5

File Systems

File systems—an integral part of any operating system—have long been one of the most difficult components to observe when analyzing performance. This is largely because of the way file system data and metadata caching are implemented in the kernel but also because, until now, we simply haven't had tools that can look into these kernel subsystems. Instead, we've analyzed slow I/O at the disk storage layer with tools such as `iostat(1)`, even though this is many layers away from application latency. DTrace can be used to observe exactly how the file system responds to applications, how effective file system tuning is, and the internal operation of file system components. You can use it to answer questions such as the following.

- What files are being accessed, and how? By what or whom? Bytes, I/O counts?
- What is the source of file system latency? Is it disks, the code path, locks?
- How effective is prefetch/read-ahead? Should this be tuned?

As an example, `rwsnoop` is a DTrace-based tool, shipping with Mac OS X and OpenSolaris, that you can use to trace read and write system calls, along with the filename for file system I/O. The following shows `sshd` (the SSH daemon) accepting a login on Solaris:

```
# rwsnoop
  UID    PID CMD          D   BYTES FILE
    0 942611 sshd         R      70 <unknown>
    0 942611 sshd         R       0 <unknown>
```

continues

```
      0 942611 sshd         R    1444 /etc/gss/mech
      0 942611 sshd         R       0 /etc/gss/mech
      0 942611 sshd         R       0 /etc/krb5/krb5.conf
      0 942611 sshd         R    1894 /etc/crypto/pkcs11.conf
      0 942611 sshd         R       0 /etc/crypto/pkcs11.conf
      0 942611 sshd         R     336 /proc/942611/psinfo
      0 942611 sshd         R     553 /etc/nsswitch.conf
      0 942611 sshd         R       0 /etc/nsswitch.conf
      0 942611 sshd         R     916 /var/ak/etc/passwd
      0 942611 sshd         R       4 /.sunw/pkcs11_softtoken/objstore_info
      0 942611 sshd         R      16 /.sunw/pkcs11_softtoken/objstore_info
      0 942611 sshd         W      12 /devices/pseudo/random@0:urandom
      0 942611 sshd         R       0 /etc/krb5/krb5.conf
      0 942611 sshd         W      12 /devices/pseudo/random@0:urandom
      0 942611 sshd         R       0 /etc/krb5/krb5.conf
      0 942611 sshd         W      12 /devices/pseudo/random@0:urandom
      0 942611 sshd         R       0 /etc/krb5/krb5.conf
      0 942611 sshd         W      12 /devices/pseudo/random@0:urandom
      0 942611 sshd         W     520 <unknown>
  [...]
```

Unlike iosnoop from Chapter 4, Disk I/O, the reads and writes shown previously may be served entirely from the file system in-memory cache, with no need for any corresponding physical disk I/O.

Since rwsnoop traces syscalls, it also catches reads and writes to non–file system targets, such as sockets for network I/O (the <unknown> filenames). Or DTrace can be used to drill down into the file system and catch only file system I/O, as shown in the "Scripts" section.

Capabilities

The file system functional diagram shown in Figure 5-1 represents the flow from user applications, through the major kernel subsystems, down to the storage subsystem. The path of a data or metadata disk operation may fall into any of the following:

1. Raw I/O (/dev/rdsk)
2. File system I/O
3. File system ops (mount/umount)
4. File system direct I/O (cache bypass)
5. File system I/O
6. Cache hits (reads)/writeback (writes)
7. Cache misses (reads)/writethrough (writes)
8. Physical disk I/O

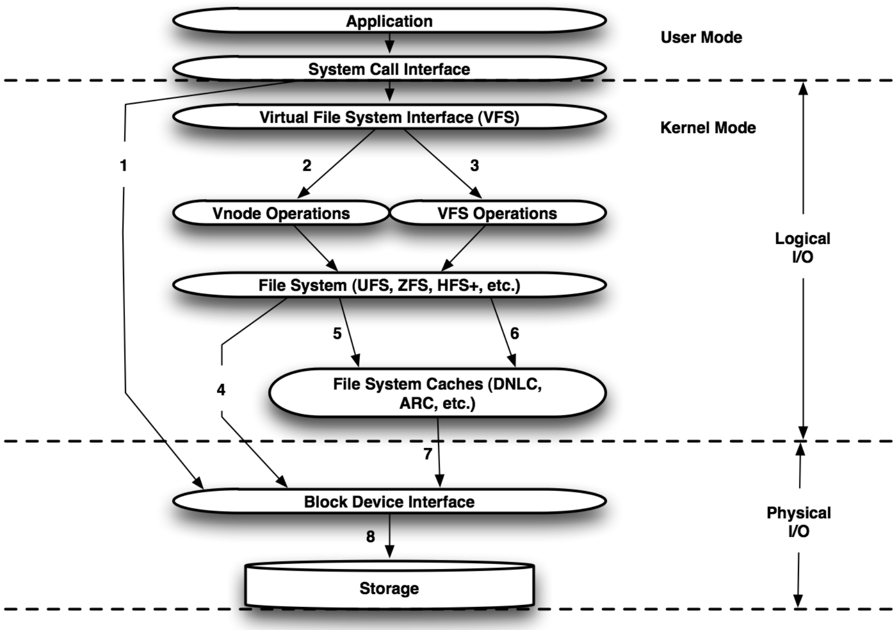

Figure 5-1 File system functional diagram

Figure 5-2 shows the logical flow of a file system read request processing through to completion. At each of the numbered items, we can use DTrace to answer questions, such as the following.

1. What are the requests? Type? Count? Read size? File offset?
2. What errors occurred? Why? For who/what?
3. How many reads were from prefetch/read ahead? (ZFS location shown.)
4. What was the cache hit rate? Per file system?
5. What is the latency of read, cache hit (request processing)?
6. What is the full request processing time (cache lookup + storage lookup)?
7. What is the volume of disk I/O? (How does it compare to 1?)
8. What is the disk I/O latency?
9. Did any disk errors occur?
10. Latency of I/O, cache miss?
11. Error latency? (May include disk retries.)

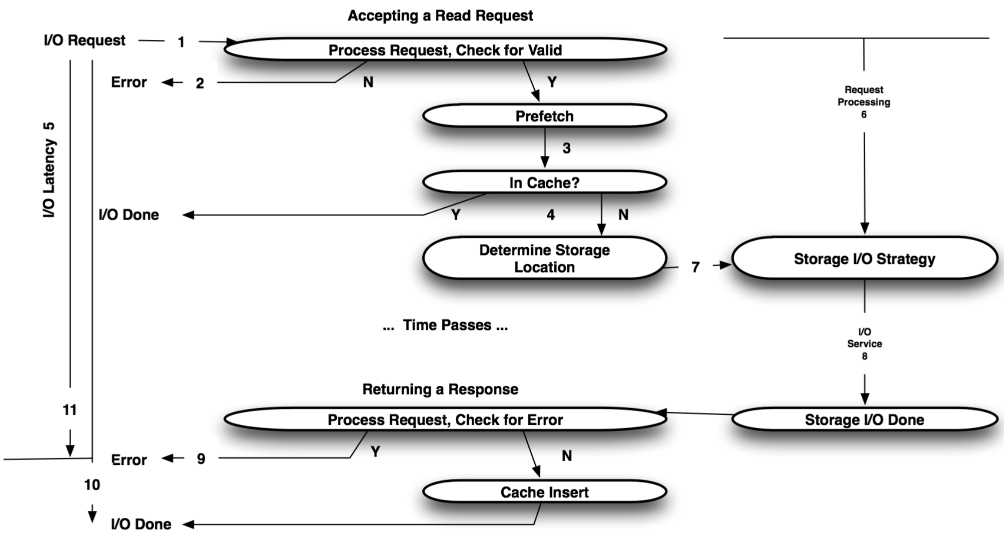

Figure 5-2 File system read operation

Figure 5-3 shows the logical flow of a file system write request processing through to completion. At each of the numbered items, we can use DTrace to answer questions, such as the following.

1. What are the requests? Type? Count? Write size? File offset?
2. What errors occurred? Why? For who/what?
3. How much of the write I/O was synchronous?
4. What is the latency of write, writeback (request processing)?
5. What is the full request processing time (cache insertion + storage lookup)?
6. What is the volume of disk I/O? (How does it compare to 1?)
7. What is the disk I/O latency for normal writes?
8. What is the disk I/O latency for synchronous writes (includes disk cache sync)?
9. Did any disk errors occur?
10. What is the latency of an I/O on a cache miss?
11. What is the error latency? (This may include disk retries.)

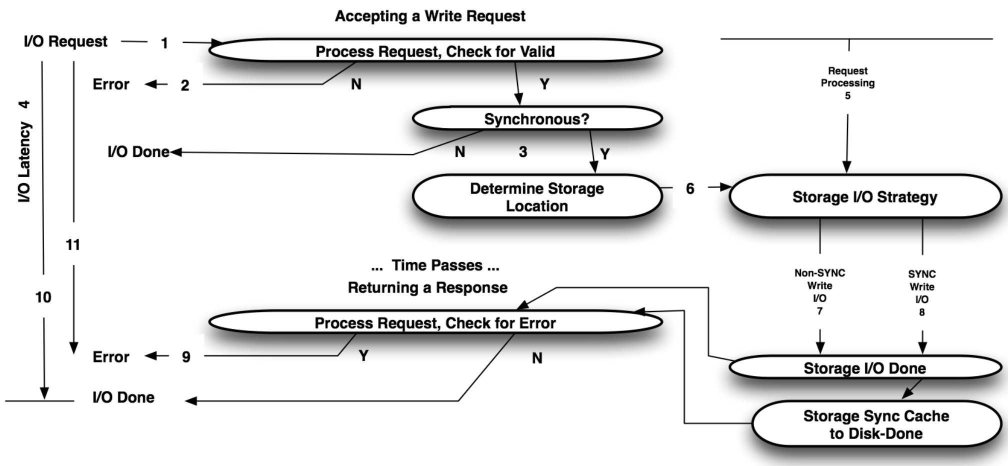

Figure 5-3 File system write operation

Logical vs. Physical I/O

Figure 5-1 labels I/O at the system call layer as "logical" and I/O at the disk layer as "physical." Logical I/O describes all requests to the file system, including those that return immediately from memory. Physical I/O consists of requests by the file system to its storage devices.

There are many reasons why the rate and volume of logical I/O may not match physical I/O, some of which may already be obvious from Figure 5-1. These include caching, read-ahead/prefetch, file system record size inflation, device sector size fragmentation, write cancellation, and asynchronous I/O. Each of these are described in the "Scripts" section for the `readtype.d` and `writetype.d` scripts, which trace and compare logical to physical I/O.

Strategy

The following approach will help you get started with disk I/O analysis using DTrace. Try the DTrace **one-liners** and **scripts** listed in the sections that follow.

1. In addition to those DTrace tools, familiarize yourself with **existing file system statistical tools**. For example, on Solaris you can use `df(1M)` to list file system usage, as well as a new tool called `fsstat(1)` to show file system I/O types. You can use the metrics from these as starting points for customization with DTrace.

2. Locate or write tools to **generate known file system I/O**, such as running the `dd` command to create files with known numbers of write I/O and to read them back. Filebench can be used to generate sophisticated I/O. It is extremely helpful to have known workloads to check against.
3. **Customize** and write your own one-liners and scripts using the syscall provider. Then try the vminfo and sysinfo providers, if available.
4. Try the currently unstable **fsinfo provider** for more detailed file system scripts, and customize the fsinfo scripts in this chapter.
5. To dig deeper than these providers allow, familiarize yourself with how the kernel and user-land processes call file system I/O by examining stack backtraces (see the "One-Liners" section). Also refer to functional diagrams of the file system subsystem, such as the generic one shown earlier, and others for specific file system types. Check published kernel texts such as *Solaris Internals* (McDougall and Mauro, 2006) and *Mac OS X Internals* (Singh, 2006).
6. Examine **kernel internals** for file systems by using the fbt provider and referring to kernel source code (if available).

Checklist

Table 5-1 describes some potential problem areas with file systems, with suggestions on how you can use DTrace to troubleshoot them.

Table 5-1 File System I/O Checklist

Issue	Description
Volume	Applications may be performing a high volume of file system I/O, which could be avoided or optimized by changing their behavior, for example, by tuning I/O sizes and file locations (tmpfs instead of nfs, for example). The file system may break up I/O into multiple physical I/O of smaller sizes, inflating the IOPS. DTrace can be used to examine file system I/O by process, filename, I/O size, and application stack trace, to identify what files are being used, how, and why.
Latency	A variety of latencies should be examined when analyzing file system I/O: • Disk I/O wait, for reads and synchronous writes • Locking in the file system • Latency of the `open()` syscall • Large file deletion time Each of these can be examined using DTrace.

Table 5-1 File System I/O Checklist (*Continued*)

Issue	Description
Queueing	Use DTrace to examine the size and wait time for file system queues, such as queueing writes for later flushing to disk. Some file systems such as ZFS use a pipeline for all I/O, with certain stages serviced by multiple threads. High latency can occur if a pipeline stage becomes a bottleneck, for example, if compression is performed; this can be analyzed using DTrace.
Caches	File system performance can depend on cache performance: File systems may use multiple caches for different data types (directory names, inodes, metadata, data) and different algorithms for cache replacement and size. DTrace can be used to examine not just the hit and miss rate of caches, but what types of data are experiencing misses, what contents are being evicted, and other internals of cache behavior.
Errors	The file system interface can return errors in many situations: invalid file offsets, permission denied, file not found, and so on. Applications are supposed to catch and deal with these errors with them appropriately, but sometimes they silently fail. Errors returned by file systems can be identified and summarized using DTrace.
Configuration	File access can be tuned by flags, such as those on the `open()` syscall. DTrace can be used to check that the optimum flags are being used by the application, or if it needs to be configured differently.

Providers

Table 5-2 shows providers you can use to trace file system I/O.

Table 5-2 Providers for File System I/O

Provider	Description
syscall	Many syscalls operate on file systems (`open()`, `stat()`, `creat()`, and so on); some operate on file descriptors to file systems (`read()`, `write()`, and so on). By examining file system activity at the syscall interface, user-land context can be examined to see why the file system is being used, such as examining user stack backtraces.
vminfo	Virtual memory info provider. This includes file system page-in and page-out probes (file system disk I/O); however, these only provide number of pages and byte counts.
fsinfo	File system info provider: This is a representation of the VFS layer for the operating system and allows tracing of file system events across different file system types, with file information for each event. This isn't considered a stable provider as the VFS interface can change and is different for different OSs. However, it is unlikely to change rapidly.

continues

Table 5-2 Providers for File System I/O (*Continued*)

Provider	Description
vfs	Virtual File System provider: This is on FreeBSD only and shows VFS and name-cache operations.
io	Trace disk I/O event details including disk, bytes, and latency. Examining stack backtraces from `io:::start` shows why file systems are calling disk I/O.
fbt	Function Boundary Tracing provider. This allows file system internals to be examined in detail, including the operation of file system caches and read ahead. This has an unstable interface and will change between releases of the operating system and file systems, meaning that scripts based on fbt may need to be slightly rewritten for each such update.

Check your operating system to see which providers are available; at the very least, syscall and fbt should be available, which provide a level of coverage of everything.

The vminfo and io providers should also be available on all versions of Solaris 10 and Mac OS X. fsinfo was added to Solaris 10 6/06 (update 2) and Solaris Nevada build 38 and is not yet available on Mac OS X.

fsinfo Provider

The fsinfo provider traces logical file system access. It exports the VFS vnode interface, a private interface for kernel file systems, so fsinfo is considered an unstable provider.

Because the vnode operations it traces are descriptive and resemble many well-known syscalls (`open()`, `close()`, `read()`, `write()`, and so on), this interface provides a generic view of what different file systems are doing and has been exported as the DTrace fsinfo provider.

Listing the fsinfo provider probes on a recent version of Solaris Nevada, we get the following results:

```
# dtrace -ln fsinfo:::
   ID   PROVIDER            MODULE                          FUNCTION NAME
30648     fsinfo           genunix                       fop_vnevent vnevent
30649     fsinfo           genunix                       fop_shrlock shrlock
30650     fsinfo           genunix                    fop_getsecattr getsecattr
30651     fsinfo           genunix                    fop_setsecattr setsecattr
30652     fsinfo           genunix                       fop_dispose dispose
30653     fsinfo           genunix                       fop_dumpctl dumpctl
30654     fsinfo           genunix                        fop_pageio pageio
30655     fsinfo           genunix                      fop_pathconf pathconf
30656     fsinfo           genunix                          fop_dump dump
30657     fsinfo           genunix                          fop_poll poll
```

```
   30658    fsinfo          genunix                        fop_delmap delmap
   30659    fsinfo          genunix                        fop_addmap addmap
   30660    fsinfo          genunix                           fop_map map
   30661    fsinfo          genunix                       fop_putpage putpage
   30662    fsinfo          genunix                       fop_getpage getpage
   30663    fsinfo          genunix                        fop_realvp realvp
   30664    fsinfo          genunix                         fop_space space
   30665    fsinfo          genunix                        fop_frlock frlock
   30666    fsinfo          genunix                           fop_cmp cmp
   30667    fsinfo          genunix                          fop_seek seek
   30668    fsinfo          genunix                      fop_rwunlock rwunlock
   30669    fsinfo          genunix                        fop_rwlock rwlock
   30670    fsinfo          genunix                           fop_fid fid
   30671    fsinfo          genunix                      fop_inactive inactive
   30672    fsinfo          genunix                         fop_fsync fsync
   30673    fsinfo          genunix                      fop_readlink readlink
   30674    fsinfo          genunix                       fop_symlink symlink
   30675    fsinfo          genunix                       fop_readdir readdir
   30676    fsinfo          genunix                         fop_rmdir rmdir
   30677    fsinfo          genunix                         fop_mkdir mkdir
   30678    fsinfo          genunix                        fop_rename rename
   30679    fsinfo          genunix                          fop_link link
   30680    fsinfo          genunix                        fop_remove remove
   30681    fsinfo          genunix                        fop_create create
   30682    fsinfo          genunix                        fop_lookup lookup
   30683    fsinfo          genunix                        fop_access access
   30684    fsinfo          genunix                       fop_setattr setattr
   30685    fsinfo          genunix                       fop_getattr getattr
   30686    fsinfo          genunix                         fop_setfl setfl
   30687    fsinfo          genunix                         fop_ioctl ioctl
   30688    fsinfo          genunix                         fop_write write
   30689    fsinfo          genunix                          fop_read read
   30690    fsinfo          genunix                         fop_close close
   30691    fsinfo          genunix                          fop_open open
```

Table 5-3 fsinfo Probes

Probe	Description
`open`	Attempts to open the file described in the `args[0] fileinfo_t`
`close`	Closes the file described in the `args[0] fileinfo_t`
`read`	Attempts to read `arg1` bytes from the file in `args[0] fileinfo_t`
`write`	Attempts to write `arg1` bytes to the file in `args[0] fileinfo_t`
`fsync`	Calls fsync to synronize the file in `args[0] fileinfo_t`

A selection of these probes is described in Table 5-3.

fileinfo_t

The `fileinfo` structure contains members to describe the file, file system, and open flags of the file that the fsinfo operation is performed on. Some of these members may not be available for particular probes and return `<unknown>`, `<none>`, or 0:

```
typedef struct fileinfo {
        string fi_name;               /* name (basename of fi_pathname) */
        string fi_dirname;            /* directory (dirname of fi_pathname) */
        string fi_pathname;           /* full pathname */
        offset_t fi_offset;           /* offset within file */
        string fi_fs;                 /* file system */
        string fi_mount;              /* mount point of file system */
        int fi_oflags;                /* open(2) flags for file descriptor */
} fileinfo_t;
```

These are translated from the kernel vnode. The `fileinfo_t` structure is also available as the file descriptor array, `fds[]`, which provides convenient file information by file descriptor number. See the one-liners for examples of its usage.

io Provider

The io provider traces physical I/O and was described in Chapter 4.

One-Liners

These one-liners are organized by provider.

syscall Provider

Some of these use the `fds[]` array, which was a later addition to DTrace; for an example of similar functionality predating `fds[]`, see the `rwsnoop` script.

For the one-liners tracing `read(2)` and `write(2)` system calls, be aware that variants may exist (`readv()`, `pread()`, `pread64()`); use the "Count read/write syscalls by syscall type" one-liner to identify which are being used. Also note that these match all reads and writes, whether they are file system based or not, unless matched in a predicate (see the "zfs" one-liner).

