast to *getmnt1*, which will search directories such as */u2/smith* and */u1/jms*. The algorithm is described tersely in Figure 6.

The technique of Figure 6 is particularly effective where many leaf file systems are mounted at or near the root of the tree structure, as the algorithm discovers them almost immediately, and has no further search to perform. While the effectiveness of the technique is determined to a great degree by the shape of the tree, it appears remarkably effective in practice. Figure 7 presents the results[7] of comparing the performance of the 1st algorithm with the

7.) Results were gathered by a broadcast remote command execution which executed the two commands simultaneously and mailed their timing statistics to a specified location. While this may introduce performance artifacts, e.g., directories being available in system buffers, system load is removed as a variable. The presentation of the data as performance ratios should eliminate or reduce the effect of such artifacts. The execution was done in mid-afternoon, when the systems have a "normal" load.

```
get_mount_points( dir )

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 );
}
```

Figure 6: Getmnt2 Algorithm

2nd one in a tabular form.

4. An Altered Search Strategy

In early 1988, breadth-first search was attempted; the algorithm is shown in Figure 8.

The change in search strategy was made in an attempt to reduce the number of *stat()* calls still further. It was quite successful, as profile results[8] in Figure 9 show.

This reduction in the number of calls (note that *open()* and *chdir()* have disappeared and that *read()* has become a minor cost; also note that while they are reduced in number, each *stat()* call has become about 50 percent more costly) should translate into a performance improvement; the table[9] in Figure 10 shows that while the magnitude of the improvement is not uniform (breadth-first search is affected by the tree shape as well) the performance always improves, and improves significantly on average.

To see why the performance increases, consider the tree in Figure 4, and imagine that */usr/spool/news* is not a mount point and that a directory *font* with many subdirectories is found previous to *src* in */usr.* Since */usr* is not a leaf (*src* remains to be found), the *font* directory must be searched to the depth bound, at a potentially large cost. Breadth-first search will postpone such "deep dives" until they are necessary for correctness.

There are other performance improvements in play here, although their effect is not dramatic. Some of the more interesting were:

1.) *Path bunching*; in an attempt to reduce the cost of *chdir()* calls, we traded the number of calls against a slightly increased complexity for each call. The idea is quite simple; the canonical directory search, shown in Figure 11, can be accomplished as shown in Figure 12.

    Note that we've made five calls to *chdir()* do the work of eight. A routine *cheap_cd(from, to)* attempts to optimize directory changes.

2.) Directories are read in with a single system call. A set of routines designed to emulate the 4.2BSD [4] directory access routines was written; the entire directory is read into an allocated buffer, and subsequent calls for directory entries are satisfied from this buffer. While space utilization might be a problem where a directory structure allocation routine is called by components in a recursive search, the strategy employed in *getmnt3* completes its examination of a given directory before beginning another; hence the space allocated to directory buffers is proportional to the single largest directory examined.

3.) The information gathered about the kernel table of mounted file systems was expanded to a complete tree rather than derived set membership data; this has the advantage that we can note changes in the tree

---

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

9.) The information was gathered as in the previous table. The systems available for measurement have changed over time; as can be discerned from the table components and the observed times for *getmnt2*.

