

This article aims to show that the creation of several thousand files in the same directory can cause major latency issues.

The tests were carried out using an Ultra 20 (x64-based) Solaris 10 system platform, using 400000 file creation operations.

Timings were also monitored when 10000 files were created in each of 40 directories (*resulting in the same number of file creations*). When multiple directories were used, two tests were performed. The first test created 10000 files on a per directory basis. The second test created a file across all 40 directories, thereby striping across the directories, and repeated this process 10000 times.

The results of all these tests were then compared to see which operation provided the greatest latency.

This article is copyrighted to Context-Switch Limited, Egham, Surrey, UK.

This PDF file can be located at the following URL:

<http://www.middleworld.com/docs/latency1.pdf>



The Premise

Programmers may sometimes choose to write data to many smaller files, in a single directory, rather than one large file. When testing their program design, the number of files generated during the test phase may be quite small (*up to 1000s of files*).

But what happens when the program goes live and the volume of data being handled causes the number of generated files to be in the 100,000+ range.

This exercise aims to show that the creation of hundreds of thousands of files, especially in the same directory, can cause severe latency problems, resulting in poor performance of the software.

The *dtrace* program is used to monitor the time take to create each file.

The article also contains a description of the *dtrace* program that was used and explains why certain programming constructs were applied.

Creating a File

When a new file is created in a directory, a number of events must take place:

1. The file name must be verified as being unique
2. The inode number must be assigned to the file name
3. The inode number, file name length and the file name must be added to the contents of the directory file
4. The inode details of the file need to be generated and, if the file has a content greater than zero, the data content must be written to disk in the allocated disk blocks (*probably being cached in RAM before the write to disk takes place*).

To see the actions that occur when a file is created, we used the `truss` command to observe the system calls that were made. The output is shown below:

```
ksh# truss -f -o /tmp/truss.out touch newfile
ksh#
ksh# cat /tmp/truss.out
29765: execve("/usr/bin/touch", 0xFFBFFD3C, 0xFFBFFD48)  argc = 2
29765: resolvepath("/usr/lib/ld.so.1", "/lib/ld.so.1", 1023) = 12
29765: resolvepath("/usr/bin/touch", "/usr/bin/touch", 1023) = 14
...
29765: munmap(0xFF254000, 57344) = 0
29765: memcntl(0xFF250000, 6576, MC_ADVISE, MADV_WILLNEED, 0, 0) = 0
29765: close(3) = 0
29765: munmap(0xFF270000, 8192) = 0
29765: stat64("newfile", 0xFFBFFB98) Err#2 ENOENT
29765: creat64("newfile", 0666) = 3
29765: close(3) = 0
29765: stat64("newfile", 0xFFBFFB98) = 0
29765: utimes("newfile", 0x00000000) = 0
29765: _exit(0)
ksh#
```

We see that the file is created using a derivative of the `creat` system call:

```
ksh# man creat
...
creat - create a new file or rewrite an existing one
...
```

In this example, the `creat64` system call derivative of `creat` is being used.

If we wish to trace the creation details for files using the `creat` system calls, we need to determine which `dtrace` function name to use:

```
ksh# dtrace -l | grep 'syscall.*creat'
 22  syscall                creat entry
 23  syscall                creat return
276  syscall                lwp_create entry
277  syscall                lwp_create return
334  syscall                timer_create entry
335  syscall                timer_create return
394  syscall                creat64 entry
395  syscall                creat64 return
ksh#
```

We can see, in the list shown above, that there are two possible `creat` system calls that could be made: `creat` and `creat64`. Each has an entry and a return point.

The entry point would be when the system call is first made.

The return point is when the system call has been completed. When the return point is reached, we can assume that the file has been created and added to the directory contents.

File Creation

Multiple-files in a Single-directory

We wish to check the time taken to create a file when there are already thousands of files in the same directory. To create several thousand files, we will make use of the following shell script:

```
ksh# more createfiles1
#!/bin/ksh
# Korn shell script

# A script used to create multiple files in the same directory

integer a=1

while (( a <= 400000 ))
do
    touch file${a}           # create a file with a unique name
    (( a = a + 1 ))         # increment the counter
done
```

It is worth noting that each file being created will only increase the size of the directory file, itself, as no data is being stored in each file.

All of the file creation took place in a newly-created directory.