Trace file opens with process name:

```
dtrace -n 'syscall::open*:entry { printf("%s %s", execname, copyinstr(arg0)); }'
```

Trace file `creat()` calls with file and process name:

```
dtrace -n 'syscall::creat*:entry { printf("%s %s", execname, copyinstr(arg0)); }'
```

Frequency count `stat()` file calls:

```
dtrace -n 'syscall::stat*:entry { @[copyinstr(arg0)] = count(); }'
```

Tracing the cd(1) command:

```
dtrace -n 'syscall::chdir:entry { printf("%s -> %s", cwd, copyinstr(arg0)); }'
```

Count read/write syscalls by syscall type:

```
dtrace -n 'syscall::*read*:entry,syscall::*write*:entry { @[probefunc] = count(); }'
```

Syscall read(2) by filename:

```
dtrace -n 'syscall::read:entry { @[fds[arg0].fi_pathname] = count(); }'
```

Syscall write(2) by filename:

```
dtrace -n 'syscall::write:entry { @[fds[arg0].fi_pathname] = count(); }'
```

Syscall read(2) by file system type:

```
dtrace -n 'syscall::read:entry { @[fds[arg0].fi_fs] = count(); }'
```

Syscall write(2) by file system type:

```
dtrace -n 'syscall::write:entry { @[fds[arg0].fi_fs] = count(); }'
```

Syscall read(2) by process name for the zfs file system only:

```
dtrace -n 'syscall::read:entry /fds[arg0].fi_fs == "zfs"/ { @[execname] = count(); }'
```

Syscall write(2) by process name and file system type:

```
dtrace -n 'syscall::write:entry { @[execname, fds[arg0].fi_fs] = count(); }
    END { printa("%18s %16s %16@d\n", @); }'
```

vminfo Provider

This processes paging in from the file system:

```
dtrace -n 'vminfo:::fspgin { @[execname] = sum(arg0); }'
```

fsinfo Provider

You can count file system calls by VFS operation:

```
dtrace -n 'fsinfo::: { @[probename] = count(); }'
```

You can count file system calls by mountpoint:

```
dtrace -n 'fsinfo::: { @[args[0]->fi_mount] = count(); }'
```

Bytes read by filename:

```
dtrace -n 'fsinfo:::read { @[args[0]->fi_pathname] = sum(arg1); }'
```

Bytes written by filename:

```
dtrace -n 'fsinfo:::write { @[args[0]->fi_pathname] = sum(arg1); }'
```

Read I/O size distribution by file system mountpoint:

```
dtrace -n 'fsinfo:::read { @[args[0]->fi_mount] = quantize(arg1); }'
```

Write I/O size distribution by file system mountpoint:

```
dtrace -n 'fsinfo:::write { @[args[0]->fi_mount] = quantize(arg1); }'
```

vfs Provider

Count file system calls by VFS operation:

```
dtrace -n 'vfs:vop::entry { @[probefunc] = count(); }'
```

Namecache hit/miss statistics:

```
dtrace -n 'vfs:namecache:lookup: { @[probename] = count(); }'
```

sdt Provider

You can find out who is reading from the ZFS ARC (in-DRAM cache):

```
dtrace -n 'sdt:::arc-hit,sdt:::arc-miss { @[stack()] = count(); }'
```

fbt Provider

The fbt provider instruments a particular operating system and version; these one-liners may therefore require modifications to match the software version you are running.

VFS: You can count file system calls at the fop interface (Solaris):

```
dtrace -n 'fbt::fop_*:entry { @[probefunc] = count(); }'
```

VFS: You can count file system calls at the VNOP interface (Mac OS X):

```
dtrace -n 'fbt::VNOP_*:entry { @[probefunc] = count(); }'
```

VFS: You can count file system calls at the VOP interface (FreeBSD):

```
dtrace -n 'fbt::VOP_*:entry { @[probefunc] = count(); }'
```

ZFS: You can show SPA sync with pool name and TXG number:

```
dtrace -n 'fbt:zfs:spa_sync:entry
{ printf("%s %d", stringof(args[0]->spa_name), arg1); }'
```

One-Liners: syscall Provider Examples

Trace File Opens with Process Name

Tracing opens can be a quick way of getting to know software. Software will often call `open()` on config files, log files, and device files. Sometimes tracing `open()` is a quicker way to find where config and log files exist than to read through the product documentation.

```
# dtrace -n 'syscall::open*:entry { printf("%s %s", execname, copyinstr(arg0)); }'
 29  87276                       open:entry dmake /var/ld/ld.config
 29  87276                       open:entry dmake /lib/libnsl.so.1
 29  87276                       open:entry dmake /lib/libsocket.so.1
 29  87276                       open:entry dmake /lib/librt.so.1
 29  87276                       open:entry dmake /lib/libm.so.1
 29  87276                       open:entry dmake /lib/libc.so.1
 29  87672                     open64:entry dmake /var/run/name_service_door
 29  87276                       open:entry dmake /etc/nsswitch.conf
 12  87276                       open:entry sh /var/ld/ld.config
 12  87276                       open:entry sh /lib/libc.so.1
dtrace: error on enabled probe ID 1 (ID 87672: syscall::open64:entry): invalid address
 (0x8225aff) in action #2 at DIF offset 28
 12  87276                       open:entry sh /var/ld/ld.config
 12  87276                       open:entry sh /lib/libc.so.1
[...]
```

The probe definition uses `open*` so that both `open()` and `open64()` versions are traced. This one-liner has caught a software build in progress; the process names `dmake` and `sh` can be seen, and the files they were opening are mostly library files under `/lib`.

The `dtrace` error is likely due to `copyinstr()` operating on a text string that hasn't been faulted into the process's virtual memory address space yet. The page fault would happen during the `open()` syscall, but we've traced it before it has happened. This can be solved by saving the address on `open*:entry` and using `copyinstr()` on `open*:return`, after the string is in memory.

Trace File creat() Calls with Process Name

This also caught a software build in progress. Here the `cp` command is creating files as part of the build. The Bourne shell `sh` also appears to be creating `/dev/null`; this is happening as part of shell redirection.

```
# dtrace -n 'syscall::creat*:entry { printf("%s %s", execname, copyinstr(arg0)); }'
dtrace: description 'syscall::creat*:entry ' matched 2 probes
```

```
CPU     ID                    FUNCTION:NAME
 25  87670                    creat64:entry cp /builds/brendan/ak-on-new/proto/root_i3
86/platform/i86xpv/kernel/misc/amd64/xpv_autoconfig
 31  87670                    creat64:entry sh /dev/null
  0  87670                    creat64:entry cp /builds/brendan/ak-on-new/proto/root_i3
86/platform/i86xpv/kernel/drv/xdf
 20  87670                    creat64:entry sh /dev/null
 26  87670                    creat64:entry sh /dev/null
 27  87670                    creat64:entry sh /dev/null
 31  87670                    creat64:entry cp /builds/brendan/ak-on-new/proto/root_i3
86/usr/lib/llib-l300.ln
  0  87670                    creat64:entry cp /builds/brendan/ak-on-new/proto/root_i3
86/kernel/drv/amd64/iwscn
 12  87670                    creat64:entry cp /builds/brendan/ak-on-new/proto/root_i3
86/platform/i86xpv/kernel/drv/xnf
 16  87670                    creat64:entry sh obj32/ao_mca_disp.c
[...]
```

Frequency Count stat() Files

As a demonstration of frequency counting instead of tracing and of examining the `stat()` syscall, this frequency counts filenames from `stat()`:

```
# dtrace -n 'syscall::stat*:entry { @[copyinstr(arg0)] = count(); }'
dtrace: description 'syscall::stat*:entry ' matched 5 probes
^C

  /builds/brendan/ak-on-new/proto/root_i386/kernel/drv/amd64/mxfe/mxfe
1
  /builds/brendan/ak-on-new/proto/root_i386/kernel/drv/amd64/rtls/rtls
1
  /builds/brendan/ak-on-new/proto/root_i386/usr/kernel/drv/ii/ii                1
  /lib/libmakestate.so.1                                            1
  /tmp/dmake.stdout.10533.189.ejaOKu                                1
[...output truncated...]
  /ws/onnv-tools/SUNWspro/SS12/prod/lib/libmd5.so.1               105
  /ws/onnv-tools/SUNWspro/SS12/prod/lib/sys/libc.so.1              105
  /ws/onnv-tools/SUNWspro/SS12/prod/lib/sys/libmd5.so.1              105
  /ws/onnv-tools/SUNWspro/SS12/prod/bin/../lib/libc.so.1              106
  /ws/onnv-tools/SUNWspro/SS12/prod/bin/../lib/lib_I_dbg_gen.so.1              107
  /lib/libm.so.1                                                  112
  /lib/libelf.so.1                                                136
  /lib/libdl.so.1                                                 151
  /lib/libc.so.1                                                  427
  /tmp                                                            638
```

During tracing, `stat()` was called on `/tmp` 638 times. A wildcard is used in the probe name so that this one-liner matches both `stat()` and `stat64()`; however, applications could be using other variants such as `xstat()` that this isn't matching.

Tracing cd

You can trace the current working directory (pwd) and chdir directory (cd) using the following one-liner:

```
# dtrace -n 'syscall::chdir:entry { printf("%s -> %s", cwd, copyinstr(arg0)); }'
dtrace: description 'syscall::chdir:entry ' matched 1 probe
CPU     ID      FUNCTION:NAME
  4  87290       chdir:entry /builds/brendan/ak-on-new/usr/src/uts/intel -> aac
  5  87290       chdir:entry /builds/brendan/ak-on-new/usr/src/uts/intel -> amd64_gart
  8  87290       chdir:entry /builds/brendan/ak-on-new/usr/src/uts/intel -> amr
  9  87290       chdir:entry /builds/brendan/ak-on-new/usr/src/uts/intel -> agptarget
 12  87290       chdir:entry /builds/brendan/ak-on-new/usr/src/uts/intel -> aggr
 12  87290       chdir:entry /builds/brendan/ak-on-new/usr/src/uts/intel -> agpgart
 16  87290       chdir:entry /builds/brendan/ak-on-new/usr/src/uts/intel -> ahci
 16  87290       chdir:entry /builds/brendan/ak-on-new/usr/src/uts/intel -> arp
[...]
```

This output shows a software build iterating over subdirectories.

Reads by File System Type

During this build, tmpfs is currently receiving the most reads: 128,645 during this trace, followed by ZFS at 65,919.

```
# dtrace -n 'syscall::read:entry { @[fds[arg0].fi_fs] = count(); }'
dtrace: description 'syscall::read:entry ' matched 1 probe
^C

  specfs                                                           22
  sockfs                                                           28
  proc                                                            103
  <none>                                                          136
  nfs4                                                            304
  fifofs                                                         1571
  zfs                                                           65919
  tmpfs                                                        128645
```

Note that this one-liner is matching only the read variant of the read() syscall. On Solaris, applications may be calling readv(), pread(), or pread64(); Mac OS X has readv(), pread(), read_nocancel(), and pread_nocancel(); and FreeBSD has more, including aio_read(). You can match all of these using wildcards:

```
solaris# dtrace -ln 'syscall::*read*:entry'
   ID   PROVIDER            MODULE                          FUNCTION NAME
87272    syscall                                                read entry
87418    syscall                                            readlink entry
87472    syscall                                               readv entry
87574    syscall                                               pread entry
87666    syscall                                             pread64 entry
```

However, this also matches `readlink()`, and our earlier one-liner assumes that `arg0` is the file descriptor, which is not the case for `readlink()`. Tracing all read types properly will require a short script rather than a one-liner.

Writes by File System Type

This one-liner matches all variants of write, assuming that `arg0` is the file descriptor. In this example, most of the writes were to tmpfs (`/tmp`).

```
# dtrace -n 'syscall::*write*:entry { @[fds[arg0].fi_fs] = count(); }'
dtrace: description 'syscall::write:entry ' matched 1 probe
^C

  specfs                                                            2
  nfs4                                                             47
  sockfs                                                           55
  zfs                                                             154
  fifofs                                                          243
  tmpfs                                                         22245
```

Writes by Process Name and File System Type

This example extends the previous one-liner to include the process name:

```
# dtrace -n 'syscall::write:entry { @[execname, fds[arg0].fi_fs] = count(); }
END { printa("%18s %16s %16@d\n", @); }'
dtrace: description 'syscall::write:entry ' matched 2 probes
^C
CPU     ID                    FUNCTION:NAME
 25      2                             :END               ar         zfs           1
            dtrace           specfs                1
                sh           fifofs                1
              sshd           specfs                1
  ssh-socks5-proxy           fifofs                2
             uname           fifofs                3
               sed              zfs                4
               ssh           fifofs               10
             strip              zfs               15
[...truncated...]
               gas            tmpfs              830
             acomp            tmpfs             2072
               ube            tmpfs             2487
             ir2hf            tmpfs             2608
             iropt            tmpfs             3364
```

Now we can see the processes that were writing to tmpfs: `iropt`, `ir2hf`, and so on.

One-Liners: vminfo Provider Examples

Processes Paging in from the File System

The vminfo provider has a probe for file system page-ins, which can give a very rough idea of which processes are reading from disk via a file system:

```
# dtrace -n 'vminfo:::fspgin { @[execname] = sum(arg0); }'
dtrace: description 'vminfo:::fspgin ' matched 1 probe
^C

  dmake                                                             1
  scp                                                               2
  sched                                                            42
```

This worked a little: Both `dmake` and `scp` are responsible for paging in file system data. However, it has identified `sched` (the kernel) as responsible for the most page-ins. This could be because of read-ahead occurring in kernel context; more DTrace will be required to explain where the `sched` page-ins were from.

One-Liners: fsinfo Provider Examples

File System Calls by fs Operation

This uses the fsinfo provider, if available. Since it traces file system activity at the VFS layer, it will see activity from all file system types: ZFS, UFS, HSFS, and so on.

```
# dtrace -n 'fsinfo::: { @[probename] = count(); }'
dtrace: description 'fsinfo::: ' matched 44 probes
^C

  rename                                                            2
  symlink                                                           4
  create                                                            6
  getsecattr                                                        6
  seek                                                              8
  remove                                                           10
  poll                                                             40
  readlink                                                         40
  write                                                            42
  realvp                                                           52
  map                                                             144
  read                                                            171
  addmap                                                          192
  open                                                            193
  delmap                                                          194
  close                                                           213
  readdir                                                         225
  dispose                                                         230
  access                                                          248
  ioctl                                                           421
  rwlock                                                          436
  rwunlock                                                        436
```

```
  getpage                                                     1700
  getattr                                                     3221
  cmp                                                        48342
  putpage                                                    77557
  inactive                                                   80786
  lookup                                                     86059
```

The most frequent vnode operation was `lookup()`, called 86,059 times while this one-liner was tracing.

File System Calls by Mountpoint

The fsinfo provider has `fileinfo_t` as `args[0]`. Here the mountpoint is frequency counted by fsinfo probe call, to get a rough idea of how busy (by call count) file systems are as follows:

```
# dtrace -n 'fsinfo::: { @[args[0]->fi_mount] = count(); }'
dtrace: description 'fsinfo::: ' matched 44 probes
^C

  /home                                                          8
  /builds/bmc                                                    9
  /var/run                                                      11
  /builds/ahl                                                   24
  /home/brendan                                                 24
  /etc/svc/volatile                                             47
  /etc/svc                                                      50
  /var                                                          94
  /net/fw/export/install                                       176
  /ws                                                          252
  /lib/libc.so.1                                               272
  /etc/mnttab                                                  388
  /ws/onnv-tools                                              1759
  /builds/brendan                                            17017
  /tmp                                                      156487
  /                                                         580819
```

Even though I'm doing a source build in `/builds/brendan`, it's the root file system on / that has received the most file system calls.

Bytes Read by Filename

The fsinfo provider gives an abstracted file system view that isn't dependent on syscall variants such as `read()`, `pread()`, `pread64()`, and so on.

```
# dtrace -n 'fsinfo:::read { @[args[0]->fi_pathname] = sum(arg1); }'
dtrace: description 'fsinfo:::read ' matched 1 probe
^C

  /usr/bin/chmod                                               317
  /home/brendan/.make.machines                                 572
```

continues

```
    /usr/bin/chown                                                 951
    <unknown>                                                     1176
    /usr/bin/chgrp                                                1585
    /usr/bin/mv                                                   1585
  [...output truncated...]
    /builds/brendan/ak-on-new/usr/src/uts/intel/Makefile.rules           325056
    /builds/brendan/ak-on-new/usr/src/uts/intel/Makefile.intel.shared           415752
    /builds/brendan/ak-on-new/usr/src/uts/intel/arn/.make.state           515044
    /builds/brendan/ak-on-new/usr/src/uts/Makefile.uts           538440
    /builds/brendan/ak-on-new/usr/src/Makefile.master            759744
    /builds/brendan/ak-on-new/usr/src/uts/intel/ata/.make.state           781904
    /builds/brendan/ak-on-new/usr/src/uts/common/Makefile.files           991896
    /builds/brendan/ak-on-new/usr/src/uts/common/Makefile.rules          1668528
    /builds/brendan/ak-on-new/usr/src/uts/intel/genunix/.make.state          5899453
```

The file being read the most is a `.make.state` file: During tracing, more than 5MB was read from the file. The fsinfo provider traces these reads to the file system: The file may have been entirely cached in DRAM or read from disk. To determine how the read was satisfied by the file system, we'll need to DTrace further down the I/O stack (see the "Scripts" section and Chapter 4, Disk I/O).

Bytes Written by Filename

During tracing, a `.make.state.tmp` file was written to the most, with more than 1MB of writes. As with reads, this is writing to the file system. This may not write to disk until sometime later, when the file system flushes dirty data.

```
# dtrace -n 'fsinfo:::write { @[args[0]->fi_pathname] = sum(arg1); }'
dtrace: description 'fsinfo:::write ' matched 1 probe
^C

  /tmp/DAA1RaGkd                                                   22
  /tmp/DAA5JaO6c                                                   22
[...truncated...]
  /tmp/iroptEAA.1524.dNaG.c                                    250588
  /tmp/acompBAA.1443.MGay0c                                    305541
  /tmp/iroptDAA.1443.OGay0c                                    331906
  /tmp/acompBAA.1524.aNaG.c                                    343015
  /tmp/iroptDAA.1524.cNaG.c                                    382413
  /builds/brendan/ak-on-new/usr/src/cmd/fs.d/.make.state.tmp          1318590
```

Read I/O Size Distribution by File System Mountpoint

This output shows a distribution plot of read size by file system. The `/builds/brendan` file system was usually read at between 1,024 and 131,072 bytes per read. The largest read was in the 1MB to 2MB range.

```
# dtrace -n 'fsinfo:::read { @[args[0]->fi_mount] = quantize(arg1); }'
dtrace: description 'fsinfo:::read ' matched 1 probe
^C
```

```
  /builds/bmc
           value  ------------- Distribution ------------- count
              -1 |                                         0
               0 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 2
               1 |                                         0

[...output truncated...]

  /builds/brendan
           value  ------------- Distribution ------------- count
              -1 |                                         0
               0 |@                                        15
               1 |                                         0
               2 |                                         0
               4 |                                         0
               8 |                                         0
              16 |                                         0
              32 |                                         0
              64 |@@                                       28
             128 |                                         0
             256 |                                         0
             512 |@@                                       28
            1024 |@@@@@@@                                  93
            2048 |@@@@                                     52
            4096 |@@@@@@                                   87
            8192 |@@@@@@@                                  94
           16384 |@@@@@@@@                                 109
           32768 |@@                                       31
           65536 |@@                                       30
          131072 |                                         0
          262144 |                                         2
          524288 |                                         1
         1048576 |                                         1
         2097152 |                                         0
```

Write I/O Size Distribution by File System Mountpoint

During tracing, `/tmp` was written to the most (listed last), mostly with I/O sizes between 4KB and 8KB.

```
# dtrace -n 'fsinfo:::write { @[args[0]->fi_mount] = quantize(arg1); }'
dtrace: description 'fsinfo:::write ' matched 1 probe
^C

  /etc/svc/volatile
           value  ------------- Distribution ------------- count
             128 |                                         0
             256 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 34
             512 |                                         0
[...]

  /tmp
           value  ------------- Distribution ------------- count
               2 |                                         0
               4 |                                         1
               8 |                                         4
              16 |@@@@                                     121
              32 |@@@@                                     133
              64 |@@                                       56
             128 |@@                                       51
```

continues

```
             256 |@                                        46
             512 |@                                        39
            1024 |@                                        32
            2048 |@@                                       52
            4096 |@@@@@@@@@@@@@@@@@@@@@@@@                 820
            8192 |                                         0
```

One-Liners: sdt Provider Examples

Who Is Reading from the ZFS ARC?

This shows who is performing reads to the ZFS ARC (the in-DRAM file system cache for ZFS) by counting the stack backtraces for all ARC accesses. It uses SDT probes, which have been in the ZFS ARC code for a while:

```
# dtrace -n 'sdt:::arc-hit,sdt:::arc-miss { @[stack()] = count(); }'
dtrace: description 'sdt:::arc-hit,sdt:::arc-miss ' matched 3 probes
^C
[...]

              zfs`arc_read+0x75
              zfs`dbuf_prefetch+0x131
              zfs`dmu_prefetch+0x8f
              zfs`zfs_readdir+0x4a2
              genunix`fop_readdir+0xab
              genunix`getdents64+0xbc
              unix`sys_syscall32+0x101
              245

              zfs`dbuf_hold_impl+0xea
              zfs`dbuf_hold+0x2e
              zfs`dmu_buf_hold_array_by_dnode+0x195
              zfs`dmu_buf_hold_array+0x73
              zfs`dmu_read_uio+0x4d
              zfs`zfs_read+0x19a
              genunix`fop_read+0x6b
              genunix`read+0x2b8
              genunix`read32+0x22
              unix`sys_syscall32+0x101
              457

              zfs`dbuf_hold_impl+0xea
              zfs`dbuf_hold+0x2e
              zfs`dmu_buf_hold+0x75
              zfs`zap_lockdir+0x67
              zfs`zap_cursor_retrieve+0x74
              zfs`zfs_readdir+0x29e
              genunix`fop_readdir+0xab
              genunix`getdents64+0xbc
              unix`sys_syscall32+0x101
             1004

              zfs`dbuf_hold_impl+0xea
              zfs`dbuf_hold+0x2e
              zfs`dmu_buf_hold+0x75
              zfs`zap_lockdir+0x67
              zfs`zap_lookup_norm+0x55
              zfs`zap_lookup+0x2d
```

```
                zfs`zfs_match_find+0xfd
                zfs`zfs_dirent_lock+0x3d1
                zfs`zfs_dirlook+0xd9
                zfs`zfs_lookup+0x104
                genunix`fop_lookup+0xed
                genunix`lookuppnvp+0x3a3
                genunix`lookuppnat+0x12c
                genunix`lookupnameat+0x91
                genunix`cstatat_getvp+0x164
                genunix`cstatat64_32+0x82
                genunix`lstat64_32+0x31
                unix`sys_syscall32+0x101
               2907
```

This output is interesting because it demonstrates four different types of ZFS ARC read. Each stack is, in order, as follows.