System	getmnt1			getmnt2			getmnt1/getmnt2		
name	real	user	sys	real	user	sys	real	user	sys
S1	295.23	12.76	66.86	40.03	1.48	15.10	7.38	8.62	4.43
S2	191.73	10.88	49.86	86.48	1.03	14.58	2.22	10.56	3.42
S3	1284.78	10.21	68.83	32.80	0.31	6.31	39.17	32.94	10.91
S4	580.01	12.70	85.41	27.83	0.81	10.86	20.84	15.68	7.86
S5	200.78	6.85	32.23	21.48	0.70	8.60	9.35	9.79	3.75
S6	564.96	16.48	90.71	18.18	0.53	9.40	31.08	31.09	9.65
S7	467.20	15.48	66.18	31.13	0.65	9.68	15.01	23.82	6.84
S8	424.08	12.01	65.78	21.01	0.71	10.10	20.18	16.92	6.51
S9	407.78	10.25	55.61	50.61	0.86	8.83	8.06	11.92	6.30
S10	337.50	12.46	66.31	68.20	0.76	10.95	4.95	16.39	6.06
S11	435.81	12.83	59.96	29.33	0.53	8.78	14.86	24.21	6.83
S12	272.63	9.10	53.35	25.85	0.88	8.85	10.55	10.34	6.03
S13	288.00	14.60	66.95	33.21	1.28	15.70	8.67	11.41	4.26
S14	548.25	15.83	90.10	24.00	0.71	10.56	22.84	22.30	8.53
S15	194.33	13.46	69.43	25.15	1.08	13.98	7.73	12.46	4.97
S16	764.03	7.76	43.76	106.06	0.68	8.06	7.20	11.41	5.43
S17	272.75	5.41	18.20	15.78	0.21	3.41	17.28	25.76	5.34
S18	244.50	9.71	44.43	27.83	0.68	8.85	8.79	14.28	5.02
S19	59.71	3.56	17.60	15.20	0.70	8.55	3.93	5.09	2.06
S20	369.45	6.00	74.53	27.70	0.70	7.63	13.34	8.57	9.77
S21	338.31	7.75	43.15	173.21	1.98	22.96	1.95	3.91	1.88
S22	318.31	9.43	66.98	32.58	0.91	12.05	9.77	10.36	5.56
S23	156.10	4.15	22.68	25.78	0.61	8.93	6.06	6.80	2.54
S24	233.98	8.08	46.20	18.70	0.58	9.06	12.51	13.93	5.10
S25	141.81	8.40	45.81	25.80	0.85	11.76	5.50	9.88	3.90
S26	250.83	10.26	48.80	20.13	0.71	8.06	12.46	14.45	6.05
S27	664.38	18.33	127.05	36.11	0.71	11.33	18.40	25.82	11.21
S28	83.80	5.90	27.15	18.11	0.86	11.30	4.63	6.86	2.40
S29	131.45	6.76	31.90	19.35	0.60	9.26	6.79	11.27	3.44
S30	353.08	11.61	52.28	67.35	0.83	9.00	5.24	13.99	5.81
S31	172.35	11.95	59.18	49.61	2.21	31.56	3.47	5.41	1.88
S32	297.26	9.73	48.33	46.70	1.65	23.93	6.37	5.90	2.02
S33	186.41	7.35	47.03	40.53	1.76	24.68	4.60	4.18	1.91
S34	231.95	10.60	54.83	40.58	1.80	24.11	5.72	5.89	2.27
S35	322.06	15.76	94.00	60.86	2.15	35.26	5.29	7.33	2.67
S36	129.38	9.13	44.80	39.26	1.63	26.13	3.30	5.60	1.71
S37	499.51	11.81	68.58	57.16	0.91	12.70	8.74	12.98	5.40
S38	191.56	7.91	36.65	48.83	0.63	11.85	3.92	12.56	3.09
S39	269.80	7.35	45.51	137.60	1.95	26.55	1.96	3.77	1.71
S40	336.33	15.13	63.23	22.41	0.78	9.21	15.01	19.40	6.87

Figure 7: Timex(1) derived performance data, 1st vs. 2nd (All times in seconds)

structure as we detect and remove leaves; in particular 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.

4.1. At what cost?

The major cost of the changes is clearly in code complexity. Breadth-first search had seemed unnatural at first, and we feared overuse of memory[10], but this was not a problem. The code complexity increased from *getmnt1*

10.) In fact, we suspect that the memory utilization of *getmnt3* is comparable to that of *getmnt2*. On S16, a *size(1)* of *getmnt2* shows `13812+1624+2780=18216` and *getmnt3* shows `15292+3748+2768=21808`. Utilization of dynamic memory, measured by

```
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;
   }
}
```

Figure 8: Getmnt3 Algorithm

%Time	Seconds	Cumsecs	#Calls	msec/call	Name
88.1	2.60	2.60	1767	1.471	_stat
3.4	0.10	2.70	7680	0.013	_match
1.7	0.05	2.75	1623	0.031	_malloc
1.7	0.05	2.80	1	50.	_do_devs
1.1	0.03	2.83			mcount
1.1	0.03	2.87	39	0.9	_write
0.6	0.02	2.88	236	0.07	_append_List
0.6	0.02	2.90	129	0.13	_read
0.6	0.02	2.92	560	0.03	_insert
0.6	0.02	2.93	77	0.2	__doprnt
0.6	0.02	2.95	114	0.15	_lseek

Figure 9: Top 10 lines of profile for getmnt3

to *getmnt2* mainly as a result of the examination of kernel memory; the amount of source code for */bin/ps* is representative of the program complexity necessary when examining the kernel.

As well as making the code harder to understand, the knowledge of so many system details is an impediment to portability. The increase in source lines, including comments, was on the order of 80 percent (from 445 to 818) from *getmnt1* to *getmnt2; getmnt3* showed a 60 percent (from 818 to 1317) increase. Since, as a result of the changes, the code is less portable, more effort will be required to maintain it for each new architecture or UNIX release. In any case, we have archived copies of the previous versions so that the design history can be followed by maintainers, and we can backtrack from ill-advised designs.

5. Summary

We are particularly pleased with the performance increase shown on our home system, S16 (an aggregate improvement factor of 72!). The general problem of inverting a name to i-number (or descriptor) mapping is hard without extra kernel support. We have looked at a specific subset of this problem which is tractable due to the problem characteristics. In particular, we know that the items of interest are near the top of the file hierarchy. This

---

comparing the value of *sbrk(0)* at the start of execution to its value at the end of execution shows that *getmnt2* uses 23552 bytes while *getmnt3* uses 38912 bytes. It is somewhat tricky to compare stack segment utilization, but the deeper recursion involved in *getmnt2's* algorithm suggests that more of the segment is used. Of course, *getmnt3's* utilization will climb dramatically with greater depth, as more information about ancestors must be stored.