Multiple-files in Multiple-Directories

To compare the time taken to create the same number of files but stored in a number of directories, we used the following script:

```
ksh# more createfiles2
#!/bin/ksh
# Korn shell script
# A script used to create 10000 files in 40 sub-directories

if [[ -d testdtrace ]]
then
    rmdir testdtrace || exit 2
fi

mkdir testdtrace && cd testdtrace || exit 2

integer numarg=1
while (( numarg <= 40 ))
do
    mkdir subdir${numarg}
    (( numarg = numarg + 1 ))
done

# now start the main shell script program
integer a=1

# in each sub-directory, create 10000 files
for dirlist in subdir*
do
    while (( a <= 10000 ))
    do
        touch ${dirlist}/file${a} # create file in sub-directory
        (( a = a + 1 ))           # increment the counter
    done

    integer a=1
done
```

As before, the test directory and multiple sub-directories were newly-created for the purpose of the test.

Files striped across Multiple-Directories

To compare the time taken to create the same number of files but across a number of directories, we used the following script:

```
ksh# more createfiles3
#!/bin/ksh
# Korn shell script
# A script used to create 10000 files striped across 40 sub-directories

if [[ -d testdtrace ]]
then
    rmdir testdtrace || exit 2
fi

mkdir testdtrace && cd testdtrace || exit 2

integer numarg=1
while (( numarg <= 40 ))
do
    mkdir subdir${numarg}
    (( numarg = numarg + 1 ))
done

# now start the main shell script program
integer a=1
dirnames=subdir*

# in each sub-directory, create 10000 files
while (( a <= 10000 ))
do
    for dirlist in subdir*
    do
        touch ${dirlist}/file${a} # create file in sub-directory
    done

    (( a = a + 1 )) # increment the counter
done
```

Again, the test directory and multiple sub-directories were newly-created for the purpose of the test.

The *dtrace* program

The creation of the files will be performed using the `touch` command. We can, therefore, use the command name `touch` as our predicate value in the following *dtrace* program:

```
ksh# cat manyfiles.d
#!/usr/sbin/dtrace -s

#pragma D option quiet

int COUNTER;

syscall::creat*:entry
/execname == "touch"/
{
    self->etcheck = timestamp;
    COUNTER = COUNTER + 1;
}

syscall::creat*:return
{
    this->rtcheck = timestamp - self->etcheck;
    this->msecs = ( this->rtcheck / 1000 );

    printf("%9d : %10d msecs\n", COUNTER, this->msecs);

    self->etcheck = 0;
}
```

The *dtrace* program is described on the following page.

A description of the *dtrace* program

The *dtrace* program is called with the `-qs` options. The `-q` option causes the *dtrace* program to work in quiet mode. The `-s` option is a required option as *dtrace* is reading its program from a script file.

First, an integer variable is declared:

```
int COUNTER; /* declare an integer variable */
```

Next, a *dtrace* probe is specified with a predicate. The probe is the `creat` system call (*in either of its forms*) when the system call is first made (entry). The predicate ensures that the probe is only fired when the executable making the system call is the `touch` command:

```
syscall::creat*:entry  
/execname == "touch"/
```

The actions, associated with the firing of the probe are contained within the opening and closing curled-braces. The actions are as follows:

- a. Record the current time details. The timestamp variable is an internal variable that contains the current nanosecond timestamp counter. By saving this value in the newly-declared `self->etcheck` variable, we are able to subsequently compute the time difference between the entry and return times.

```
self->etcheck = timestamp;
```

- b. Increment the value stored in the `COUNTER` variable. This variable will be used to verify that 400000 files have been created using the `touch` command.

```
COUNTER = COUNTER + 1;
```

Note – Using a `self->` variable type, the *dtrace* code in the kernel will cause this variable content to be stored in an internal buffer. The variable contents can thus be used by another clause in the *dtrace* program.