1. `prefetch read`: ZFS performs prefetch before reading from the ARC. Some of the prefetch requests will actually just be cache hits; only the prefetch requests that miss the ARC will pull data from disk.
2. `syscall read`: Most likely a process reading from a file on ZFS.
3. `read dir`: Fetching directory contents.
4. `stat`: Fetching file information.

Scripts

Table 5-4 summarizes the scripts that follow and the providers they use.

Table 5-4 Script Summary

Script	Target	Description	Providers
`sysfs.d`	Syscalls	Shows reads and writes by process and mountpoint	syscall
`fsrwcount.d`	Syscalls	Counts read/write syscalls by file system and type	syscall
`fsrwtime.d`	Syscalls	Measures time in read/write syscalls by file system	syscall
`fsrtpk.d`	Syscalls	Measures file system read time per kilobyte	syscall
`rwsnoop`	Syscalls	Traces syscall read and writes, with FS details	syscall
`mmap.d`	Syscalls	Traces `mmap()` of files with details	syscall
`fserrors.d`	Syscalls	Shows file system syscall errors	syscall

continues

Table 5-4 Script Summary (*Continued*)

Script	Target	Description	Providers
fswho.d[1]	VFS	Summarizes processes and file read/writes	fsinfo
readtype.d[1]	VFS	Compares logical vs. physical file system reads	fsinfo, io
writetype.d[1]	VFS	Compares logical vs. physical file system writes	fsinfo, io
fssnoop.d	VFS	Traces file system calls using fsinfo	fsinfo
solvfssnoop.d	VFS	Traces file system calls using fbt on Solaris	fbt
macvfssnoop.d	VFS	Traces file system calls using fbt on Mac OS X	fbt
vfssnoop.d	VFS	Traces file system calls using vfs on FreeBSD	vfs
sollife.d	VFS	Shows file creation and deletion on Solaris	fbt
maclife.d	VFS	Shows file creation and deletion on Mac OS X	fbt
vfslife.d	VFS	Shows file creation and deletion on FreeBSD	vfs
dnlcps.d	VFS	Shows Directory Name Lookup Cache hits by process[2]	fbt
fsflush_cpu.d	VFS	Shows file system flush tracer CPU time[2]	fbt
fsflush.d	VFS	Shows file system flush statistics[2]	profile
ufssnoop.d	UFS	Traces UFS calls directly using fbt[2]	fbt
ufsreadahead.d	UFS	Shows UFS read-ahead rates for sequential I/O[2]	fbt
ufsimiss.d	UFS	Traces UFS inode cache misses with details[2]	fbt
zfssnoop.d	ZFS	Traces ZFS calls directly using fbt[2]	fbt
zfsslower.d	ZFS	Traces slow HFS+ read/writes[2]	fbt
zioprint.d	ZFS	Shows ZIO event dump[2]	fbt
ziosnoop.d	ZFS	Shows ZIO event tracing, detailed[2]	fbt
ziotype.d	ZFS	Shows ZIO type summary by pool[2]	fbt
perturbation.d	ZFS	Shows ZFS read/write time during given perturbation[2]	fbt
spasync.d	ZFS	Shows SPA sync tracing with details[2]	fbt
hfssnoop.d	HFS+	Traces HFS+ calls directly using fbt[3]	fbt
hfsslower.d	HFS+	Traces slow HFS+ read/writes[3]	fbt
hfsfileread.d	HFS+	Shows logical/physical reads by file[3]	fbt
pcfsrw.d	PCFS	Traces pcfs (FAT16/32) read/writes[2]	fbt
cdrom.d	HSFS	Traces CDROM insertion and mount[2]	fbt
dvd.d	UDFS	Traces DVD insertion and mount[2]	fbt
nfswizard.d	NFS	Summarizes NFS performance client-side[2]	io

Table 5-4 Script Summary (*Continued*)

Script	Target	Description	Providers
`nfs3sizes.d`	NFSv3	Shows NFSv3 logical vs physical read sizes[2]	fbt
`nfs3fileread.d`	NFSv3	Shows NFSv3 logical vs physical reads by file[2]	fbt
`tmpusers.d`	TMPFS	Shows users of `/tmp` and `tmpfs` by tracing `open()`[2]	fbt
`tmpgetpage.d`	TMPFS	Measures whether tmpfs paging is occurring, with I/O time[2]	fbt

[1] This uses the fsinfo provider, currently available only on Oracle Solaris.

[2] This is written for Oracle Solaris.

[3] This is written for Apple Mac OS X.

There is an emphasis on the syscall and VFS layer scripts, since these can be used on any underlying file system type.

Note that the fbt provider is considered an "unstable" interface, because it instruments a specific operating system or application version. For this reason, scripts that use the fbt provider may require changes to match the version of the software you are using. These scripts have been included here as examples of D programming and of the kind of data that DTrace can provide for each of these topics. See Chapter 12, Kernel, for more discussion about using the fbt provider.

Syscall Provider

File system tracing scripts based on the syscall provider are generic and work across all file systems. At the syscall level, you can see "logical" file system I/O, the I/O that the application requests from the file system. Actual disk I/O occurs after file system processing and may not match the requested logical I/O (for example, rounding I/O size up to the file system block size).

sysfs.d

The `sysfs.d` script traces read and write syscalls to show which process is performing reads and writes on which file system.

Script

This script is written to work on both Solaris and Mac OS X. Matching all the possible `read()` variants (`read()`, `readv()`, `pread()`, `pread64()`, `read_nocancel()`, and so on) for Solaris and Mac OS X proved a little tricky and led to the probe definitions on lines 11 to 14. Attempting to match `syscall::*read*:entry` doesn't work, because it matches `readlink()` and pthread syscalls (on Mac OS X), neither of which we are trying to trace (we want a `read()` style syscall with a file descriptor as `arg0`, for line 17 to use).

The `-Z` option prevents DTrace on Solaris complaining about line 14, which is just there for the Mac OS X `read_nocancel()` variants. Without it, this script wouldn't execute because DTrace would fail to find probes for `syscall::*read*nocancel:entry`.

```
1   #!/usr/sbin/dtrace -Zs
2
3   #pragma D option quiet
4
5   dtrace:::BEGIN
6   {
7           printf("Tracing... Hit Ctrl-C to end.\n");
8   }
9
10  /* trace read() variants, but not readlink() or __pthread*() (macosx) */
11  syscall::read:entry,
12  syscall::readv:entry,
13  syscall::pread*:entry,
14  syscall::*read*nocancel:entry,
15  syscall::*write*:entry
16  {
17          @[execname, probefunc, fds[arg0].fi_mount] = count();
18  }
19
20  dtrace:::END
21  {
22          printf("  %-16s %-16s %-30s %7s\n", "PROCESS", "SYSCALL",
23              "MOUNTPOINT", "COUNT");
24          printa("  %-16s %-16s %-30s %@7d\n", @);
25  }
```

Script sysfs.d

Example

This was executed on a software build server. The busiest process name during tracing was `diff`, performing reads on the `/ws/ak-on-gate/public` file system. This was probably multiple `diff(1)` commands; the `sysfs.d` script could be modified to include a PID if it was desirable to split up the PIDs (although in this case it helps to aggregate the build processes together).

Some of the reads and writes to the `/` mountpoint may have been to device paths in `/dev`, including `/dev/tty` (terminal); to differentiate between these and I/O to the root file system, enhance the script to include a column for `fds[arg0].fi_fs`—the file system type (see `fsrwcount.d`).

```
# sysfs.d
Tracing... Hit Ctrl-C to end.
^C
```

```
  PROCESS          SYSCALL          MOUNTPOINT                           COUNT
  hg               write            /devices                                 1
  in.mpathd        read             /                                        1
  in.mpathd        write            /                                        1
[...truncated...]
  nawk             write            /tmp                                    36
  dmake            write            /builds/brendan                         40
  nawk             write            /ws/ak-on-gate/public                   50
  dmake            read             /var                                    54
  codereview       write            /tmp                                    61
  ksh93            write            /ws/ak-on-gate/public                   65
  expand           read             /                                       69
  nawk             read             /                                       69
  expand           write            /                                       72
  sed              read             /tmp                                   100
  nawk             read             /tmp                                   113
  dmake            read             /                                      209
  dmake            read             /builds/brendan                        249
  hg               read             /                                      250
  hg               read             /builds/fishgk                         260
  sed              read             /ws/ak-on-gate/public                  430
  diff             read             /ws/ak-on-gate/public                 2592
```

fsrwcount.d

You can count read/write syscall operations by file system and type.

Script

This is similar to `sysfs.d`, but it prints the file system type instead of the process name:

```
#!/usr/sbin/dtrace -Zs

#pragma D option quiet

dtrace:::BEGIN
{
        printf("Tracing... Hit Ctrl-C to end.\n");
}

/* trace read() variants, but not readlink() or __pthread*() (macosx) */
syscall::read:entry,
syscall::readv:entry,
syscall::pread*:entry,
syscall::*read*nocancel:entry,
syscall::*write*:entry
{
        @[fds[arg0].fi_fs, probefunc, fds[arg0].fi_mount] = count();
}

dtrace:::END
{
        printf("  %-9s  %-16s %-40s %7s\n", "FS", "SYSCALL", "MOUNTPOINT",
            "COUNT");
        printa("  %-9.9s  %-16s %-40s %@7d\n", @);
}
```

Script fsrwcount.d

Example

Here's an example of running `fsrwcount.d` on Solaris:

```
# fsrwcount.d
Tracing... Hit Ctrl-C to end.
^C
  FS         SYSCALL          MOUNTPOINT                                 COUNT
  specfs     write            /                                              1
  nfs4       read             /ws/onnv-tools                                 3
  zfs        read             /builds/bmc                                    5
  nfs4       read             /home/brendan                                 11
  zfs        read             /builds/ahl                                   16
  sockfs     writev           /                                             20
  zfs        write            /builds/brendan                               30
  <none>     read             <none>                                        33
  sockfs     write            /                                             34
  zfs        read             /var                                          88
  sockfs     read             /                                            104
  zfs        read             /builds/fishgk                               133
  nfs4       write            /ws/ak-on-gate/public                        171
  tmpfs      write            /tmp                                         197
  zfs        read             /builds/brendan                              236
  tmpfs      read             /tmp                                         265
  fifofs     write            /                                            457
  fifofs     read             /                                            625
  zfs        read             /                                            809
  nfs4       read             /ws/ak-on-gate/public                       1673
```

During a software build, this has shown that most of the file system syscalls were reads to the NFSv4 share `/ws/ak-on-gate/public`. The busiest ZFS file systems were / followed by `/builds/brendan`.

Here's an example of running `fsrwcount.d` on Mac OS X:

```
# fsrwcount.d
Tracing... Hit Ctrl-C to end.
^C
  FS         SYSCALL          MOUNTPOINT                                 COUNT
  devfs      write            dev                                            2
  devfs      write_nocancel   dev                                            2
  <unknown   write_nocancel   <unknown (not a vnode)>                        3
  hfs        write_nocancel   /                                              6
  devfs      read             dev                                            7
  devfs      read_nocancel    dev                                            7
  hfs        write            /                                             18
  <unknown   write            <unknown (not a vnode)>                       54
  hfs        read_nocancel    /                                             55
  <unknown   read             <unknown (not a vnode)>                      134
  hfs        pwrite           /                                            155
  hfs        read             /                                            507
  hfs        pread            /                                           1760
```

This helps explain line 24, which truncated the FS field to nine characters (`%9.9s`). On Mac OS X, `<unknown (not a vnode>)` may be returned, and without the truncation the columns become crooked. These nonvnode operations may be reads and writes to sockets.

fsrwtime.d

The `fsrwtime.d` script measures the time spent in read and write syscalls, with file system information. The results are printed in distribution plots by microsecond.

Script

If averages or sums are desired instead, change the aggregating function on line 20 and the output formatting on line 26:

```
1   #!/usr/sbin/dtrace -Zs
2
3   /* trace read() variants, but not readlink() or __pthread*() (macosx) */
4   syscall::read:entry,
5   syscall::readv:entry,
6   syscall::pread*:entry,
7   syscall::*read*nocancel:entry,
8   syscall::*write*:entry
9   {
10          self->fd = arg0;
11          self->start = timestamp;
12  }
13
14  syscall::*read*:return,
15  syscall::*write*:return
16  /self->start/
17  {
18          this->delta = (timestamp - self->start) / 1000;
19          @[fds[self->fd].fi_fs, probefunc, fds[self->fd].fi_mount] =
20              quantize(this->delta);
21          self->fd = 0; self->start = 0;
22  }
23
24  dtrace:::END
25  {
26          printa("\n  %s %s (us) \t%s%@d", @);
27  }
```

Script fsrwtime.d

The syscall return probes on lines 14 and 15 use more wildcards without fear of matching unwanted syscalls (such as `readlink()`), since it also checks for `self->start` to be set in the predicate, which will be true only for the syscalls that matched the precise set on lines 4 to 8.

Example

This output shows that `/builds/brendan`, a ZFS file system, mostly returned reads between 8 us and 127 us. These are likely to have returned from the ZFS file system cache, the ARC. The single read that took more than 32 ms is likely to have been returned from disk. More DTracing can confirm.

```
# fsrwtime.d
dtrace: script 'fsrwtime.d' matched 18 probes
^C
CPU     ID                    FUNCTION:NAME
  8      2                             :END
  specfs read (us)      /devices
           value  ------------- Distribution ------------- count
               4 |                                         0
               8 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 1
              16 |                                         0
[...]

  zfs write (us)        /builds/brendan
           value  ------------- Distribution ------------- count
               8 |                                         0
              16 |@@@@@                                    4
              32 |@@@@@@@@@@@@@@                           11
              64 |@@@@@@@@@@@@@@@@@@@@@                    17
             128 |                                         0

  zfs read (us)         /builds/brendan
           value  ------------- Distribution ------------- count
               4 |                                         0
               8 |@@@@@@@@@@@@@@@@                         72
              16 |@@@@@@@@@@                               44
              32 |@@@@@@@                                  32
              64 |@@@@@                                    24
             128 |                                         0
             256 |@                                        3
             512 |                                         1
            1024 |                                         0
            2048 |                                         0
            4096 |                                         0
            8192 |                                         0
           16384 |                                         0
           32768 |                                         1
           65536 |                                         0
```

fsrtpk.d

As an example of a different way to analyze time, the `fsrtpk.d` script shows file system read *time per kilobyte*.

Script

This is similar to the `fsrwtime.d` script, but here we divide the time by the number of kilobytes, as read from `arg0` (rval) on read return:

```
#!/usr/sbin/dtrace -Zs

/* trace read() variants, but not readlink() or __pthread*() (macosx) */
syscall::read:entry,
syscall::readv:entry,
syscall::pread*:entry,
syscall::*read*nocancel:entry
{
        self->fd = arg0;
        self->start = timestamp;
}
```

```

syscall::*read*:return
/self->start && arg0 > 0/
{
        this->kb = (arg1 / 1024) ? arg1 / 1024 : 1;
        this->ns_per_kb = (timestamp - self->start) / this->kb;
        @[fds[self->fd].fi_fs, probefunc, fds[self->fd].fi_mount] =
            quantize(this->ns_per_kb);
}

syscall::*read*:return
{
        self->fd = 0; self->start = 0;
}

dtrace:::END
{
        printa("\n  %s %s (ns per kb) \t%s%@d", @);
}
```

Script fsrtpk.d

Example

For the same interval, compare `fsrwtime.d` and `fsrtpk.d`:

```
# fsrwtime.d
[...]
  zfs read (us)         /export/fs1
           value  ------------- Distribution ------------- count
               0 |                                         0
               1 |                                         7
               2 |                                         63
               4 |                                         10
               8 |                                         15
              16 |@                                        3141
              32 |@@@@@@                                   27739
              64 |@@@@@@@@@@@                              55730
             128 |@@@@@@@@                                 39625
             256 |@@@@@@@                                  34358
             512 |@@@@                                     18700
            1024 |@@                                       8514
            2048 |@@                                       8407
            4096 |                                         361
            8192 |                                         32
           16384 |                                         1
           32768 |                                         0

# fsrtpk.d
[...]
  zfs read (ns per kb)  /export/fs1
           value  ------------- Distribution ------------- count
             128 |                                         0
             256 |@@@@@@@@@@@@@@@@@@@@@@                   109467
             512 |@@@@@@@@@@@@@@@@                         79390
            1024 |@@                                       7643
            2048 |                                         106
            4096 |                                         2
            8192 |                                         0
```

From `fstime.d`, the reads to zfs are quite varied, mostly falling between 32 us and 1024 us. The reason was not varying ZFS performance but varying requested I/O sizes to cached files: Larger I/O sizes take longer to complete because of the movement of data bytes in memory.

The read time per kilobyte is much more consistent, regardless of the I/O size, returning between 256 ns and 1023 ns per kilobyte read.

rwsnoop

The `rwsnoop` script traces `read()` and `write()` syscalls across the system, printing process and size details as they occur. Since these are usually frequent syscalls, the output can be verbose and also prone to feedback loops (this is because the lines of output from `dtrace(1M)` are performed using `write()`, which are also traced by DTrace, triggering more output lines, and so on). The `-n` option can be used to avoid this, allowing process names of interest to be specified.

These syscalls are generic and not exclusively for file system I/O; check the `FILE` column in the output of the script for those that are reading and writing to files.

Script

Since most of this 234-line script handles command-line options, the only interesting DTrace parts are included here. The full script is in the DTraceToolkit and can also be found in `/usr/bin/rwsnoop` on Mac OS X.

The script saves various details in thread-local variables. Here the direction and size of `read()` calls are saved:

```
182   syscall::*read:return
183   /self->ok/
184   {
185           self->rw = "R";
186           self->size = arg0;
187   }
```

which it then prints later:

```
202   syscall::*read:return,
203   syscall::*write:entry
[...]
225           printf("%5d %6d %-12.12s %1s %7d %s\n",
226               uid, pid, execname, self->rw, (int)self->size, self->vpath);
```

This is straightforward. What's not straightforward is the way the file path name is fetched from the file descriptor saved in `self->fd` (line 211):

```
syscall::*read:return,
syscall::*write:entry
/self->ok/
{
        /*
         * Fetch filename
         */
        this->filistp = curthread->t_procp->p_user.u_finfo.fi_list;
        this->ufentryp = (uf_entry_t *)((uint64_t)this->filistp +
            (uint64_t)self->fd * (uint64_t)sizeof(uf_entry_t));
        this->filep = this->ufentryp->uf_file;
        this->vnodep = this->filep != 0 ? this->filep->f_vnode : 0;
        self->vpath = this->vnodep ? (this->vnodep->v_path != 0 ?
            cleanpath(this->vnodep->v_path) : "<unknown>") : "<unknown>";
```

This lump of code digs out the path name from the Solaris kernel and was written this way because `rwsnoop` predates the `fds` array being available in Solaris. With the availability of the `fds[]` array, that entire block of code can be written as follows:

```
self->vpath = fds[self->fd].fi_pathname
```

unless you are using a version of DTrace that doesn't yet have the `fds` array, such as FreeBSD, in which case you can try writing the FreeBSD version of the previous code block.

Examples

The following examples demonstrate the use of the `rwsnoop` script.

Usage: rwsnoop.d.

```
# rwsnoop -h
USAGE: rwsnoop [-hjPtvZ] [-n name] [-p pid]

                -j       # print project ID
                -P       # print parent process ID
                -t       # print timestamp, us
                -v       # print time, string
                -Z       # print zone ID
                -n name  # this process name only
                -p PID   # this PID only
   eg,
        rwsnoop          # default output
        rwsnoop -Z       # print zone ID
        rwsnoop -n bash  # monitor processes named "bash"
```

Web Server. Here `rwsnoop` is used to trace all Web server processes named `httpd` (something that PID-based tools such as `truss(1M)` or `strace` cannot do easily):

```
# rwsnoop -tn httpd
TIME              UID    PID CMD          D   BYTES FILE
6854075939432      80 713149 httpd        R     495 <unknown>
6854075944873      80 713149 httpd        R     495 /wiki/includes/WebResponse.php
6854075944905      80 713149 httpd        R       0 /wiki/includes/WebResponse.php
6854075944921      80 713149 httpd        R       0 /wiki/includes/WebResponse.php
6854075946102      80 713149 httpd        W     100 <unknown>
6854075946261      80 713149 httpd        R     303 <unknown>
6854075946592      80 713149 httpd        W       5 <unknown>
6854075959169      80 713149 httpd        W      92 /var/apache2/2.2/logs/access_log
6854076038294      80 713149 httpd        R       0 <unknown>
6854076038390      80 713149 httpd        R      -1 <unknown>
6854206429906      80 713251 httpd        R    4362 /wiki/includes/LinkBatch.php
6854206429933      80 713251 httpd        R       0 /wiki/includes/LinkBatch.php
6854206429952      80 713251 httpd        R       0 /wiki/includes/LinkBatch.php
6854206432875      80 713251 httpd        W      92 <unknown>
6854206433300      80 713251 httpd        R      52 <unknown>
6854206434656      80 713251 httpd        R    6267 /wiki/includes/SiteStats.php
[...]
```

The files that `httpd` is reading can be seen in the output, along with the log file it is writing to. The `<unknown>` file I/O is likely to be the socket I/O for HTTP, because it reads requests and responds to clients.

mmap.d

Although many of the scripts in this chapter examine file system I/O by tracing reads and writes, there is another way to read or write file data: `mmap()`. This system call maps a region of a file to the memory of the user-land process, allowing reads and writes to be performed by reading and writing to that memory segment. The `mmap.d` script traces mmap calls with details including the process name, filename, and flags used with `mmap()`.