System name	getmnt2			getmnt3			getmnt2/getmnt3		
	real	user	sys	real	user	sys	real	user	sys
S1	10.96	0.98	9.91	4.86	0.31	4.40	2.26	3.16	2.25
S2	26.11	0.96	10.23	5.55	0.23	4.06	4.70	4.17	2.52
S3	33.61	0.60	11.11	11.28	0.28	4.60	2.98	2.14	2.42
S4	18.08	0.68	9.10	2.93	0.13	1.21	6.17	5.23	7.52
S8	86.70	0.66	10.73	22.20	0.25	4.26	3.91	2.64	2.52
S9	8.33	0.68	7.63	3.68	0.30	3.38	2.26	2.27	2.26
S10	15.45	0.85	11.01	9.68	0.20	4.35	1.60	4.25	2.53
S11	9.51	0.51	8.91	6.86	0.25	3.26	1.39	2.04	2.73
S12	12.25	0.76	9.03	4.56	0.25	3.68	2.69	3.04	2.45
S13	16.68	1.01	10.20	5.00	0.23	4.18	3.34	4.39	2.44
S41	21.93	0.36	9.21	4.45	0.21	3.33	4.93	1.71	2.77
S14	13.31	0.83	11.18	4.33	0.28	3.76	3.07	2.96	2.97
S15	13.48	0.93	8.93	7.73	0.26	3.91	1.74	3.58	2.28
S16	71.08	1.13	11.86	7.11	0.26	2.66	10.00	4.35	4.46
S17	4.61	0.31	4.21	2.16	0.18	1.90	2.13	1.72	2.22
S18	18.11	0.53	8.68	5.88	0.33	3.65	3.08	1.61	2.38
S19	9.31	0.50	8.80	3.93	0.11	3.78	2.37	4.55	2.33
S20	8.25	0.63	7.48	1.26	0.13	1.11	6.55	4.85	6.74
S21	190.18	2.03	23.35	161.61	1.71	16.60	1.18	1.19	1.41
S22	70.26	1.03	12.00	4.48	0.10	1.78	15.68	10.30	6.74
S23	10.41	0.80	8.23	1.48	0.06	1.38	7.03	13.33	5.96
S24	8.76	0.73	8.00	6.23	0.25	3.61	1.41	2.92	2.22
S25	9.71	0.61	7.96	1.35	0.13	1.20	7.19	4.69	6.63
S26	43.08	0.60	8.45	15.73	0.31	3.71	2.74	1.94	2.28
S27	13.48	0.76	10.01	4.95	0.35	3.95	2.72	2.17	2.53
S29	32.06	0.86	8.80	10.71	0.21	3.86	2.99	4.10	2.28
S30	27.55	0.81	8.85	6.81	0.18	4.26	4.05	4.50	2.08
S31	24.85	1.56	21.65	14.33	0.50	9.98	1.73	3.12	2.17
S32	26.01	1.43	23.43	12.08	0.83	10.11	2.15	1.72	2.32
S33	30.25	1.71	24.98	11.48	0.85	10.35	2.64	2.01	2.41
S34	29.70	1.53	24.35	11.80	0.58	9.96	2.52	2.64	2.44
S35	268.40	2.03	34.90	114.31	1.01	15.26	2.35	2.01	2.29
S36	30.00	1.86	26.61	13.05	0.71	11.70	2.30	2.62	2.27
S37	17.16	0.96	10.83	7.73	0.31	4.96	2.22	3.10	2.18
S38	68.65	2.25	18.95	11.38	0.26	4.05	6.03	8.65	4.68
S39	38.60	1.83	26.53	9.28	0.65	8.45	4.16	2.82	3.14
S40	20.93	0.55	8.98	14.51	0.21	4.18	1.44	2.62	2.15

Figure 10: Timex(1) derived performance data, 2nd vs. 3rd (All times in seconds)

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

Figure 11: Searching through a directory

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

Figure 12: Alternate Search through a directory

allowed us to employ an algorithm to prune nodes from our search; this algorithm can be combined with a breadth-first search strategy to yield a significant performance improvement for this problem. 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.

6. Acknowledgements

Henry Wong, Lorenzo Bonnani and John Ashmead instigated the first round of performance analysis and improvements. Discussions with John Ashmead and Gerald Maguire prompted the second round, resulting in the third and possibly final version of the program described here.

7. References

[1] Robert Penn Cagle, ''Process Suspension and Resumption in the UNIX System V Operating System,'' University of Illinois Computer Science Department, UIUCDCS-R-86-1240 (January 1986).

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

[3] 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.

[4] W. Joy, *4.2BSD System Manual*, 1982.

[5] 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).

[6] Elliott I. Organick, *The Multics System*, Massachusetts Institute of Technology Press (1972).

[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] D.M. Ritchie and K.L. Thompson, ''The UNIX Operating System,'' *Communications of the ACM* **17**, pp. 365-375 (July 1974).

[9] K.L. Thompson, ''UNIX Implementation,'' *The Bell System Technical Journal* **57**(6, Part 2), pp. 1931-1946 (July-August 1978).