Then, we specify another `dtrace` probe for the `creat` system call return point:

```
syscall::creat*:return
```

The actions, associated with the firing of the probe are again contained within the opening and closing curled-braces. The actions are as follows:

- a. Record the time delay. We save the value of the different between the current timestamp and the `self->etcheck` contents in the newly-declared `this->rtcheck` variable. This now provides us with the difference, in nanoseconds, between the entry and return of the system call.

```
this->rtcheck = timestamp - self->etcheck;
```

- b. We divide the value stored in `this->rtcheck` by 1000 to convert the nanosecond value into a millisecond value. This value is stored in the `this->msecs` variable.

```
this->msecs = ( this->rtcheck / 1000 );
```

- c. Formatted output is printed, showing the current value of `COUNTER` and the millisecond time for that, particular, file creation.

```
printf("%9d : %10d msecs\n", COUNTER, this->msecs);
```

- d. The `self->etcheck` variable is set to a value of zero (0). This causes this particular variable to be dropped from the internal buffer.

```
self->etcheck = 0;
```

Note – Using the `this->` variable type causes the `dtrace` kernel code to drop the variable once the program clause is completed. The variable, therefore, is not stored in the internal buffer. This helps to prevent buffer overflows.

Checking the Latency

The *dtrace* program was started approximately 10 seconds before the appropriate script was executed. This was done to allow *dtrace* to create the internal buffer structures and activate the appropriate probes.

Output from the *dtrace* program was filtered through the *sed* utility, so that output was produced each time another 100 files had been created:

```
ksh# ./manyfiles.d | sed '/000 :/w /var/tmp/test.log'
```

The output from the three tests is shown (*in an abridged form*) in the table below:

File Number	Single Directory	Multiple Directories	Striped Directories
1000 :	39 msec	44 msec	57 msec
2000 :	33 msec	38 msec	34 msec
3000 :	39 msec	46 msec	35 msec
4000 :	38 msec	39 msec	33 msec
5000 :	38 msec	52 msec	38 msec
...			
51000 :	57 msec	48 msec	42 msec
52000 :	44 msec	42 msec	42 msec
53000 :	61 msec	51 msec	48 msec
54000 :	73 msec	44 msec	49 msec
55000 :	60 msec	47 msec	41 msec
...			
151000 :	101 msec	40 msec	56 msec
152000 :	97 msec	46 msec	60 msec
153000 :	105 msec	46 msec	51 msec
154000 :	104 msec	38 msec	48 msec
155000 :	99 msec	50 msec	55 msec
...			
251000 :	228 msec	46 msec	73 msec
252000 :	234 msec	41 msec	74 msec
253000 :	238 msec	49 msec	75 msec
254000 :	242 msec	43 msec	75 msec
255000 :	233 msec	43 msec	66 msec
...			
400000 :	397 msec	55 msec	86 msec

Latency Times

From these results we can see the following:

1. When all of the files were created in a single directory, the latency of creation time increases as more file names are added to the directory.

The increase in latency is probably related to having such a large number of file names to search through.

2. When fewer files are created in the same directory, the latency can be kept relatively low.

Searching through the directory for possible matching names can be achieved more quickly because of the reduced number of files within that directory.

3. Creating files in a striped manner, where one file was created in each of the sub-directories and this process repeated 10000 times, was not as good for latency as when 10000 files were created in one operation for each directory.

Having to switch the search for matching file names from one directory to another in rapid succession appears to add to the latency time.

Conclusion

If software is going to be designed to write hundreds of thousands of files, it would be better to create thousands (*or tens of thousands*) of files in one directory, then switch the target directory to another directory, repeating the process until all the required files have been created.

This test was carried out using the Solaris UNIX™ File System (UFS). As other file system types can exhibit different characteristics, it would be worth trying the tests, again, on an alternative file system to see whether or not the results prove to be similar.

Also, as no data was saved in any of the created files, the latency issue may have been affected.

We have proved, however, that there can be significant differences in the performance of software that are caused by the design approach of the software developer.

It may even have been better to write all the data to a single file for subsequent analysis. *We shall leave that for another article.*

Note – This is the first in a number of planned articles exploring the application of the *dtrace* utility to Solaris performance issues.

About the author

This article was written by Jeff Turner of Context-Switch Limited.

Jeff has been involved in IT training for more than 20 years and has written a number of articles for a variety of UNIX-related magazines.

He has also designed and developed training courses for Sun Microsystems and other major IT companies during his career.

For the past 16 years, Jeff has been running his own training and consultancy company, providing training design and delivery services to companies around the world. Jeff has a particular interest in Solaris performance.

Jeff lives in the UK with his wife, two cats and a number of UNIX/Linux computers.