Script

This script was written for Oracle Solaris and uses the preprocessor (`-C` on line 1) so that the `sys/mman.h` file can be included (line 3):

```
1     #!/usr/sbin/dtrace -Cs
2
3     #include <sys/mman.h>
4
5     #pragma D option quiet
6     #pragma D option switchrate=10hz
7
8     dtrace:::BEGIN
9     {
10          printf("%6s %-12s %-4s %-8s %-8s %-8s %s\n", "PID",
11              "PROCESS", "PROT", "FLAGS", "OFFS(KB)", "SIZE(KB)", "PATH");
12     }
13
14    syscall::mmap*:entry
15    /fds[arg4].fi_pathname != "<none>"/
```

```
{
        /* see mmap(2) and /usr/include/sys/mman.h */
        printf("%6d %-12.12s %s%s%s  %s%s%s%s%s%s%s%s %-8d %-8d %s\n",
            pid, execname,
            arg2 & PROT_EXEC  ? "E" : "-",  /* pages can be executed */
            arg2 & PROT_WRITE ? "W" : "-",  /* pages can be written */
            arg2 & PROT_READ  ? "R" : "-",  /* pages can be read */
            arg3 & MAP_INITDATA  ? "I" : "-",     /* map data segment */
            arg3 & MAP_TEXT      ? "T" : "-",     /* map code segment */
            arg3 & MAP_ALIGN     ? "L" : "-",     /* addr specifies alignment */
            arg3 & MAP_ANON      ? "A" : "-",     /* map anon pages directly */
            arg3 & MAP_NORESERVE ? "N" : "-",     /* don't reserve swap area */
            arg3 & MAP_FIXED     ? "F" : "-",     /* user assigns address */
            arg3 & MAP_PRIVATE   ? "P" : "-",     /* changes are private */
            arg3 & MAP_SHARED    ? "S" : "-",     /* share changes */
            arg5 / 1024, arg1 / 1024, fds[arg4].fi_pathname);
}
```

Script mmap.d

Example

While tracing, the `cp(1)` was executed to copy a 100MB file called `100m`:

```
solaris# cp /export/fs1/100m /export/fs2
```

The file was read by `cp(1)` by mapping it to memory, 8MB at a time:

```
solaris# mmap.d
   PID PROCESS      PROT FLAGS    OFFS(KB) SIZE(KB) PATH
  2652 cp           E-R  --L---P- 0        32       /lib/libc.so.1
  2652 cp           E-R  -T---FP- 0        1274     /lib/libc.so.1
  2652 cp           EWR  I----FP- 1276     27       /lib/libc.so.1
  2652 cp           E-R  --L---P- 0        32       /lib/libsec.so.1
  2652 cp           E-R  -T---FP- 0        62       /lib/libsec.so.1
  2652 cp           -WR  I----FP- 64       15       /lib/libsec.so.1
  2652 cp           E-R  --L---P- 0        32       /lib/libcmdutils.so.1
  2652 cp           E-R  -T---FP- 0        11       /lib/libcmdutils.so.1
  2652 cp           -WR  I----FP- 12       0        /lib/libcmdutils.so.1
  2652 cp           --R  -------S 0        8192     /export/fs1/100m
  2652 cp           --R  -----F-S 8192     8192     /export/fs1/100m
  2652 cp           --R  -----F-S 16384    8192     /export/fs1/100m
  2652 cp           --R  -----F-S 24576    8192     /export/fs1/100m
  2652 cp           --R  -----F-S 32768    8192     /export/fs1/100m
  2652 cp           --R  -----F-S 40960    8192     /export/fs1/100m
  2652 cp           --R  -----F-S 49152    8192     /export/fs1/100m
  2652 cp           --R  -----F-S 57344    8192     /export/fs1/100m
  2652 cp           --R  -----F-S 65536    8192     /export/fs1/100m
  2652 cp           --R  -----F-S 73728    8192     /export/fs1/100m
  2652 cp           --R  -----F-S 81920    8192     /export/fs1/100m
  2652 cp           --R  -----F-S 90112    8192     /export/fs1/100m
  2652 cp           --R  -----F-S 98304    4096     /export/fs1/100m
^C
```

The output also shows the initialization of the `cp(1)` command because it maps libraries as executable segments.

fserrors.d

Errors can be particularly interesting when troubleshooting system issues, including errors returned by the file system in response to application requests. This script traces all errors at the syscall layer, providing process, path name, and error number information. Many of these errors may be "normal" for the application and handled correctly by the application code. This script merely reports that they happened, not how they were then handled (if they were handled).

Script

This script traces variants of `read()`, `write()`, `open()`, and `stat()`, which are handled a little differently depending on how to retrieve the path information. It can be enhanced to include other file system system calls as desired:

```
#!/usr/sbin/dtrace -s

#pragma D option quiet

dtrace:::BEGIN
{
       trace("Tracing syscall errors... Hit Ctrl-C to end.\n");
}

syscall::read*:entry, syscall::write*:entry { self->fd = arg0; }
syscall::open*:entry, syscall::stat*:entry  { self->ptr = arg0; }

syscall::read*:return, syscall::write*:return
/(int)arg0 < 0 && self->fd > 2/
{
        self->path = fds[self->fd].fi_pathname;
}

syscall::open*:return, syscall::stat*:return
/(int)arg0 < 0 && self->ptr/
{
        self->path = copyinstr(self->ptr);
}

syscall::read*:return, syscall::write*:return,
syscall::open*:return, syscall::stat*:return
/(int)arg0 < 0 && self->path != NULL/
{
        @[execname, probefunc, errno, self->path] = count();
        self->path = 0;
}

syscall::read*:return, syscall::write*:return { self->fd = 0; }
syscall::open*:return, syscall::stat*:return  { self->ptr = 0; }

dtrace:::END
{
        printf("%16s %16s %3s %8s %s\n", "PROCESSES", "SYSCALL", "ERR",
            "COUNT", "PATH");
        printa("%16s %16s %3d %@8d %s\n", @);
}
```

Script fserrors.d

Example

`fserrors.d` was run for one minute on a wiki server (running both TWiki and MediaWiki):

```
# fserrors.d
       PROCESSES          SYSCALL ERR    COUNT PATH
            sshd             open   2        1 /etc/hosts.allow
            sshd             open   2        1 /etc/hosts.deny
[...output truncated...]
            sshd           stat64   2        2 /root/.ssh/authorized_keys
            sshd           stat64   2        2 /root/.ssh/authorized_keys2
          locale             open   2        4 /var/ld/ld.config
            sshd             open   2        5 /var/run/tzsync
            view           stat64   2        7 /usr/local/twiki/data/Main/NFS.txt
            view           stat64   2        8 /usr/local/twiki/data/Main/ARC.txt
            view           stat64   2       11 /usr/local/twiki/data/Main/TCP.txt
            Xorg             read  11       27 <unknown>
            view           stat64   2       32 /usr/local/twiki/data/Main/NOTES.txt
           httpd             read  11       35 <unknown>
            view           stat64   2       85 /usr/local/twiki/data/Main/DRAM.txt
            view           stat64   2      174 /usr/local/twiki/data/Main/ZFS.txt
            view           stat64   2      319 /usr/local/twiki/data/Main/IOPS.txt
```

While tracing, processes with the name `view` attempted to `stat64()` an `IOPS.txt` file 319 times, each time encountering error number 2 (file not found). The `view` program was short-lived and not still running on the system and so was located by using a DTrace one-liner to catch its execution:

```
# dtrace -n 'proc:::exec-success { trace(curpsinfo->pr_psargs); }'
dtrace: description 'proc:::exec-success ' matched 1 probe
CPU     ID                    FUNCTION:NAME
  2  23001     exec_common:exec-success   /usr/bin/perl -wT /usr/local/twiki/bin/view
```

It took a little more investigation to find the reason behind the `stat64()` calls: TWiki automatically detects terms in documentation by searching for words in all capital letters and then checks whether there are pages for those terms. Since TWiki saves everything as text files, it checks by running `stat64()` on the file system for those pages (indirectly, since it is a Perl program). If this sounds suboptimal, use DTrace to measure the CPU time spent calling `stat64()` to quantify this behavior—`stat()` is typically a fast call.

fsinfo Scripts

The fsinfo provider traces file system activity at the VFS layer, allowing all file system activity to be traced within the kernel from one provider. The probes it exports contain mapped file info and byte counts where appropriate. It is currently available only on Solaris; FreeBSD has a similar provider called vfs.

fswho.d

This script uses the fsinfo provider to show which processes are reading and writing to which file systems, in terms of kilobytes.

Script

This is similar to the earlier `sysfs.d` script, but it can match all file system reads and writes without tracing all the syscalls that may be occurring. It can also easily access the size of the reads and writes, provided as `arg1` by the fsinfo provider (which isn't always easy at the syscall provider: Consider `readv()`).

```
#!/usr/sbin/dtrace -s

#pragma D option quiet

dtrace:::BEGIN
{
        printf("Tracing... Hit Ctrl-C to end.\n");
}

fsinfo:::read,
fsinfo:::write
{
        @[execname, probename == "read" ? "R" : "W", args[0]->fi_fs,
            args[0]->fi_mount] = sum(arg1);
}

dtrace:::END
{
        normalize(@, 1024);
        printf("  %-16s  %1s %12s  %-10s %s\n", "PROCESSES", "D", "KBYTES",
            "FS", "MOUNTPOINT");
        printa("  %-16s  %1.1s %@12d  %-10s %s\n", @);
}
```

Script fswho.d

Example

The source code was building on a ZFS share while `fswho.d` was run:

```
# fswho.d
Tracing... Hit Ctrl-C to end.
^C
  PROCESSES         D       KBYTES  FS         MOUNTPOINT
  tail              R            0  zfs        /builds/ahl
  tail              R            0  zfs        /builds/bmc
  sshd              R            0  sockfs     /
  sshd              W            0  sockfs     /
  ssh-socks5-proxy  R            0  sockfs     /
```

```
  sh                W           1  tmpfs      /tmp
  dmake             R           1  nfs4       /home/brendan
[...output truncated...]
  id                R          68  zfs        /var
  cp                R         133  zfs        /builds/brendan
  scp               R         224  nfs4       /net/fw/export/install
  install           R         289  zfs        /
  dmake             R         986  zfs        /
  cp                W        1722  zfs        /builds/brendan
  dmake             W       13357  zfs        /builds/brendan
  dmake             R       21820  zfs        /builds/brendan
```

`fswho.d` has identified that processes named `dmake` read 21MB from the `/builds/brendan` share and wrote back 13MB. Various other process file system activity has also been identified, which includes socket I/O because the kernel implementation serves these via a sockfs file system.

readtype.d

This script shows the type of reads by file system and the amount for comparison, differentiating between logical reads (syscall layer) and physical reads (disk layer). There are a number of reasons why the rate of logical reads will not equal physical reads.

- **Caching**: Logical reads may return from a DRAM cache without needing to be satisfied as physical reads from the storage devices.
- **Read-ahead/prefetch**: The file system may detect a sequential access pattern and request data to prewarm the cache before it has been requested logically. If it is then never requested logically, more physical reads may occur than logical.
- **File system record size**: The file system on-disk structure may store data as addressable blocks of a certain size (record size), and physical reads to storage devices will be in units of this size. This may inflate reads between logical and physical, because they are rounded up to record-sized reads for the physical storage devices.
- **Device sector size**: Despite the file system record size, there may still be a minimum physical read size required by the storage device, such as 512 bytes for common disk drives (sector size).

As an example of file system record size inflation, consider a file system that employs a fixed 4KB record size, while an application is performing random 512-byte reads. Each logical read will be 512 bytes in size, but each physical read will be 4KB—reading an extra 3.5KB that will not be used (or is unlikely to be used, because the workload is random). This makes for an 8x inflation between logical and physical reads.

Script

This script uses the fsinfo provider to trace logical reads and uses the io provider to trace physical reads. It is based on `rfsio.d` from the DTraceToolkit.

```
#!/usr/sbin/dtrace -s

#pragma D option quiet

inline int TOP = 20;
self int trace;
uint64_t lbytes;
uint64_t pbytes;

dtrace:::BEGIN
{
        trace("Tracing... Output every 5 secs, or Ctrl-C.\n");
}

fsinfo:::read
{
        @io[args[0]->fi_mount, "logical"] = count();
        @bytes[args[0]->fi_mount, "logical"] = sum(arg1);
        lbytes += arg1;
}

io:::start
/args[0]->b_flags & B_READ/
{
        @io[args[2]->fi_mount, "physical"] = count();
        @bytes[args[2]->fi_mount, "physical"] = sum(args[0]->b_bcount);
        pbytes += args[0]->b_bcount;
}

profile:::tick-5s,
dtrace:::END
{
        trunc(@io, TOP);
        trunc(@bytes, TOP);
        printf("\n%Y:\n", walltimestamp);
        printf("\n Read I/O (top %d)\n", TOP);
        printa(" %-32s %10s %10@d\n", @io);
        printf("\n Read Bytes (top %d)\n", TOP);
        printa(" %-32s %10s %10@d\n", @bytes);
        printf("\nphysical/logical bytes rate: %d%%\n",
            lbytes ? 100 * pbytes / lbytes : 0);
        trunc(@bytes);
        trunc(@io);
        lbytes = pbytes = 0;
}
```

Script readtype.d

Examples

Examples include uncached file system read and cache file system read.

Uncached File System Read. Here the /usr file system is archived, reading through the files sequentially:

```
# readtype.d
Tracing... Output every 5 secs, or Ctrl-C.

2010 Jun 19 07:42:50:

 Read I/O (top 20)
 /                                    logical          13
 /export/home                         logical          23
 /tmp                                 logical         428
 /usr                                physical        1463
 /usr                                 logical        2993

 Read Bytes (top 20)
 /tmp                                 logical           0
 /                                    logical        1032
 /export/home                         logical       70590
 /usr                                 logical    11569675
 /usr                                physical    11668480

physical/logical bytes rate: 102%
```

The physical/logical throughput rate was 102 percent during this interval. The reasons for the inflation may be because of both sector size (especially when reading any file smaller than 512 bytes) and read-ahead (where tracing has caught the physical but not yet the logical reads).

Cache File System Read. Following on from the previous example, the /usr file system was reread:

```
# readtype.d
Tracing... Output every 5 secs, or Ctrl-C.

2010 Jun 19 07:44:05:

 Read I/O (top 20)
 /                                   physical           5
 /                                    logical          21
 /export/home                         logical          54
 /tmp                                 logical         865
 /usr                                physical        3005
 /usr                                 logical       14029

 Read Bytes (top 20)
 /tmp                                 logical           0
 /                                    logical        1372
 /                                   physical       24576
 /export/home                         logical      166561
 /usr                                physical    16015360
 /usr                                 logical    56982746

physical/logical bytes rate: 27%
```

Now much of data is returning from the cache, with only 27 percent being read from disk. We can see the difference this makes to the application: The first example showed a logical read throughput of 11MB during the five-second interval as the data was read from disk; the logical rate in this example is now 56MB during five seconds.

writetype.d

As a companion to `readtype.d`, this script traces file system writes, allowing types to be compared. Logical writes may differ from physical writes for the following reasons (among others):

- **Asynchronous writes**: The default behavior[1] for many file systems is that logical writes dirty data in DRAM, which is later flushed to disk by an asynchronous thread. This allows the application to continue without waiting for the disk writes to complete. The effect seen in `writetype.d` will be logical writes followed some time later by physical writes.
- **Write canceling**: Data logically written but not yet physically written to disk is logically overwritten, canceling the previous physical write.
- **File system record size**: As described earlier for `readtype.d`.
- **Device sector size**: As described earlier for `readtype.d`.
- **Volume manager**: If software volume management is used, such as applying levels of RAID, writes may be inflated depending on the RAID configuration. For example, software mirroring will cause logical writes to be doubled when they become physical.

Script

This script is identical to `readtype.d` except for the following lines:

```
15  fsinfo:::write

22  io:::start
23  /!(args[0]->b_flags & B_READ)/

36          printf("\n Write I/O (top %d)\n", TOP);

38          printf("\n Write Bytes (top %d)\n", TOP);
```

Now fsinfo is tracing writes, and the `io:::start` predicate also matches writes.

1. For times when the application requires the data to be written on stable storage before continuing, `open()` flags such as `O_SYNC` and `O_DSYNC` can be used to inform the file system to write immediately to stable storage.

Example

The `writetype.d` script was run for ten seconds. During the first five seconds, an application wrote data to the file system:

```
# writetype.d
Tracing... Output every 5 secs, or Ctrl-C.

2010 Jun 19 07:59:10:

 Write I/O (top 20)
 /var                                 logical          1
 /                                    logical          3
 /export/ufs1                         logical          9
 /export/ufs1                        physical        696

 Write bytes (top 20)
 /                                    logical        208
 /var                                 logical        704
 /export/ufs1                        physical    2587648
 /export/ufs1                         logical    9437184

physical/logical throughput rate: 24%

2010 Jun 19 07:59:15:

 Write I/O (top 20)
 /                                    logical          2
 /export/ufs1                        physical        238

 Write bytes (top 20)
 /                                    logical        752
 /export/ufs1                        physical    7720960

physical/logical throughput rate: 805%
```

In the first five-second summary, more logical bytes were written than physical, because writes were buffered in the file system cache but not yet flushed to disk. The second output shows those writes finishing being flushed to disk.

fssnoop.d

This script traces all file system activity by printing every event from the fsinfo provider with user, process, and size information, as well as path information if available. It also prints all the event data line by line, without trying to summarize it into reports, making the output suitable for other postprocessing if desired. The section that follows demonstrates rewriting this script for other providers and operating systems.

Script

Since this traces all file system activity, it may catch sockfs activity and create a feedback loop where the DTrace output to the file system or your remote network session is traced. To work around this, it accepts an optional argument of the process name to trace and excludes dtrace processes by default (line 14). For more sophisticated arguments, the script could be wrapped in the shell like `rwsnoop` so that `getopts` can be used.

```
1   #!/usr/sbin/dtrace -s
2
3   #pragma D option quiet
4   #pragma D option defaultargs
5   #pragma D option switchrate=10hz
6
7   dtrace:::BEGIN
8   {
9          printf("%-12s %6s %6s %-12.12s %-12s %-6s %s\n", "TIME(ms)", "UID",
10             "PID", "PROCESS", "CALL", "BYTES", "PATH");
11  }
12
13  fsinfo:::
14  /execname != "dtrace" && ($$1 == NULL || $$1 == execname)/
15  {
16          printf("%-12d %6d %6d %-12.12s %-12s %-6d %s\n", timestamp / 1000000,
17             uid, pid, execname, probename, arg1, args[0]->fi_pathname);
18  }
```

Script fssnoop.d

So that the string argument `$$1` could be optional, line 4 sets the `defaultargs` option, which sets `$$1` to `NULL` if it wasn't provided at the command line. Without `defaultargs`, DTrace would error unless an argument is provided.

Examples

The default output prints all activity:

```
# fssnoop.d
TIME(ms)        UID    PID PROCESS      CALL         BYTES  PATH
924434524         0   2687 sshd         poll         0      <unknown>
924434524         0   2687 sshd         rwlock       0      <unknown>
924434524         0   2687 sshd         write        112    <unknown>
924434524         0   2687 sshd         rwunlock     0      <unknown>
[...]
```

Since it was run over an SSH session, it sees its own socket writes to sockfs by the sshd process. An output file can be specified to prevent this:

```
# fssnoop.d -o out.log
# cat out.log
TIME(ms)        UID    PID PROCESS      CALL        BYTES  PATH
924667432         0   7108 svcs         lookup      0      /usr/share/lib/zoneinfo
924667432         0   7108 svcs         lookup      0      /usr/share/lib/zoneinfo/UTC
```

```
924667432        0   7108 svcs         getattr     0      /usr/share/lib/zoneinfo/UTC
924667432        0   7108 svcs         access      0      /usr/share/lib/zoneinfo/UTC
924667432        0   7108 svcs         open        0      /usr/share/lib/zoneinfo/UTC
924667432        0   7108 svcs         getattr     0      /usr/share/lib/zoneinfo/UTC
924667432        0   7108 svcs         rwlock      0      /usr/share/lib/zoneinfo/UTC
924667432        0   7108 svcs         read        56     /usr/share/lib/zoneinfo/UTC
924667432        0   7108 svcs         rwunlock    0      /usr/share/lib/zoneinfo/UTC
924667432        0   7108 svcs         close       0      /usr/share/lib/zoneinfo/UTC
[...]
```

This has caught the execution of the Oracle Solaris `svcs(1)` command, which was listing system services. The UTC file was read in this way 204 times (the output was many pages long), which is twice for every line of output that `svcs(1)` printed, which included a date.

To filter on a particular process name, you can provided as an argument. Here, the file system calls from the `ls(1)` command were traced:

```
# fssnoop.d ls
TIME(ms)         UID    PID PROCESS       CALL         BYTES  PATH
924727221          0   7111 ls            rwlock       0      /tmp
924727221          0   7111 ls            readdir      1416   /tmp
924727221          0   7111 ls            rwunlock     0      /tmp
924727221          0   7111 ls            rwlock       0      /tmp
[...]
```

VFS Scripts

VFS is the Virtual File System, a kernel interface that allows different file systems to integrate into the same kernel code. It provides an abstraction of a file system with the common calls: read, write, open, close, and so on. Interfaces and abstractions can make good targets for DTracing, since they are often documented and relatively stable (compared to the implementation code).

The fsinfo provider for Solaris traces at the VFS level, as shown by the scripts in the previous "fsinfo" section. FreeBSD has the vfs provider for this purpose, demonstrated in this section. When neither vfs or fsinfo is available, VFS can be traced using the fbt[2] provider. fbt is an unstable interface: It exports kernel functions and data structures that may change from release to release. The following scripts were based on OpenSolaris circa December 2009 and on Mac OS X version 10.6, and they may not work on other releases without changes. Even if these scripts no longer execute, they can still be treated as examples of D programming and for the sort of data that DTrace can make available for VFS analysis.

2. See the "fbt Provider" section in Chapter 12 for more discussion about use of the fbt provider.

To demonstrate the different ways VFS can be traced and to allow these to be compared, the `fssnoop.d` script has been written in four ways:

- `fssnoop.d`: fsinfo provider based (OpenSolaris), shown previously
- `solvfssnoop.d`: fbt provider based (Solaris)
- `macvfssnoop.d`: fbt provider based (Mac OS X)
- `vfssnoop.d`: vfs provider based (FreeBSD)

Because these scripts trace common VFS events, they can be used as starting points for developing other scripts. This section also includes three examples that trace file creation and deletion on the different operating systems (`sollife.d`, `maclife.d`, and `vfslife.d`).

Note that VFS can cover more than just on-disk file systems; whichever kernel modules use the VFS abstraction may also be traced by these scripts, including terminal output (writes to `/dev/pts` or `dev/tty` device files).

solvfssnoop.d

To trace VFS calls in the Oracle Solaris kernel, the fop interface can be traced using the fbt provider. (This is also the location that the fsinfo provider instruments.) Here's an example of listing `fop` probes:

```
solaris# dtrace -ln 'fbt::fop_*:entry'
   ID   PROVIDER            MODULE                          FUNCTION NAME
36831        fbt           genunix                      fop_inactive entry
38019        fbt           genunix                        fop_addmap entry
38023        fbt           genunix                        fop_access entry
38150        fbt           genunix                        fop_create entry
38162        fbt           genunix                        fop_delmap entry
38318        fbt           genunix                        fop_frlock entry
38538        fbt           genunix                        fop_lookup entry
38646        fbt           genunix                         fop_close entry
[...output truncated...]
```

The function names include the names of the VFS calls. Although the fbt provider is considered an unstable interface, tracing kernel interfaces such as `fop` is expected to be the safest use of fbt possible—`fop` doesn't change much (but be aware that it can and has).

Script

This script traces many of the common VFS calls at the Oracle Solaris `fop` interface, including `read()`, `write()` and `open()`. See `/usr/include/sys/vnode.h` for the full list. Additional calls can be added to `solvfssnoop.d` as desired.

```
1     #!/usr/sbin/dtrace -s
2
3     #pragma D option quiet
4     #pragma D option defaultargs
5     #pragma D option switchrate=10hz
6
7     dtrace:::BEGIN
8     {
9           printf("%-12s %6s %6s %-12.12s %-12s %-4s %s\n", "TIME(ms)", "UID",
10              "PID", "PROCESS", "CALL", "KB", "PATH");
11    }
12
13    /* see /usr/include/sys/vnode.h */
14
15    fbt::fop_read:entry, fbt::fop_write:entry
16    {
17          self->path = args[0]->v_path;
18          self->kb = args[1]->uio_resid / 1024;
19    }
20
21    fbt::fop_open:entry
22    {
23          self->path = (*args[0])->v_path;
24          self->kb = 0;
25    }
26
27    fbt::fop_close:entry, fbt::fop_ioctl:entry, fbt::fop_getattr:entry,
28    fbt::fop_readdir:entry
29    {
30          self->path = args[0]->v_path;
31          self->kb = 0;
32    }
33
34    fbt::fop_read:entry, fbt::fop_write:entry, fbt::fop_open:entry,
35    fbt::fop_close:entry, fbt::fop_ioctl:entry, fbt::fop_getattr:entry,
36    fbt::fop_readdir:entry
37    /execname != "dtrace" && ($$1 == NULL || $$1 == execname)/
38    {
39          printf("%-12d %6d %6d %-12.12s %-12s %-4d %s\n", timestamp / 1000000,
40              uid, pid, execname, probefunc, self->kb,
41              self->path != NULL ? stringof(self->path) : "<null>");
42    }
43
44    fbt::fop_read:entry, fbt::fop_write:entry, fbt::fop_open:entry,
45    fbt::fop_close:entry, fbt::fop_ioctl:entry, fbt::fop_getattr:entry,
46    fbt::fop_readdir:entry
47    {
48          self->path = 0; self->kb = 0;
49    }
```

Script solvfssnoop.d

Lines 15 to 32 probe different functions and populate the `self->path` and `self->kb` variables so that they are printed out in a common block of code on lines 39 to 41.

Example

As with `fssnoop.d`, this script accepts an optional argument for the process name to trace. Here's an example of tracing `ls -l`:

```
solaris# solvfssnoop.d ls
TIME(ms)       UID    PID PROCESS     CALL         KB   PATH
2499844          0   1152 ls          fop_close    0    /var/run/name_service_door
2499844          0   1152 ls          fop_close    0    <null>
2499844          0   1152 ls          fop_close    0    /dev/pts/2
2499844          0   1152 ls          fop_getattr  0    /usr/bin/ls
2499844          0   1152 ls          fop_getattr  0    /lib/libc.so.1
2499844          0   1152 ls          fop_getattr  0    /usr/lib/libc/libc_hwcap1.so.1
2499844          0   1152 ls          fop_getattr  0    /lib/libc.so.1
2499844          0   1152 ls          fop_getattr  0    /usr/lib/libc/libc_hwcap1.so.1
[...]
2499851          0   1152 ls          fop_getattr  0    /var/tmp
2499851          0   1152 ls          fop_open     0    /var/tmp
2499851          0   1152 ls          fop_getattr  0    /var/tmp
2499852          0   1152 ls          fop_readdir  0    /var/tmp
2499852          0   1152 ls          fop_getattr  0    /var/tmp/ExrUaWjc
[...]
2500015          0   1152 ls          fop_open     0    /etc/passwd
2500015          0   1152 ls          fop_getattr  0    /etc/passwd
2500015          0   1152 ls          fop_getattr  0    /etc/passwd
2500015          0   1152 ls          fop_getattr  0    /etc/passwd
2500015          0   1152 ls          fop_ioctl    0    /etc/passwd
2500015          0   1152 ls          fop_read     1    /etc/passwd
2500016          0   1152 ls          fop_getattr  0    /etc/passwd
2500016          0   1152 ls          fop_close    0    /etc/passwd
[...]
```

The output has been truncated to highlight three stages of `ls` that can be seen in the VFS calls: command initialization, reading the directory, and reading system databases.

macvfssnoop.d

To trace VFS calls in the Mac OS X kernel, the VNOP interface can be traced using the fbt provider. Here's an example of listing VNOP probes:

```
macosx# dtrace -ln 'fbt::VNOP_*:entry'
ID   PROVIDER            MODULE                          FUNCTION NAME
  705        fbt       mach_kernel                         VNOP_ACCESS entry
  707        fbt       mach_kernel                        VNOP_ADVLOCK entry
  709        fbt       mach_kernel                       VNOP_ALLOCATE entry
  711        fbt       mach_kernel                       VNOP_BLKTOOFF entry
  713        fbt       mach_kernel                       VNOP_BLOCKMAP entry
  715        fbt       mach_kernel                         VNOP_BWRITE entry
  717        fbt       mach_kernel                          VNOP_CLOSE entry
  719        fbt       mach_kernel                       VNOP_COPYFILE entry
  721        fbt       mach_kernel                         VNOP_CREATE entry
  723        fbt       mach_kernel                       VNOP_EXCHANGE entry
  725        fbt       mach_kernel                          VNOP_FSYNC entry
  727        fbt       mach_kernel                        VNOP_GETATTR entry
[...output truncated...]
```

The kernel source can be inspected to determine the arguments to these calls.

Script

This script traces many of the common VFS calls at the Darwin VNOP interface, including `read()`, `write()`, and `open()`. See `sys/bsd/sys/vnode_if.h` from the source for the full list. Additional calls can be added as desired.

```
#!/usr/sbin/dtrace -s

#pragma D option quiet
#pragma D option defaultargs
#pragma D option switchrate=10hz

dtrace:::BEGIN
{
      printf("%-12s %6s %6s %-12.12s %-12s %-4s %s\n", "TIME(ms)", "UID",
          "PID", "PROCESS", "CALL", "KB", "PATH");
}

/* see sys/bsd/sys/vnode_if.h */

fbt::VNOP_READ:entry, fbt::VNOP_WRITE:entry
{
      self->path = ((struct vnode *)arg0)->v_name;
      self->kb = ((struct uio *)arg1)->uio_resid_64 / 1024;
}

fbt::VNOP_OPEN:entry
{
      self->path = ((struct vnode *)arg0)->v_name;
      self->kb = 0;
}

fbt::VNOP_CLOSE:entry, fbt::VNOP_IOCTL:entry, fbt::VNOP_GETATTR:entry,
fbt::VNOP_READDIR:entry
{
      self->path = ((struct vnode *)arg0)->v_name;
      self->kb = 0;
}

fbt::VNOP_READ:entry, fbt::VNOP_WRITE:entry, fbt::VNOP_OPEN:entry,
fbt::VNOP_CLOSE:entry, fbt::VNOP_IOCTL:entry, fbt::VNOP_GETATTR:entry,
fbt::VNOP_READDIR:entry
/execname != "dtrace" && ($$1 == NULL || $$1 == execname)/
{
      printf("%-12d %6d %6d %-12.12s %-12s %-4d %s\n", timestamp / 1000000,
          uid, pid, execname, probefunc, self->kb,
          self->path != NULL ? stringof(self->path) : "<null>");
}

fbt::VNOP_READ:entry, fbt::VNOP_WRITE:entry, fbt::VNOP_OPEN:entry,
fbt::VNOP_CLOSE:entry, fbt::VNOP_IOCTL:entry, fbt::VNOP_GETATTR:entry,
fbt::VNOP_READDIR:entry
{
      self->path = 0; self->kb = 0;
}
```

Script macvfssnoop.d

Example

An `ls -l` command was traced to compare with the other VFS script examples:

```
macosx# macvfssnoop.d ls
TIME(ms)         UID    PID PROCESS      CALL          KB   PATH
1183135202       501  57611 ls           VNOP_GETATTR  0    urandom
1183135202       501  57611 ls           VNOP_OPEN     0    urandom
1183135202       501  57611 ls           VNOP_READ     0    urandom
1183135202       501  57611 ls           VNOP_CLOSE    0    urandom
1183135202       501  57611 ls           VNOP_GETATTR  0    libncurses.5.4.dylib
1183135202       501  57611 ls           VNOP_GETATTR  0    libSystem.B.dylib
1183135202       501  57611 ls           VNOP_GETATTR  0    libSystem.B.dylib
1183135202       501  57611 ls           VNOP_GETATTR  0    libmathCommon.A.dylib
1183135203       501  57611 ls           VNOP_GETATTR  0    libmathCommon.A.dylib
[…]
1183135221       501  57611 ls           VNOP_GETATTR  0    fswho
1183135221       501  57611 ls           VNOP_GETATTR  0    macvfssnoop.d
1183135221       501  57611 ls           VNOP_GETATTR  0    macvfssnoop.d
1183135221       501  57611 ls           VNOP_GETATTR  0    new
1183135221       501  57611 ls           VNOP_GETATTR  0    oneliners
[…]
1183135225       501  57611 ls           VNOP_GETATTR  0    fswho
1183135225       501  57611 ls           VNOP_WRITE    0    ttys003
1183135225       501  57611 ls           VNOP_GETATTR  0    macvfssnoop.d
1183135225       501  57611 ls           VNOP_GETATTR  0    macvfssnoop.d
1183135225       501  57611 ls           VNOP_WRITE    0    ttys003
[…]
```

The VFS calls show three stages to `ls` on Mac OS X: command initialization, an initial check of the files, and then a second pass as output is written to the screen (`ttys003`).

vfssnoop.d

FreeBSD has the VOP interface for VFS, which is similar to the VNOP interface on Mac OS X (as traced by `macvfssnoop.d`). Instead of tracing VOP via the fbt provider, this script demonstrates the FreeBSD vfs provider.[3] Here's an example listing vfs probes:

```
freebsd# dtrace -ln vfs:::
   ID   PROVIDER            MODULE                          FUNCTION NAME
38030        vfs         namecache                      zap_negative done
38031        vfs         namecache                               zap done
38032        vfs         namecache                          purgevfs done
38033        vfs         namecache                    purge_negative done
38034        vfs         namecache                             purge done
38035        vfs         namecache                            lookup miss
38036        vfs         namecache                            lookup hit_negative
38037        vfs         namecache                            lookup hit
38038        vfs         namecache                          fullpath return
```

3. This was written by Robert Watson.

```
38039       vfs         namecache                          fullpath miss
38040       vfs         namecache                          fullpath hit
38041       vfs         namecache                          fullpath entry
38042       vfs         namecache                    enter_negative done
38043       vfs         namecache                             enter done
38044       vfs             namei                            lookup return
38045       vfs             namei                            lookup entry
38046       vfs                                                stat reg
38047       vfs                                                stat mode
38048       vfs               vop                       vop_vptocnp return
38049       vfs               vop                       vop_vptocnp entry
38050       vfs               vop                        vop_vptofh return
38051       vfs               vop                        vop_vptofh entry
[...]
```

Four different types of probes are shown in this output:

- `vfs:namecache::`: Name cache operations, including lookups (hit/miss)
- `vfs:namei::`: Filename to vnode lookups
- `vfs::stat::` Stat calls
- `vfs:vop::`: VFS operations

The `vfssnoop.d` script demonstrates three of these (`namecache`, `namei`, and `vop`).

Script

The `vfs:vop::` probes traces VFS calls on vnodes, which this script converts into path names or filenames for printing. On FreeBSD, vnodes don't contain a cached path name and may not contain a filename either unless it's in the `(struct namecache *) v_cache_dd` member. There are a few ways to tackle this; here, vnode to path or filename mappings are cached during `namei()` calls and `namecache` hits, both of which can also be traced from the vfs provider:

```
#!/usr/sbin/dtrace -s

#pragma D option quiet
#pragma D option defaultargs
#pragma D option switchrate=10hz
#pragma D option dynvarsize=4m

dtrace:::BEGIN
{
        printf("%-12s %6s %6s %-12.12s %-12s %-4s %s\n", "TIME(ms)", "UID",
            "PID", "PROCESS", "CALL", "KB", "PATH/FILE");
}

/*
 * Populate Vnode2Path from namecache hits
 */
vfs:namecache:lookup:hit
/V2P[arg2] == NULL/
```

continues

```
{
        V2P[arg2] = stringof(arg1);
}

/*
 * (Re)populate Vnode2Path from successful namei() lookups
 */
vfs:namei:lookup:entry
{
        self->buf = arg1;
}
vfs:namei:lookup:return
/self->buf != NULL && arg0 == 0/
{
        V2P[arg1] = stringof(self->buf);
}
vfs:namei:lookup:return
{
        self->buf = 0;
}

/*
 * Trace and print VFS calls
 */
vfs::vop_read:entry, vfs::vop_write:entry
{
        self->path = V2P[arg0];
        self->kb = args[1]->a_uio->uio_resid / 1024;
}

vfs::vop_open:entry, vfs::vop_close:entry, vfs::vop_ioctl:entry,
vfs::vop_getattr:entry, vfs::vop_readdir:entry
{
        self->path = V2P[arg0];
        self->kb = 0;
}

vfs::vop_read:entry, vfs::vop_write:entry, vfs::vop_open:entry,
vfs::vop_close:entry, vfs::vop_ioctl:entry, vfs::vop_getattr:entry,
vfs::vop_readdir:entry
/execname != "dtrace" && ($$1 == NULL || $$1 == execname)/
{
        printf("%-12d %6d %6d %-12.12s %-12s %-4d %s\n", timestamp / 1000000,
            uid, pid, execname, probefunc, self->kb,
            self->path != NULL ? self->path : "<unknown>");
}

vfs::vop_read:entry, vfs::vop_write:entry, vfs::vop_open:entry,
vfs::vop_close:entry, vfs::vop_ioctl:entry, vfs::vop_getattr:entry,
vfs::vop_readdir:entry
{
        self->path = 0; self->kb = 0;
}

/*
 * Tidy V2P, otherwise it gets too big (dynvardrops)
 */
vfs:namecache:purge:done,
vfs::vop_close:entry
{
        V2P[arg0] = 0;
}
```

Script vfssnoop.d

The V2P array can get large, and frequent probes events may cause dynamic variable drops. To reduce these drops, the V2P array is trimmed in lines 76 to 80, and the `dynvarsize` tunable is increased on line 6 (but may need to be set higher, depending on your workload).

Example

An `ls -l` command was traced to compare with the other VFS script examples:

```
freebsd# vfssnoop.d ls
TIME(ms)         UID    PID PROCESS      CALL         KB   PATH/FILE
167135998          0  29717 ls           vop_close    0    /bin/ls
167135999          0  29717 ls           vop_open     0    /var/run/ld-elf.so.hints
167135999          0  29717 ls           vop_read     0    /var/run/ld-elf.so.hints
167136000          0  29717 ls           vop_read     0    /var/run/ld-elf.so.hints
167136000          0  29717 ls           vop_close    0    /var/run/ld-elf.so.hints
167136000          0  29717 ls           vop_open     0    /lib/libutil.so.8
[...]
167136007          0  29717 ls           vop_getattr  0    .history
167136007          0  29717 ls           vop_getattr  1    .bash_history
167136008          0  29717 ls           vop_getattr  0    .ssh
167136008          0  29717 ls           vop_getattr  0    namecache.d
167136008          0  29717 ls           vop_getattr  0    vfssnoop.d
[...]
167136011          0  29717 ls           vop_read     0    /etc/spwd.db
167136011          0  29717 ls           vop_getattr  0    /etc/nsswitch.conf
167136011          0  29717 ls           vop_getattr  0    /etc/nsswitch.conf
167136011          0  29717 ls           vop_read     4    /etc/spwd.db
167136011          0  29717 ls           vop_getattr  0    /etc/nsswitch.conf
167136011          0  29717 ls           vop_open     0    /etc/group
[...]
```

The three stages of `ls` shown here are similar to those seen on Oracle Solaris: command initialization, reading the directory, and reading system databases. In some cases, `vfssnoop.d` is able to print full path names; in others, it prints only the filename.

sollife.d

This script shows file creation and deletion events only. It's able to identify file system churn—the rapid creation and deletion of temporary files. Like `solfssnoop.d`, it traces VFS calls using the fbt provider.

Script

This is a reduced version of `solfssnoop.d`, which traces only the `create()` and `remove()` events:

```
#!/usr/sbin/dtrace -s

#pragma D option quiet
#pragma D option switchrate=10hz

dtrace:::BEGIN
{
      printf("%-12s %6s %6s %-12.12s %-12s %s\n", "TIME(ms)", "UID",
          "PID", "PROCESS", "CALL", "PATH");
}

/* see /usr/include/sys/vnode.h */

fbt::fop_create:entry,
fbt::fop_remove:entry
{
      printf("%-12d %6d %6d %-12.12s %-12s %s/%s\n",
          timestamp / 1000000, uid, pid, execname, probefunc,
          args[0]->v_path != NULL ? stringof(args[0]->v_path) : "<null>",
          stringof(arg1));
}
```

Script sollife.d

Example

Here the script has caught the events from the `vim(1)` text editor, which opened the script in a different terminal window, made a change, and then saved and quit:

```
# sollife.d
TIME(ms)        UID    PID PROCESS      CALL         PATH
1426193948   130948 112454 vim          fop_create   /home/brendan/.sollife.d.swp
1426193953   130948 112454 vim          fop_create   /home/brendan/.sollife.d.swx
1426193956   130948 112454 vim          fop_remove   /home/brendan/.sollife.d.swx
1426193958   130948 112454 vim          fop_remove   /home/brendan/.sollife.d.swp
1426193961   130948 112454 vim          fop_create   /home/brendan/.sollife.d.swp
1426205215   130948 112454 vim          fop_create   /home/brendan/4913
1426205230   130948 112454 vim          fop_remove   /home/brendan/4913
1426205235   130948 112454 vim          fop_create   /home/brendan/sollife.d
1426205244   130948 112454 vim          fop_remove   /home/brendan/sollife.d~
1426205246   130948 112454 vim          fop_create   /home/brendan/.viminfz.tmp
1426205256   130948 112454 vim          fop_remove   /home/brendan/.viminfo
1426205262   130948 112454 vim          fop_remove   /home/brendan/.sollife.d.swp
```

The output shows the temporary swap files created and then removed by `vim`. This script could be enhanced to trace `rename()` events as well, which may better explain how `vim` is managing these files.

maclife.d

This is the `sollife.d` script, written for Mac OS X. As with `macvfssnoop.d`, it uses the fbt provider to trace VNOP interface calls:

```
#!/usr/sbin/dtrace -s

#pragma D option quiet
#pragma D option switchrate=10hz

dtrace:::BEGIN
{
      printf("%-12s %6s %6s %-12.12s %-12s %s\n", "TIME(ms)", "UID",
          "PID", "PROCESS", "CALL", "DIR/FILE");
}

/* see sys/bsd/sys/vnode_if.h */

fbt::VNOP_CREATE:entry,
fbt::VNOP_REMOVE:entry
{
      this->path = ((struct vnode *)arg0)->v_name;
      this->name = ((struct componentname *)arg2)->cn_nameptr;
      printf("%-12d %6d %6d %-12.12s %-12s %s/%s\n",
          timestamp / 1000000, uid, pid, execname, probefunc,
          this->path != NULL ? stringof(this->path) : "<null>",
          stringof(this->name));
}
```

Script maclife.d

vfslife.d

This is the `sollife.d` script, written for FreeBSD. As with `vfssnoop.d`, it uses the vfs provider. This time it attempts to retrieve a directory name from the directory vnode namecache entry (`v_cache_dd`), instead of using DTrace to cache vnode to path translations.

```
#!/usr/sbin/dtrace -s

#pragma D option quiet
#pragma D option switchrate=10hz

dtrace:::BEGIN
{
      printf("%-12s %6s %6s %-12.12s %-12s %s\n", "TIME(ms)", "UID",
          "PID", "PROCESS", "CALL", "DIR/FILE");
}

/* see sys/bsd/sys/vnode_if.h */

vfs::vop_create:entry,
vfs::vop_remove:entry
{
      this->dir = args[0]->v_cache_dd != NULL ?
          stringof(args[0]->v_cache_dd->nc_name) : "<null>";
      this->name = args[1]->a_cnp->cn_nameptr != NULL ?
          stringof(args[1]->a_cnp->cn_nameptr) : "<null>";

      printf("%-12d %6d %6d %-12.12s %-12s %s/%s\n",
          timestamp / 1000000, uid, pid, execname, probefunc,
          this->dir, this->name);
}
```

Script vfslife.d

dnlcps.d

The Directory Name Lookup Cache is a Solaris kernel facility used to cache path names to vnodes. This script shows its hit rate by process, which can be poor when path names are used that are too long for the DNLC. A similar script can be written for the other operating systems; FreeBSD has the `vfs:namecache:lookup:` probes for this purpose.

Script

```
#!/usr/bin/sbin/dtrace -s
[...]
#pragma D option quiet

dtrace:::BEGIN
{
        printf("Tracing... Hit Ctrl-C to end.\n");
}

fbt::dnlc_lookup:return
{
        this->code = arg1 == 0 ? 0 : 1;
        @Result[execname, pid] = lquantize(this->code, 0, 1, 1);
}

dtrace:::END
{
        printa(" CMD: %-16s PID: %d\n%@d\n", @Result);
}
```

Script dnlcps.d

Example

The DNLC lookup result is shown in a distribution plot for visual comparison. Here, a `tar(1)` command had a high hit rate (hit == 1) compared to misses.

```
# dnlcps.d
Tracing... Hit Ctrl-C to end.
^C
[...]

 CMD: tar              PID: 7491

          value  ------------- Distribution ------------- count
            < 0 |                                         0
              0 |@@                                       273
           >= 1 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@   6777
```

See Also

For more examples of DNLC tracing using DTrace, the DTraceToolkit has `dnlcstat` and `dnlcsnoop`, the latter printing DNLC lookup events as they occur; for example:

```
# dnlcsnoop.d
   PID CMD          TIME HIT PATH
  9185 bash            9   Y /etc
  9185 bash            3   Y /etc
 12293 bash            9   Y /usr
 12293 bash            3   Y /usr/bin
 12293 bash            4   Y /usr/bin/find
 12293 bash            7   Y /lib
 12293 bash            3   Y /lib/ld.so.1
 12293 find            6   Y /usr
 12293 find            3   Y /usr/bin
 12293 find            3   Y /usr/bin/find
[...]
```

fsflush_cpu.d

`fsflush` is the kernel file system flush thread on Oracle Solaris, which scans memory periodically for dirty data (data written to DRAM but not yet written to stable storage devices) and issues device writes to send it to disk. This thread applies to different file systems including UFS but does not apply to ZFS, which has its own way of flushing written data (transaction group sync).

Since system memory had become large (from megabytes to gigabytes since fsflush was written), the CPU time for `fsflush` to scan memory had become a performance issue that needed observability; at the time, DTrace didn't exist, and this was solved by adding a virtual process to `/proc` with the name `fsflush` that could be examined using standard process-monitoring tools (`ps(1)`, `prstat(1M)`):

```
solaris# ps -ecf | grep fsflush
    root     3      0  SYS  60   Nov 14 ?        1103:59 fsflush
```

Note the `SYS` scheduling class, identifying that this is a kernel thread.

The `fsflush_cpu.d` script prints `fsflush` information including the CPU time using DTrace.

Script

This script uses the fbt provider to trace the `fsflush_do_pages()` function and its logical calls to write data using `fop_putpage()`. The io provider is also used to measure physical device I/O triggered by `fsflush`.

```
#!/usr/sbin/dtrace -s

#pragma D option quiet

dtrace:::BEGIN
{
        trace("Tracing fsflush...\n");
```

continues

```
8           @fopbytes = sum(0); @iobytes = sum(0);
9   }
10
11  fbt::fsflush_do_pages:entry
12  {
13          self->vstart = vtimestamp;
14  }
15
16  fbt::fop_putpage:entry
17  /self->vstart/
18  {
19          @fopbytes = sum(arg2);
20  }
21
22  io:::start
23  /self->vstart/
24  {
25          @iobytes = sum(args[0]->b_bcount);
26          @ionum = count();
27  }
28
29  fbt::fsflush_do_pages:return
30  /self->vstart/
31  {
32          normalize(@fopbytes, 1024);
33          normalize(@iobytes, 1024);
34          this->delta = (vtimestamp - self->vstart) / 1000000;
35          printf("%Y %4d ms, ", walltimestamp, this->delta);
36          printa("fop: %7@d KB, ", @fopbytes);
37          printa("device: %7@d KB ", @iobytes);
38          printa("%5@d I/O", @ionum);
39          printf("\n");
40          self->vstart = 0;
41          clear(@fopbytes); clear(@iobytes); clear(@ionum);
42  }
```

Script fsflush_cpu.d

Script subtleties include the following.

- Lines 19, 25, and 26 use aggregations instead of global variables, for reliability on multi-CPU environments.
- Lines 36 to 38 print aggregations in separate `printa()` statements instead of a single statement, so this worked on the earliest versions of DTrace on Oracle Solaris, when support for multiple aggregations in a single `printa()` did not yet exist.
- Line 8 and using `clear()` instead of `trunc()` on line 41 are intended to ensure that the aggregations will be printed. Without them, if an aggregation contains no data, the `printa()` statement will be skipped, and the output line will miss elements.
- Since only `fsflush_do_pages()` is traced, only the flushing of pages is considered in the CPU time reported, not the flushing of inodes (the script could be enhanced to trace that as well).

Example

A line is printed for each `fsflush` run, showing the CPU time spent in `fsflush`, the amount of logical data written via the `fop` interface, and the number of physical data writes issued to the storage devices including the physical I/O count:

```
# fsflush_cpu.d
Tracing fsflush...
2010 Jun 20 04:15:52   24 ms, fop:     228 KB, device:     216 KB    54 I/O
2010 Jun 20 04:15:53   26 ms, fop:     260 KB, device:     244 KB    61 I/O
2010 Jun 20 04:15:54   35 ms, fop:    1052 KB, device:    1044 KB   261 I/O
2010 Jun 20 04:15:56   52 ms, fop:    1548 KB, device:    1532 KB   383 I/O
2010 Jun 20 04:15:57   60 ms, fop:    2756 KB, device:    2740 KB   685 I/O
2010 Jun 20 04:15:58   41 ms, fop:    1484 KB, device:    1480 KB   370 I/O
2010 Jun 20 04:15:59   37 ms, fop:    1284 KB, device:    1272 KB   318 I/O
2010 Jun 20 04:16:00   38 ms, fop:     644 KB, device:     632 KB   157 I/O
[...]
```

To demonstrate this, we needed dirty data for `fsflush` to write out. We did this by writing data to a UFS file system, performing a random 4KB write workload to a large file.

We found that applying a sequential write workload did not leave dirty data for `fsflush` to pick up, meaning that the writes to disk were occurring via a different code path. That different code path can be identified using DTrace, by looking at the stack backtraces when disk writes are being issued:

```
# dtrace -n 'io:::start /!(args[0]->b_flags & B_READ)/ { @[stack()] = count(); }'
dtrace: description 'io:::start ' matched 6 probes
^C
[...]
              ufs`lufs_write_strategy+0x100
              ufs`ufs_putapage+0x439
              ufs`ufs_putpages+0x308
              ufs`ufs_putpage+0x82
              genunix`fop_putpage+0x28
              genunix`segmap_release+0x24f
              ufs`wrip+0x4b5
              ufs`ufs_write+0x211
              genunix`fop_write+0x31
              genunix`write+0x287
              genunix`write32+0xe
              unix`sys_syscall32+0x101
             3201
```

So, `fop_putpage()` is happening directly from the `ufs_write()`, rather than `fsflush`.

fsflush.d

The previous script (`fsflush_cpu.d`) was an example of using DTrace to create statistics of interest. This is an example of retrieving existing kernel statistics—if they are available—and printing them out. It was written by Jon Haslam[4] and published in *Solaris Internals* (McDougall and Mauro, 2006).

Statistics are maintained in the kernel to count `fsflush` pages scanned, modified pages found, run time (CPU time), and more.

```
usr/src/uts/common/fs/fsflush.c:
/*
 * some statistics for fsflush_do_pages
 */
typedef struct {
        ulong_t fsf_scan;       /* number of pages scanned */
        ulong_t fsf_examined;   /* number of page_t's actually examined, can */
                                /* be less than fsf_scan due to large pages */
        ulong_t fsf_locked;     /* pages we actually page_lock()ed */
        ulong_t fsf_modified;   /* number of modified pages found */
        ulong_t fsf_coalesce;   /* number of page coalesces done */
        ulong_t fsf_time;       /* nanoseconds of run time */
        ulong_t fsf_releases;   /* number of page_release() done */
} fsf_stat_t;

fsf_stat_t fsf_recent;  /* counts for most recent duty cycle */
fsf_stat_t fsf_total;   /* total of counts */
```

They are kept in a global variable called `fsf_total` of `fsf_stat_t`, which the `fsflush.d` script reads using the ` kernel variable prefix.

Script

Since the counters are incremental, it prints out the delta every second:

```
#!/usr/sbin/dtrace -s

#pragma D option quiet

BEGIN
{
   lexam = 0; lscan = 0; llock = 0; lmod = 0; lcoal = 0; lrel = 0; lti = 0;
   printf("%10s %10s %10s %10s %10s %10s %10s\n", "SCANNED", "EXAMINED",
       "LOCKED", "MODIFIED", "COALESCE", "RELEASES", "TIME(ns)");
}

tick-1s
/lexam/
{
    printf("%10d %10d %10d %10d %10d %10d %10d\n", `fsf_total.fsf_scan,
        `fsf_total.fsf_examined - lexam, `fsf_total.fsf_locked - llock,
        `fsf_total.fsf_modified - lmod, `fsf_total.fsf_coalesce - lcoal,
        `fsf_total.fsf_releases - lrel, `fsf_total.fsf_time - ltime);
    lexam = `fsf_total.fsf_examined;
    lscan = `fsf_total.fsf_scan;
    llock = `fsf_total.fsf_locked;
    lmod = `fsf_total.fsf_modified;
    lcoal = `fsf_total.fsf_coalesce;
```

4. This was originally posted at *http://blogs.sun.com/jonh/entry/fsflush_revisited_in_d*.

```
    lrel = `fsf_total.fsf_releases;
    ltime = `fsf_total.fsf_time;
}

/*
 * First time through
 */

tick-1s
/!lexam/
{
    lexam = `fsf_total.fsf_examined;
    lscan = `fsf_total.fsf_scan;
    llock = `fsf_total.fsf_locked;
    lmod = `fsf_total.fsf_modified;
    lcoal = `fsf_total.fsf_coalesce;
    ltime = `fsf_total.fsf_time;
    lrel = `fsf_total.fsf_releases;
}
```

Script fsflush.d

This script uses the profile provider for the `tick-1s` probes, which is a stable provider. The script itself isn't considered stable, because it retrieves kernel internal statistics that may be subject to change (`fsf_stat_t`).

Example

```
solaris# fsflush.d
   SCANNED   EXAMINED     LOCKED   MODIFIED   COALESCE   RELEASES    TIME(ns)
     34871      34872       2243        365          0          0     3246343
     34871      34872       1576        204          0          0     2727493
     34871      34872       1689        221          0          0     2904566
     34871      34872        114         19          0          0     2221724
     34871      34872       1849        892          0          0     3297796
     34871      34872       1304        517          0          0     3408503
[...]
```

UFS Scripts

UFS is the Unix File System, based on Fast File System (FFS), and was the main file system used by Solaris until ZFS. UFS exists on other operating systems, including FreeBSD, where it can also be examined using DTrace. Although the on-disk structures and basic operation of UFS are similar, the implementation of UFS differs between operating systems. This is noticeable when listing the UFS probes via the fbt provider:

```
solaris# dtrace -ln 'fbt::ufs_*:' | wc -l
     403

freebsd# dtrace -ln 'fbt::ufs_*:' | wc -l
     107
```

For comparison, only those beginning with `ufs_` are listed. The fbt provider on Oracle Solaris can match the module name as `ufs`, so the complete list of UFS probes can be listed using `fbt:ufs::` (which shows 832 probes).

This section demonstrates UFS tracing on Oracle Solaris and is intended for those wanting to dig deeper into file system internals, beyond what is possible at the syscall and VFS layers. A basic understanding of UFS internals is assumed, which you can study in Chapter 15, The UFS File System, of *Solaris Internals* (McDougall and Mauro, 2006).

Since there is currently no stable UFS provider, the fbt[5] provider is used. fbt is an unstable interface: It exports kernel functions and data structures that may change from release to release. The following scripts were based on OpenSolaris circa December 2009 and may not work on other OSs and releases without changes. Even if these scripts no longer execute, they can still be treated as examples of D programming and for the sort of data that DTrace can make available for UFS analysis.

ufssnoop.d

This script uses the fbt provider to trace and print UFS calls from within the ufs kernel module. It provides a raw dump of what UFS is being requested to do, which can be useful for identifying load issues. Since the output is verbose and inclusive, it is suitable for post-processing, such as filtering for events of interest.

The script is included here to show that this is possible and how it might look. This is written for a particular version of Oracle Solaris ZFS and will need tweaks to work on other versions. The functionality and output is similar to `solvfssnoop.d` shown earlier.

Script

Common UFS requests are traced: See the probe names on lines 33 to 35. This script can be enhanced to include more request types as desired: See the source file on line 12 for the list.

```
1   #!/usr/sbin/dtrace -Zs
2
3   #pragma D option quiet
4   #pragma D option switchrate=10hz
5
6   dtrace:::BEGIN
7   {
8           printf("%-12s %6s %6s %-12.12s %-12s %-4s %s\n", "TIME(ms)", "UID",
9               "PID", "PROCESS", "CALL", "KB", "PATH");
10  }
```

5. See the “fbt Provider” section in Chapter 12 for more discussion about use of the fbt provider.

```
11
12  /* see uts/common/fs/ufs/ufs_vnops.c */
13
14  fbt::ufs_read:entry, fbt::ufs_write:entry
15  {
16          self->path = args[0]->v_path;
17          self->kb = args[1]->uio_resid / 1024;
18  }
19
20  fbt::ufs_open:entry
21  {
22          self->path = (*(struct vnode **)arg0)->v_path;
23          self->kb = 0;
24  }
25
26  fbt::ufs_close:entry, fbt::ufs_ioctl:entry, fbt::ufs_getattr:entry,
27  fbt::ufs_readdir:entry
28  {
29          self->path = args[0]->v_path;
30          self->kb = 0;
31  }
32
33  fbt::ufs_read:entry, fbt::ufs_write:entry, fbt::ufs_open:entry,
34  fbt::ufs_close:entry, fbt::ufs_ioctl:entry, fbt::ufs_getattr:entry,
35  fbt::ufs_readdir:entry
36  {
37          printf("%-12d %6d %6d %-12.12s %-12s %-4d %s\n", timestamp / 1000000,
38              uid, pid, execname, probefunc, self->kb,
39              self->path != NULL ? stringof(self->path) : "<null>");
40          self->path = 0; self->kb = 0;
41  }
```

Script ufssnoop.d

As another lesson in the instability of the fbt provider, the `ufs_open()` call doesn't exist on earlier versions of UFS. For this script to provide some functionality without it, the `-Z` option is used on line 1 so that the script will execute despite missing a probe, and line 22 casts `arg0` instead of using `args[0]` so that the script compiles.

Example

To test this script, the `dd(1)` command was used to perform three 8KB reads from a file:

```
solaris# ufssnoop.d
TIME(ms)        UID    PID PROCESS      CALL         KB   PATH
1155732900        0   8312 dd           ufs_open     0    /mnt/1m
1155732901        0   8312 dd           ufs_read     8    /mnt/1m
1155732901        0   8312 dd           ufs_read     8    /mnt/1m
1155732901        0   8312 dd           ufs_read     8    /mnt/1m
1155732901        0   8312 dd           ufs_close    0    /mnt/1m
1155739611        0   8313 ls           ufs_getattr  0    /mnt
1155739611        0   8313 ls           ufs_getattr  0    /mnt
[...]
```

The events have been traced correctly. The `TIME(ms)` column showed no delay between these reads, suggesting that the data returned from DRAM cache. This column can also be used for postsorting, because the output may become shuffled slightly on multi-CPU systems.

ufsreadahead.d

Oracle Solaris UFS uses read-ahead to improve the performance of sequential workloads. This is where a sequential read pattern is detected, allowing UFS to predict the next requested reads and issue them before they are actually requested, to prewarm the cache.

The `ufsreadahead.d` script shows bytes read by UFS and those requested by read-ahead. This can be used on a known sequential workload to check that read-ahead is working correctly and also on an unknown workload to determine whether it is sequential or random.

Script

Since this script is tracing UFS internals using the fbt provider and will require maintenance, it has been kept as simple as possible:

```
#!/usr/sbin/dtrace -s

fbt::ufs_getpage:entry
{
        @["UFS read (bytes)"] = sum(arg2);
}

fbt::ufs_getpage_ra:return
{
        @["UFS read ahead (bytes)"] = sum(arg1);
}
```

Script ufsreadahead.d

Example

The following example shows the use of `ufsreadahead.d` examining a sequential/streaming read workload:

```
solaris# ufsreadahead.d
dtrace: script './ufsreadahead.d' matched 2 probes
^C

  UFS read ahead (bytes)                                        70512640
  UFS read (bytes)                                              71675904
```

This was a known sequential read workload. The output shows that about 71MB were reads from UFS and 70MB were from read-ahead, suggesting that UFS has correctly detected this as sequential. (It isn't certain, since the script isn't checking that the read-ahead data was then actually read by anyone.)

Here we see the same script applied to a random read workload:

```
solaris# ufsreadahead.d
dtrace: script './ufsreadahead.d' matched 2 probes
^C

  UFS read (bytes)                                            2099136
```

This was a known random read workload that performed 2MB of reads from UFS. No read-ahead was triggered, which is what we would expect (hope).

See Also

For more examples of UFS read-ahead analysis using DTrace, see the `fspaging.d` and `fsrw.d` scripts from the DTraceToolkit, which trace I/O from the syscall layer to the storage device layer. Here's an example:

```
solaris# fsrw.d
Event            Device RW     Size Offset Path
sc-read               .  R     8192      0 /mnt/bigfile
  fop_read            .  R     8192      0 /mnt/bigfile
    disk_io        sd15  R     8192      0 /mnt/bigfile
    disk_ra        sd15  R     8192      8 /mnt/bigfile
sc-read               .  R     8192      8 /mnt/bigfile
  fop_read            .  R     8192      8 /mnt/bigfile
    disk_ra        sd15  R    81920     16 /mnt/bigfile
    disk_ra        sd15  R     8192     96 <none>
    disk_ra        sd15  R     8192     96 /mnt/bigfile
sc-read               .  R     8192     16 /mnt/bigfile
  fop_read            .  R     8192     16 /mnt/bigfile
    disk_ra        sd15  R   131072    104 /mnt/bigfile
    disk_ra        sd15  R  1048576    232 /mnt/bigfile
sc-read               .  R     8192     24 /mnt/bigfile
  fop_read            .  R     8192     24 /mnt/bigfile
sc-read               .  R     8192     32 /mnt/bigfile
  fop_read            .  R     8192     32 /mnt/bigfile
[...]
```

This output shows five syscall reads (`sc-read`) of 8KB in size, starting from file offset 0 and reaching file offset 32 (kilobytes). The first of these syscall reads triggers an 8KB VFS read (`fop_read`), which triggers a disk read to satisfy it (`disk_io`); also at this point, UFS read-ahead triggers the next 8KB to be read from disk (`disk_ra`). The next syscall read triggers three more read-aheads. The last read-ahead seen in this output shows a 1MB read from offset 232, and yet the syscall interface—what's actually being requested of UFS—has only had three 8KB reads at this point. That's optimistic!

ufsimiss.d

The Oracle Solaris UFS implementation uses an inode cache to improve the performance of inode queries. There are various kernel statistics we can use to observe the performance of this cache, for example:

```
solaris# kstat -p ufs::inode_cache:hits ufs::inode_cache:misses 1
ufs:0:inode_cache:hits  580003
ufs:0:inode_cache:misses        1294907

ufs:0:inode_cache:hits  581810
ufs:0:inode_cache:misses        1299367

ufs:0:inode_cache:hits  582973
ufs:0:inode_cache:misses        1304608
[...]
```

These counters show a high rate of inode cache misses. DTrace can investigate these further: The `ufsimiss.d` script shows the process and filename for each inode cache miss.

Script

The parent directory vnode and filename pointers are cached on `ufs_lookup()` for later printing if an inode cache miss occurred, and `ufs_alloc_inode()` was entered:

```
#!/usr/sbin/dtrace -s

#pragma D option quiet
#pragma D option switchrate=10hz

dtrace:::BEGIN
{
        printf("%6s %-16s %s\n", "PID", "PROCESS", "INODE MISS PATH");
}

fbt::ufs_lookup:entry
{
        self->dvp = args[0];
        self->name = arg1;
}

fbt::ufs_lookup:return
{
        self->dvp = 0;
        self->name = 0;
}

fbt::ufs_alloc_inode:entry
```

```
/self->dvp && self->name/
{
        printf("%6d %-16s %s/%s\n", pid, execname,
            stringof(self->dvp->v_path), stringof(self->name));
}

Script ufsimiss.d
```

Example

Here the UFS inode cache misses were caused by `find(1)` searching `/usr/share/man`:

```
solaris# ufsimiss.d
   PID PROCESS          INODE MISS PATH
 22966 find             /usr/share/man/sman3tiff/TIFFCheckTile.3tiff
 22966 find             /usr/share/man/sman3tiff/TIFFClientOpen.3tiff
 22966 find             /usr/share/man/sman3tiff/TIFFCurrentRow.3tiff
 22966 find             /usr/share/man/sman3tiff/TIFFDefaultStripSize.3tiff
 22966 find             /usr/share/man/sman3tiff/TIFFFileno.3tiff
 22966 find             /usr/share/man/sman3tiff/TIFFGetVersion.3tiff
 22966 find             /usr/share/man/sman3tiff/TIFFIsMSB2LSB.3tiff
 22966 find             /usr/share/man/sman3tiff/TIFFIsTiled.3tiff
 22966 find             /usr/share/man/sman3tiff/TIFFIsUpSampled.3tiff
[...]
```

ZFS Scripts

ZFS is an advanced file system and volume manager available on Oracle Solaris. Its features include 128-bit capacity, different RAID types, copy-on-write transactions, snapshots, clones, dynamic striping, variable block size, end-to-end checksumming, built-in compression, data-deduplication, support for hybrid storage pools, quotas, and more. The interaction of these features is interesting for those examining file system performance, and they have become a common target for DTrace.

ZFS employs an I/O pipeline (ZIO) that ends with aggregation of I/O at the device level. By the time an I/O is sent to disk, the content may refer to multiple files (specifically, there is no longer a single `vnode_t` for that I/O). Because of this, the io provider on ZFS can't show the path name for I/O; this has been filed as a bug (CR 6266202 "DTrace io provider doesn't work with ZFS"). At the time of writing, this bug has not been fixed. The ZFS path name of disk I/O can still be fetched with a little more effort using DTrace; the `ziosnoop.d` script described next shows one way to do this. For reads, it may be possible to simply identify slow reads at the ZFS interface, as demonstrated by the `zfsslower.d` script.

This section demonstrates ZFS tracing on Oracle Solaris and is intended for those wanting to dig deeper into file system internals, beyond what is possible at the syscall and VFS layers. An understanding of ZFS internals is assumed.

Since there is currently no stable ZFS provider, the fbt[6] provider is used. fbt is an unstable interface: It exports kernel functions and data structures that may change from release to release. The following scripts were based on OpenSolaris circa December 2009 and may not work on other OSs and releases without changes. Even if these scripts no longer execute, they can still be treated as examples of D programming and for the sort of data that DTrace can make available for ZFS analysis.

zfssnoop.d

This script uses the fbt provider to trace and print ZFS calls from within the zfs kernel module. It provides a raw dump of what ZFS is being requested to do, which can be useful for identifying load issues. Since the output is verbose and inclusive, it is suitable for postprocessing, such as filtering for events of interest. The functionality and output is similar to `solvfssnoop.d` shown earlier.

Script

Common ZFS requests are traced; see the probe names on lines 33 to 35. This script can be enhanced to include more request types as desired; see the source file on line 12 for the list.

```
1   #!/usr/sbin/dtrace -s
2
3   #pragma D option quiet
4   #pragma D option switchrate=10hz
5
6   dtrace:::BEGIN
7   {
8           printf("%-12s %6s %6s %-12.12s %-12s %-4s %s\n", "TIME(ms)", "UID",
9               "PID", "PROCESS", "CALL", "KB", "PATH");
10  }
11
12  /* see uts/common/fs/zfs/zfs_vnops.c */
13
14  fbt::zfs_read:entry, fbt::zfs_write:entry
15  {
16          self->path = args[0]->v_path;
17          self->kb = args[1]->uio_resid / 1024;
18  }
19
20  fbt::zfs_open:entry
21  {
22          self->path = (*args[0])->v_path;
23          self->kb = 0;
24  }
25
26  fbt::zfs_close:entry, fbt::zfs_ioctl:entry, fbt::zfs_getattr:entry,
27  fbt::zfs_readdir:entry
```

6. See the “fbt Provider” section in Chapter 12 for more discussion about use of the fbt provider.

```
{
        self->path = args[0]->v_path;
        self->kb = 0;
}

fbt::zfs_read:entry, fbt::zfs_write:entry, fbt::zfs_open:entry,
fbt::zfs_close:entry, fbt::zfs_ioctl:entry, fbt::zfs_getattr:entry,
fbt::zfs_readdir:entry
{
        printf("%-12d %6d %6d %-12.12s %-12s %-4d %s\n", timestamp / 1000000,
            uid, pid, execname, probefunc, self->kb,
            self->path != NULL ? stringof(self->path) : "<null>");
        self->path = 0; self->kb = 0;
}
```

Script zfssnoop.d

The `TIME(ms)` column can be used for postsorting, because the output may become shuffled slightly on multi-CPU systems.

Example

The following script was run on a desktop to identify ZFS activity:

```
solaris# zfssnoop.d
TIME(ms)        UID    PID PROCESS      CALL         KB   PATH
19202174470     102  19981 gnome-panel  zfs_getattr  0    /export/home/claire/.gnome2/
vfolders
19202174470     102  19981 gnome-panel  zfs_getattr  0    /export/home/claire/.gnome2/
vfolders
19202174470     102  19981 gnome-panel  zfs_getattr  0    /export/home/claire/.gnome2/
vfolders
19202174470     102  19981 gnome-panel  zfs_getattr  0    /export/home/claire/.gnome2/
vfolders
19202174470     102  19981 gnome-panel  zfs_getattr  0    /export/home/claire/.recentl
y-used
19202175400     101   2903 squid        zfs_open     0    /squidcache/05/03
19202175400     101   2903 squid        zfs_getattr  0    /squidcache/05/03
19202175400     101   2903 squid        zfs_readdir  0    /squidcache/05/03
19202175400     101   2903 squid        zfs_readdir  0    /squidcache/05/03
19202175400     101   2903 squid        zfs_close    0    /squidcache/05/03
19202175427     102  23885 firefox-bin  zfs_getattr  0    /export/home/claire/.recentl
yused.xbe
l
19202176030     102  13622 nautilus     zfs_getattr  0    /export/home/claire/Desktop
19202176215     102  23885 firefox-bin  zfs_read     3    /export/home/claire/.mozilla
/firefox/3c8k4kh0.default/Cache/_CACHE_002_
19202176216     102  23885 firefox-bin  zfs_read     3    /export/home/claire/.mozilla
/firefox/3c8k4kh0.default/Cache/_CACHE_002_
19202176215     102  23885 firefox-bin  zfs_read     0    /export/home/claire/.mozilla
/firefox/3c8k4kh0.default/Cache/_CACHE_001_
19202176216     102  23885 firefox-bin  zfs_read     0    /export/home/claire/.mozilla
/firefox/3c8k4kh0.default/Cache/_CACHE_001_
[...]
```

Various ZFS calls have been traced, including `gnome-panel` checking file attributes and `firefox-bin` reading cache files.

zfsslower.d

This is a variation of the `zfssnoop.d` script intended for the analysis of performance issues. `zfsslower.d` shows the time for read and write I/O in milliseconds. A minimum number of milliseconds can be provided as an argument when running the script, which causes it to print only I/O equal to or slower than the provided milliseconds.

Because of CR 6266202 (mentioned earlier), we currently cannot trace disk I/O with ZFS filename information using the io provider arguments. `zfsslower.d` may be used as a workaround: By executing it with a minimum time that is likely to ensure that it is disk I/O (for example, at least 2 ms), we can trace likely disk I/O events with ZFS filename information.

Script

The `defaultargs` pragma is used on line 4 so that an optional argument can be provided of the minimum I/O time to print. If no argument is provided, the minimum time is zero, since `$1` will be 0 on line 11.

```
1   #!/usr/sbin/dtrace -s
2
3   #pragma D option quiet
4   #pragma D option defaultargs
5   #pragma D option switchrate=10hz
6
7   dtrace:::BEGIN
8   {
9           printf("%-20s %-16s %1s %4s %6s %s\n", "TIME", "PROCESS",
10              "D", "KB", "ms", "FILE");
11          min_ns = $1 * 1000000;
12  }
13
14  /* see uts/common/fs/zfs/zfs_vnops.c */
15
16  fbt::zfs_read:entry, fbt::zfs_write:entry
17  {
18          self->path = args[0]->v_path;
19          self->kb = args[1]->uio_resid / 1024;
20          self->start = timestamp;
21  }
22
23  fbt::zfs_read:return, fbt::zfs_write:return
24  /self->start && (timestamp - self->start) >= min_ns/
25  {
26          this->iotime = (timestamp - self->start) / 1000000;
27          this->dir = probefunc == "zfs_read" ? "R" : "W";
28          printf("%-20Y %-16s %1s %4d %6d %s\n", walltimestamp,
29              execname, this->dir, self->kb, this->iotime,
30              self->path != NULL ? stringof(self->path) : "<null>");
31  }
32
33  fbt::zfs_read:return, fbt::zfs_write:return
34  {
35          self->path = 0; self->kb = 0; self->start = 0;
36  }
```

Script zfsslower.d

Example

Here the `zfsslower.d` script was run with an argument of 1 to show only ZFS reads and writes that took 1 millisecond or longer:

```
solaris# zfsslower.d 1
TIME                 PROCESS          D   KB     ms FILE
2010 Jun 26 03:28:49 cat              R    8     14 /export/home/brendan/randread.pl
2010 Jun 26 03:29:04 cksum            R    4      5 /export/home/brendan/perf.tar
2010 Jun 26 03:29:04 cksum            R    4     20 /export/home/brendan/perf.tar
2010 Jun 26 03:29:04 cksum            R    4     34 /export/home/brendan/perf.tar
2010 Jun 26 03:29:04 cksum            R    4      7 /export/home/brendan/perf.tar
2010 Jun 26 03:29:04 cksum            R    4     12 /export/home/brendan/perf.tar
2010 Jun 26 03:29:04 cksum            R    4      1 /export/home/brendan/perf.tar
2010 Jun 26 03:29:04 cksum            R    4     81 /export/home/brendan/perf.tar
[...]
```

The files accessed here were not cached and had to be read from disk.

zioprint.d

The ZFS I/O pipeline (ZIO) is of particular interest for performance analysis or troubleshooting, because it processes, schedules, and issues device I/O. It does this through various stages whose function names (and hence fbt provider probe names) have changed over time. Because of this, a script that traces specific ZIO functions would execute only on a particular kernel version and would require regular maintenance to match kernel updates.

The `zioprint.d` script addresses this by matching all zio functions using a wildcard, dumping data generically, and leaving the rest to postprocessing of the output (for example, using Perl).

Script

This script prints the first five arguments on function entry as hexadecimal integers, whether or not that's meaningful (which can be determined later during postprocessing). For many of these functions, the first argument on entry is the address of a `zio_t`, so a postprocessor can use that address as a key to follow that zio through the stages. The return offset and value are also printed.

```
#!/usr/sbin/dtrace -s

#pragma D option quiet
#pragma D option switchrate=10hz

dtrace:::BEGIN
{
        printf("%-16s %-3s %-22s %-6s %s\n", "TIME(us)", "CPU", "FUNC",
            "NAME", "ARGS");
}
```

continues

```
11
12  fbt::zio_*:entry
13  {
14          printf("%-16d %-3d %-22s %-6s %x %x %x %x %x\n", timestamp / 1000,
15              cpu, probefunc, probename, arg0, arg1, arg2, arg3, arg4);
16  }
17
18  fbt::zio_*:return
19  {
20          printf("%-16d %-3d %-22s %-6s %x %x\n", timestamp / 1000, cpu,
21              probefunc, probename, arg0, arg1);
22  }
```

Script zioprint.d

This script can be reused to dump events from any kernel area by changing the probe names on lines 12 and 18.

Example

The script is intended to be used to write a dump file (either by using shell redirection `>` or via the `dtrace(1M) -o` option) for postprocessing. Since the script is generic, it is likely to execute on any kernel version and produce a dump file, which can be especially handy in situations with limited access to the target system but unlimited access to any other system (desktop/laptop) for postprocessing.

```
solaris# zioprint.d
TIME(us)         CPU FUNC                   NAME   ARGS
1484927856573    0   zio_taskq_dispatch     entry  ffffff4136711c98 2 0 4a 49
1484927856594    0   zio_taskq_dispatch     return ac ffffff4456fc8090
1484927856616    0   zio_interrupt          return 1d ffffff4456fc8090
1484927856630    0   zio_execute            entry  ffffff4136711c98 ffffff4456fc8090
a477aa00 a477aa00 c2244e36f410a
1484927856643    0   zio_vdev_io_done       entry  ffffff4136711c98 ffffff4456fc8090
a477aa00 a477aa00 12
1484927856653    0   zio_wait_for_children  entry  ffffff4136711c98 0 1 a477aa00 12
1484927856658    0   zio_wait_for_children  return 7b 0
1484927856667    0   zio_vdev_io_done       return 117 100
[...]
```

The meaning of each hexadecimal argument can be determined by reading the ZFS source for that kernel version. For example, the `zio_wait_for_children()` calls shown earlier have the function prototype:

```
usr/src/uts/common/fs/zfs/zio.c:

static boolean_t
zio_wait_for_children(zio_t *zio, enum zio_child child, enum zio_wait_type wait)
```

which means that the entry traced earlier has a `zio_t` with address `ffffff4136711c98` and a `zio_wait_type` of 1 (`ZIO_WAIT_DONE`). The additional arguments printed (`a477aa00` and `12`) are leftover register values that are not part of the function entry arguments.

ziosnoop.d

The `ziosnoop.d` script is an enhancement of `zioprint.d`, by taking a couple of the functions and printing useful information from the kernel—including the pool name and file path name. The trade-off is that these additions make the script more fragile and may require maintenance to match kernel changes.

Script

The `zio_create()` and `zio_done()` functions were chosen as start and end points for ZIO (`zio_destroy()` may be a better endpoint, but it didn't exist on earlier kernel versions). For `zio_create()`, information about the requested I/O including pool name and file path name (if known) are printed. On `zio_done()`, the results of the I/O, including device path (if present) and error values, are printed.

```
#!/usr/sbin/dtrace -s

#pragma D option quiet
#pragma D option defaultargs
#pragma D option switchrate=10hz

dtrace:::BEGIN
{
        start = timestamp;
        printf("%-10s %-3s %-12s %-16s %s\n", "TIME(us)", "CPU",
            "ZIO_EVENT", "ARG0", "INFO (see script)");
}

fbt::zfs_read:entry, fbt::zfs_write:entry   { self->vp = args[0]; }
fbt::zfs_read:return, fbt::zfs_write:return { self->vp = 0; }

fbt::zio_create:return
/$1 || args[1]->io_type/
{
        /* INFO: pool zio_type zio_flag bytes path */
        printf("%-10d %-3d %-12s %-16x %s %d %x %d %s\n",
            (timestamp - start) / 1000, cpu, "CREATED", arg1,
            stringof(args[1]->io_spa->spa_name), args[1]->io_type,
            args[1]->io_flags, args[1]->io_size, self->vp &&
            self->vp->v_path ? stringof(self->vp->v_path) : "<null>");
}

fbt::zio_*:entry
/$1/
{
        printf("%-10d %-3d %-12s %-16x\n", (timestamp - start) / 1000, cpu,
            probefunc, arg0);
}
```

continues

```

fbt::zio_done:entry
/$1 || args[0]->io_type/
{
        /* INFO: io_error vdev_state vdev_path */
        printf("%-10d %-3d %-12s %-16x %d %d %s\n", (timestamp - start) / 1000,
            cpu, "DONE", arg0, args[0]->io_error,
            args[0]->io_vd ? args[0]->io_vd->vdev_state : 0,
            args[0]->io_vd && args[0]->io_vd->vdev_path ?
            stringof(args[0]->io_vd->vdev_path) : "<null>");
}
```

Script ziosnoop.d

By default, only `zio_create()` and `zio_done()` are traced; if an optional argument of 1 (nonzero) is provided, the script traces all other zio functions as well.

Examples

This is the default output:

```
solaris# ziosnoop.d
TIME(us)    CPU ZIO_EVENT    ARG0             INFO (see script)
75467       2   CREATED      ffffff4468f79330 pool0 1 40440 131072 /pool0/fs1/1t
96330       2   CREATED      ffffff44571b1360 pool0 1 40 131072 /pool0/fs1/1t
96352       2   CREATED      ffffff46510a7cc0 pool0 1 40440 131072 /pool0/fs1/1t
96363       2   CREATED      ffffff4660b4a048 pool0 1 40440 131072 /pool0/fs1/1t
24516       5   DONE         ffffff59a619ecb0 0 7 /dev/dsk/c0t5000CCA20ED60516d0s0
24562       5   DONE         ffffff4141ecd340 0 7 <null>
24578       5   DONE         ffffff4465456320 0 0 <null>
34836       5   DONE         ffffff4141f8dca8 0 7 /dev/dsk/c0t5000CCA20ED60516d0s0
34854       5   DONE         ffffff414d8e8368 0 7 <null>
34867       5   DONE         ffffff446c3de9b8 0 0 <null>
44818       5   DONE         ffffff5b3defd968 0 7 /dev/dsk/c0t5000CCA20ED60164d0s0
[...]
```

Note the `TIME(us)` column—the output is shuffled. To see it in the correct order, write to a file and postsort on that column.

Running `ziosnoop.d` with an argument of 1 will execute verbose mode, printing all zio calls. Here it is written to a file, from which a particular `zio_t` address is searched using `grep(1)`:

```
solaris# ziosnoop.d 1 -o ziodump
solaris# more ziodump
TIME(us)    CPU ZIO_EVENT    ARG0             INFO (see script)
[...]
171324      6   CREATED      ffffff6440130368 pool0 1 40440 131072 /pool0/fs1/1t
171330      6   zio_nowait   ffffff6440130368
171332      6   zio_execute  ffffff6440130368
[...]
solaris# grep ffffff6440130368 ziodump | sort -n +0
```

```
171324     6   CREATED      ffffff6440130368 pool0 1 40440 131072 /pool0/fs1/1t
171330     6   zio_nowait   ffffff6440130368
171332     6   zio_execute  ffffff6440130368
171334     6   zio_vdev_io_start ffffff6440130368
179672     0   zio_interrupt ffffff6440130368
179676     0   zio_taskq_dispatch ffffff6440130368
179689     0   zio_execute  ffffff6440130368
179693     0   zio_vdev_io_done ffffff6440130368
179695     0   zio_wait_for_children ffffff6440130368
179698     0   zio_vdev_io_assess ffffff6440130368
179700     0   zio_wait_for_children ffffff6440130368
179702     0   zio_checksum_verify ffffff6440130368
179705     0   zio_checksum_error ffffff6440130368
179772     0   zio_done     ffffff6440130368
179775     0   DONE         ffffff6440130368 0 7 /dev/dsk/c0t5000CCA20ED60516d0s0
[...]
```

The output of `grep(1)` is passed to `sort(1)` to print the events in the correct timestamp order. Here, all events from `zio_create()` to `zio_done()` can be seen, along with the time stamp. Note the jump in time between `zio_vdev_io_start()` and `zio_interrupt()` (171334 us to 179672 us = 8 ms)—this is the device I/O time. Latency in other zio stages can be identified in the same way (which can be expedited by writing a postprocessor).

ziotype.d

The `ziotype.d` script shows what types of ZIO are being created, printing a count every five seconds.

Script

A translation table for `zio_type` is included in the `BEGIN` action, based on `zfs.h`. If `zfs.h` changes with kernel updates, this table will need to be modified to match.

```
#!/usr/sbin/dtrace -s

#pragma D option quiet

dtrace:::BEGIN
{
        /* see /usr/include/sys/fs/zfs.h */
        ziotype[0] = "null";
        ziotype[1] = "read";
        ziotype[2] = "write";
        ziotype[3] = "free";
        ziotype[4] = "claim";
        ziotype[5] = "ioctl";
        trace("Tracing ZIO...  Output interval 5 seconds, or Ctrl-C.\n");
}

fbt::zio_create:return
/args[1]->io_type/              /* skip null */
{
        @[stringof(args[1]->io_spa->spa_name),
            ziotype[args[1]->io_type] != NULL ?
```

continues

```
                  ziotype[args[1]->io_type] : "?"] = count();
}

profile:::tick-5sec,
dtrace:::END
{
        printf("\n %-32s %-10s %10s\n", "POOL", "ZIO_TYPE", "CREATED");
        printa(" %-32s %-10s %@10d\n", @);
        trunc(@);
}
```

Script zioype.d

Example

The example has identified a mostly write workload of about 12,000 write ZIO every five seconds:

```
solaris# ziotype.d
Tracing ZIO...  Output interval 5 seconds, or Ctrl-C.

 POOL                             ZIO_TYPE      CREATED
 pool0                            ioctl              28
 pool0                            free               48
 pool0                            read             1546
 pool0                            write           12375

 POOL                             ZIO_TYPE      CREATED
 pool0                            ioctl              14
 pool0                            free               24
 pool0                            read             1260
 pool0                            write           11929
[...]
```

perturbation.d

The `perturbation.d` script measures ZFS read/write performance during a given perturbation. This can be used to quantify the performance impact during events such as snapshot creation.

Script

The perturbation function name is provided as an argument, which DTrace makes available in the script as `$$1`.

```
#!/usr/sbin/dtrace -s

#pragma D option quiet
#pragma D option defaultargs

dtrace:::BEGIN
{
        printf("Tracing ZFS perturbation by %s()... Ctrl-C to end.\n", $$1);
}
```

```

fbt::$$1:entry
{
        self->pstart = timestamp;
        perturbation = 1;
}

fbt::$$1:return
/self->pstart/
{
        this->ptime = (timestamp - self->pstart) / 1000000;
        @[probefunc, "perturbation duration (ms)"] = quantize(this->ptime);
        perturbation = 0;
}

fbt::zfs_read:entry, fbt::zfs_write:entry
{
        self->start = timestamp;
}

fbt::zfs_read:return, fbt::zfs_write:return
/self->start/
{
        this->iotime = (timestamp - self->start) / 1000000;
        @[probefunc, perturbation ? "during perturbation (ms)" :
            "normal (ms)"] = quantize(this->iotime);
        self->start = 0;
}
```

Script perturbation.d

Example

Here we measure ZFS performance during snapshot creation. The `perturbation.d` script is run with the argument `zfs_ioc_snapshot`, a function call that encompasses snapshot creation (for this kernel version). While tracing, a read and write workload was executing on ZFS, and three snapshots were created:

```
solaris# perturbation.d zfs_ioc_snapshot
Tracing ZFS perturbation by zfs_ioc_snapshot()... Ctrl-C to end.
^C

  zfs_write                                           normal (ms)
           value  ------------- Distribution ------------- count
              -1 |                                         0
               0 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 348381
               1 |                                         7
               2 |                                         0

  zfs_write                                           during perturbation (ms)
           value  ------------- Distribution ------------- count
              -1 |                                         0
               0 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 276029
               1 |                                         11
               2 |                                         5
               4 |                                         0

  zfs_ioc_snapshot                                    perturbation duration (ms)
           value  ------------- Distribution ------------- count
```

continues

```
             512 |                                         0
            1024 |@@@@@@@@@@@@@@@@@@@@@@@@@@@               2
            2048 |@@@@@@@@@@@@@                             1
            4096 |                                         0

  zfs_read                                       during perturbation (ms)
           value  ------------- Distribution ------------- count
              -1 |                                         0
               0 |@                                        5
               1 |                                         0
               2 |                                         0
               4 |                                         3
               8 |@@@@@@@@@@@@                             77
              16 |@@@@@@@@@@@@@@@@@@                       117
              32 |@@@@                                     26
              64 |@@                                       16
             128 |@                                        8
             256 |                                         2
             512 |@                                        5
            1024 |                                         0

  zfs_read                                       normal (ms)
           value  ------------- Distribution ------------- count
              -1 |                                         0
               0 |@@@@                                     97
               1 |                                         0
               2 |                                         0
               4 |@                                        29
               8 |@@@@@@@@@@@@@@@@@@@@@@@@                 563
              16 |@@@@@@@@@@                               241
              32 |                                         10
              64 |                                         1
             128 |                                         0
```

The impact on performance can be seen clearly in the last distribution plots for ZFS reads. In normal operation, the time for ZFS reads was mostly between 8 ms and 31 ms. During snapshot create, some ZFS reads were taking 32 ms and longer, with the slowest five I/O in the 512-ms to 1023-ms range. Fortunately, these are outliers: Most of the I/O was still in the 8-ms to 31-ms range, despite a snapshot being created.

Another target for `perturbation.d` can be the `spa_sync()` function.

Note that `perturbation.d` cannot be run without any arguments; if that is tried, DTrace will error because the `$$1` macro variable is undefined:

```
solaris# perturbation.d
dtrace: failed to compile script perturbation.d: line 11: invalid probe description "f
bt::$$1:entry": Undefined macro variable in probe description
```

A function name must be provided for DTrace to trace.

spasync.d

The `spa_sync()` function flushes a ZFS transaction group (TXG) to disk, which consists of dirty data written since the last `spa_sync()`.

Script

This script has a long history: Earlier versions were created by the ZFS engineering team and can be found in blog entries.[7] Here it has been rewritten to keep it short and to print only `spa_sync()` events that were longer than one millisecond—tunable on line 5:

```
1    #!/usr/sbin/dtrace -s
2
3    #pragma D option quiet
4
5    inline int MIN_MS = 1;
6
7    dtrace:::BEGIN
8    {
9          printf("Tracing ZFS spa_sync() slower than %d ms...\n", MIN_MS);
10         @bytes = sum(0);
11   }
12
13   fbt::spa_sync:entry
14   /!self->start/
15   {
16         in_spa_sync = 1;
17         self->start = timestamp;
18         self->spa = args[0];
19   }
20
21   io:::start
22   /in_spa_sync/
23   {
24         @io = count();
25         @bytes = sum(args[0]->b_bcount);
26   }
27
28   fbt::spa_sync:return
29   /self->start && (this->ms = (timestamp - self->start) / 1000000) > MIN_MS/
30   {
31         normalize(@bytes, 1048576);
32         printf("%-20Y %-10s %6d ms, ", walltimestamp,
33             stringof(self->spa->spa_name), this->ms);
34         printa("%@d MB %@d I/O\n", @bytes, @io);
35   }
36
37   fbt::spa_sync:return
38   {
39         self->start = 0; self->spa = 0; in_spa_sync = 0;
40         clear(@bytes); clear(@io);
41   }
```

Script spasync.d

7. See *http://blogs.sun.com/roch/entry/128k_suffice* by Roch Bourbonnais, and see *www.cuddletech.com/blog/pivot/entry.php?id=1015* by Ben Rockwood.

Example

```
solaris# spa_sync.d
Tracing ZFS spa_sync() slower than 1 ms...
2010 Jun 17 01:46:18 pool-0       2679 ms, 31 MB 2702 I/O
2010 Jun 17 01:46:18 pool-0        269 ms, 0 MB 0 I/O
2010 Jun 17 01:46:18 pool-0        108 ms, 0 MB 0 I/O
2010 Jun 17 01:46:18 system        597 ms, 0 MB 0 I/O
2010 Jun 17 01:46:18 pool-0        184 ms, 0 MB 0 I/O
2010 Jun 17 01:46:19 pool-0        154 ms, 0 MB 0 I/O
2010 Jun 17 01:46:19 system        277 ms, 0 MB 0 I/O
2010 Jun 17 01:46:19 system         34 ms, 0 MB 0 I/O
2010 Jun 17 01:46:19 pool-0        226 ms, 27 MB 1668 I/O
2010 Jun 17 01:46:19 system        262 ms, 0 MB 0 I/O
2010 Jun 17 01:46:19 system        174 ms, 0 MB 0 I/O
[...]
```

HFS+ Scripts

HFS+ is the Hierarchal File System plus from Apple, described in Technical Note TN1150[8] and *Mac OS X Internals*.

```
macosx# dtrace -ln 'fbt::hfs_*:entry'
   ID   PROVIDER            MODULE                          FUNCTION NAME
 9396        fbt       mach_kernel                 hfs_addconverter entry
 9398        fbt       mach_kernel                         hfs_bmap entry
[...]
 9470        fbt       mach_kernel                   hfs_vnop_ioctl entry
 9472        fbt       mach_kernel          hfs_vnop_makenamedstream entry
 9474        fbt       mach_kernel                hfs_vnop_offtoblk entry
 9476        fbt       mach_kernel                  hfs_vnop_pagein entry
 9478        fbt       mach_kernel                 hfs_vnop_pageout entry
 9480        fbt       mach_kernel                    hfs_vnop_read entry
 9482        fbt       mach_kernel        hfs_vnop_removenamedstream entry
 9484        fbt       mach_kernel                  hfs_vnop_select entry
 9486        fbt       mach_kernel                hfs_vnop_strategy entry
 9488        fbt       mach_kernel                   hfs_vnop_write entry
```

Some of the functions in the HFS code are declared static, so their symbol information is not available for DTrace to probe. This includes `hfs_vnop_open()` and `hfs_vnop_close()`, which are missing from the previous list. Despite this, there are still enough visible functions from HFS+ for DTrace scripting: the functions that call HFS and the functions that HFS calls.

This section is intended for those wanting to dig deeper into file system internals, beyond what is possible at the syscall and VFS layers. A basic understanding of HFS+ internals is assumed, which can be studied in Chapter 12 of *Mac OS X Internals*.

8. See *http://developer.apple.com/mac/library/technotes/tn/tn1150.html*.

Since there is currently no stable HFS+ provider, the fbt[9] provider is used. fbt is an unstable interface: It exports kernel functions and data structures that may change from release to release. The following scripts were based on Mac OS X version 10.6 and may not work on other releases without changes. Even if these scripts no longer execute, they can still be treated as examples of D programming and for the sort of data that DTrace can make available for HFS+ analysis.

hfssnoop.d

This script uses the fbt provider to trace HFS+ calls from within the kernel (this will need tweaks to work on future Mac OS X kernels). It provides a raw dump of what HFS+ is being requested to do, which can be useful for identifying load issues. Since the output is verbose and inclusive, it is suitable for postprocessing, such as filtering for events of interest. The functionality and output is similar to `macvfssnoop.d` shown earlier.

Script

This script currently only traces reads and writes. Other available `hfs_vnop_*` functions can be added, and those not visible (such as open) can be traced from an upper layer, such as VFS (via `VNOP_*`, and filtering on HFS calls only).

```
#!/usr/sbin/dtrace -s

#pragma D option quiet
#pragma D option switchrate=10hz

dtrace:::BEGIN
{
      printf("%-12s %6s %6s %-12.12s %-14s %-4s %s\n", "TIME(ms)", "UID",
          "PID", "PROCESS", "CALL", "KB", "FILE");
}

/* see bsd/hfs/hfs_vnops.c */

fbt::hfs_vnop_read:entry
{
      this->read = (struct vnop_read_args *)arg0;
      self->path = this->read->a_vp->v_name;
      self->kb = this->read->a_uio->uio_resid_64 / 1024;
}

fbt::hfs_vnop_write:entry
{
      this->write = (struct vnop_write_args *)arg0;
      self->path = this->write->a_vp->v_name;
      self->kb = this->write->a_uio->uio_resid_64 / 1024;
}
```

continues

9. See the “fbt Provider” section in Chapter 12 for more discussion about use of the fbt provider.

```

fbt::hfs_vnop_read:entry, fbt::hfs_vnop_write:entry
{
      printf("%-12d %6d %6d %-12.12s %-14s %-4d %s\n", timestamp / 1000000,
          uid, pid, execname, probefunc, self->kb,
          self->path != NULL ? stringof(self->path) : "<null>");
      self->path = 0; self->kb = 0;
}
```

Script hfssnoop.d

Example

Here the `hfssnoop.d` script has traced `vim(1)` opening itself in another window to edit it:

```
macosx# hfssnoop.d
TIME(ms)        UID    PID PROCESS      CALL           KB   FILE
1311625280      501  67349 vim          hfs_vnop_read  4    LC_COLLATE
1311625280      501  67349 vim          hfs_vnop_read  0    LC_CTYPE/..namedfork/rsrc
1311625280      501  67349 vim          hfs_vnop_read  4    LC_CTYPE
[...]
1311625288      501  67349 vim          hfs_vnop_read  8    hfssnoop.d
1311625280      501  67349 vim          hfs_vnop_read  4    LC_CTYPE
1311625280      501  67349 vim          hfs_vnop_read  4    LC_CTYPE
1311625280      501  67349 vim          hfs_vnop_read  4    LC_CTYPE
1311625280      501  67349 vim          hfs_vnop_read  54   LC_CTYPE
1311625280      501  67349 vim          hfs_vnop_read  0    LC_MONETARY
1311625280      501  67349 vim          hfs_vnop_read  0    LC_NUMERIC
1311625280      501  67349 vim          hfs_vnop_read  0    LC_TIME
1311625280      501  67349 vim          hfs_vnop_read  0    LC_MESSAGES
1311625281      501  67349 vim          hfs_vnop_read  4    xterm-color
1311625282      501  67349 vim          hfs_vnop_read  4    vimrc
1311625282      501  67349 vim          hfs_vnop_read  4    vimrc
1311625284      501  67349 vim          hfs_vnop_read  4    netrwPlugin.vim
1311625284      501  67349 vim          hfs_vnop_read  4    netrwPlugin.vim
[...]
1311625285      501  67349 vim          hfs_vnop_read  4    zipPlugin.vim
1311625286      501  67349 vim          hfs_vnop_read  4    zipPlugin.vim
1311625288      501  67349 vim          hfs_vnop_write 4    .hfssnoop.d.swp
1311625288      501  67349 vim          hfs_vnop_read  64   hfssnoop.d
```

All the files read and written while vim was loading have been traced. The final lines show a swap file being written and vim reloading the `hfssnoop.d` file. The kilobyte sizes shown are those requested; many of these reads will have returned a smaller size in bytes (which can be shown, if desired, with more DTrace).

hfsslower.d

This is a variation of the `hfssnoop.d` script, intended for the analysis of performance issues. `hfsslower.d` shows the time for read and write I/O in milliseconds. A minimum number of milliseconds can be provided as an argument when running the script, which causes it to print only that I/O equal to or slower than the provided milliseconds.

Script

The `defaultargs` pragma is used on line 4 so that an optional argument can be provided of the minimum I/O time to print. If no argument is provided, the minimum time is zero, since `$1` will be 0 on line 11.

```
1     #!/usr/sbin/dtrace -s
2
3     #pragma D option quiet
4     #pragma D option defaultargs
5     #pragma D option switchrate=10hz
6
7     dtrace:::BEGIN
8     {
9           printf("%-20s %-16s %1s %4s %6s %s\n", "TIME", "PROCESS",
10              "D", "KB", "ms", "FILE");
11          min_ns = $1 * 1000000;
12    }
13
14    /* see bsd/hfs/hfs_vnops.c */
15
16    fbt::hfs_vnop_read:entry
17    {
18          this->read = (struct vnop_read_args *)arg0;
19          self->path = this->read->a_vp->v_name;
20          self->kb = this->read->a_uio->uio_resid_64 / 1024;
21          self->start = timestamp;
22    }
23
24    fbt::hfs_vnop_write:entry
25    {
26          this->write = (struct vnop_write_args *)arg0;
27          self->path = this->write->a_vp->v_name;
28          self->kb = this->write->a_uio->uio_resid_64 / 1024;
29          self->start = timestamp;
30    }
31
32    fbt::hfs_vnop_read:return, fbt::hfs_vnop_write:return
33    /self->start && (timestamp - self->start) >= min_ns/
34    {
35          this->iotime = (timestamp - self->start) / 1000000;
36          this->dir = probefunc == "hfs_vnop_read" ? "R" : "W";
37          printf("%-20Y %-16s %1s %4d %6d %s\n", walltimestamp,
38              execname, this->dir, self->kb, this->iotime,
39              self->path != NULL ? stringof(self->path) : "<null>");
40    }
41
42    fbt::hfs_vnop_read:return, fbt::hfs_vnop_write:return
43    {
44          self->path = 0; self->kb = 0; self->start = 0;
45    }
```

Script hfslower.d

Example

Here `hfsslower.d` is run with the argument 1 so that it prints out only the I/O that took one millisecond and longer:

```
macosx# hfsslower.d 1
TIME                 PROCESS          D   KB     ms FILE
2010 Jun 23 00:44:05 mdworker32       R    0     21 sandbox-cache.db
2010 Jun 23 00:44:05 mdworker32       R    0     19 AdiumSpotlightImporter
2010 Jun 23 00:44:05 mdworker32       R   16     18 schema.xml
2010 Jun 23 00:44:05 soffice          W    1      2 sve4a.tmp
2010 Jun 23 00:44:05 soffice          W    1      3 sve4a.tmp
2010 Jun 23 00:44:05 soffice          R   31      2 sve4a.tmp
2010 Jun 23 00:44:05 fontd            R    0     22 Silom.ttf/..namedfork/rsrc
^C
```

While tracing, there was many fast (less than 1 ms) I/Os to HFS that were filtered from the output.

hfsfileread.d

This script shows both logical (VFS) and physical (disk) reads to HFS+ files, showing data requests from the in-memory cache vs. disk.

Script

This script traces the size of read requests. The size of the returned data may be smaller than was requested or zero if the read failed; the returned size could also be traced if desired.

```
#!/usr/sbin/dtrace -s

#pragma D option quiet

dtrace:::BEGIN
{
      trace("Tracing HFS+ file reads... Hit Ctrl-C to end.\n");
}

fbt::hfs_vnop_read:entry
{
      this->read = (struct vnop_read_args *)arg0;
      this->path = this->read->a_vp->v_name;
      this->bytes = this->read->a_uio->uio_resid_64;
      @r[this->path ? stringof(this->path) : "<null>"] = sum(this->bytes);
}

fbt::hfs_vnop_strategy:entry
/((struct vnop_strategy_args *)arg0)->a_bp->b_flags & B_READ/
{
      this->strategy = (struct vnop_strategy_args *)arg0;
      this->path = this->strategy->a_bp->b_vp->v_name;
      this->bytes = this->strategy->a_bp->b_bcount;
      @s[this->path ? stringof(this->path) : "<null>"] = sum(this->bytes);
}

dtrace:::END
{
      printf(" %-56s %10s %10s\n", "FILE", "READ(B)", "DISK(B)");
      printa(" %-56s %@10d %@10d\n", @r, @s);
}
```

Script hfsfileread.d

Example

While tracing, there were about 240MB of requested reads to the `ss7000_b00.vmdk` file, about 230MB of which were from disk, meaning that this file is mostly uncached. The `10m_file` was completely read; however, 0 bytes were read from disk, meaning that it was entirely cached.

```
macosx# hfsfileread.d
Tracing HFS+ file reads... Hit Ctrl-C to end.
^C
 FILE                                                          READ(B)    DISK(B)
 swapfile1                                                           0       4096
 dyld/..namedfork/rsrc                                              50          0
 dyld                                                             4636          0
 cksum                                                           12288          0
 template.odt                                                   141312     143360
 10m_file                                                     10502144          0
 ss7000_b00.vmdk                                             246251520  230264832
```

PCFS Scripts

pcfs is an Oracle Solaris driver for the Microsoft FAT16 and FAT32 file systems. Though it was once popular for diskettes, today FAT file systems are more likely to be found on USB storage devices.

Since there is currently no stable PCFS provider, the fbt provider is used here. fbt instruments a particular operating system and version, so this script may therefore require modifications to match the software version you are using.

pcfsrw.d

This script shows `read()`, `write()`, and `readdir()` calls to pcfs, with details including file path name and latency for the I/O in milliseconds.

Script

This script traces `pcfs` kernel functions; if the `pcfs` module is not loaded (no `pcfs` in use), the script will not execute because the functions will not yet be present in the kernel for DTrace to find and probe. If desired, the `-Z` option can be added to line 1, which would allow the script to be executed before `pcfs` was loaded (as is done in `cdrom.d`).

```
1   #!/usr/sbin/dtrace -s
2
3   #pragma D option quiet
4   #pragma D option switchrate=10hz
5
```

continues

```
dtrace:::BEGIN
{
        printf("%-20s %1s %4s %6s %3s %s\n", "TIME", "D", "KB",
            "ms", "ERR", "PATH");
}

fbt::pcfs_read:entry, fbt::pcfs_write:entry, fbt::pcfs_readdir:entry
{
        self->path = args[0]->v_path;
        self->kb = args[1]->uio_resid / 1024;
        self->start = timestamp;
}

fbt::pcfs_read:return, fbt::pcfs_write:return, fbt::pcfs_readdir:return
/self->start/
{
        this->iotime = (timestamp - self->start) / 1000000;
        this->dir = probefunc == "pcfs_read" ? "R" : "W";
        printf("%-20Y %1s %4d %6d %3d %s\n", walltimestamp,
            this->dir, self->kb, this->iotime, arg1,
            self->path != NULL ? stringof(self->path) : "<null>");
        self->start = 0; self->path = 0; self->kb = 0;
}
```

Script pcfsrw.d

This script prints basic information. To retrieve `pcfs`-specific information such as the FAT type, the `struct pcfs` can be retrieved from the vnode in the same way as at the start of the `pcfs_read()` function (see the source, including `VFSTOPCFS`). We've resisted including an example of this, since `struct pcfs` has changed between Solaris versions, and it would make this script much more fragile; add the appropriate code for your Solaris version.

HSFS Scripts

HSFS is the High Sierra File System (ISO 9660) driver on Oracle Solaris, used by CD-ROMs. In cases of unusual performance or errors such as failing to mount, DTrace can be used to examine the internal operation of the device driver using the fbt provider. On recent versions of Oracle Solaris, the kernel engineers have also placed sdt provider probes in hsfs for convenience:

```
solaris# dtrace -ln 'sdt:hsfs::'
   ID   PROVIDER         MODULE                        FUNCTION NAME
83019        sdt           hsfs             hsched_enqueue_io hsfs_io_enqueued
83020        sdt           hsfs          hsched_invoke_strategy hsfs_coalesced_io_
done
83021        sdt           hsfs          hsched_invoke_strategy hsfs_coalesced_io_
start
83022        sdt           hsfs          hsched_invoke_strategy hsfs_io_dequeued
83023        sdt           hsfs          hsched_invoke_strategy hsfs_deadline_expiry
83024        sdt           hsfs                  hsfs_getpage hsfs_compute_ra
83025        sdt           hsfs                  hsfs_getapage hsfs_io_done
83026        sdt           hsfs                  hsfs_getapage hsfs_io_wait
```

```
83027        sdt           hsfs                   hsfs_getpage_ra hsfs_readahead
83028        sdt           hsfs                      hsfs_ra_task hsfs_io_done_ra
83029        sdt           hsfs                      hsfs_ra_task hsfs_io_wait_ra
83030        sdt           hsfs                        hs_mountfs rootvp-failed
83031        sdt           hsfs                        hs_mountfs mount-done
[...]
```

The `*_ra` probes shown previously refer to read-ahead, a feature of the hsfs driver to request data ahead of time to prewarm the cache and improve performance (similar to UFS read-ahead).

Since there is currently no HSFS provider, the options are to use the fbt provider to examine driver internals; use the sdt provider (if present), because it has probe locations that were deliberately chosen for tracing with DTrace; or use the stable io provider by filtering on the CD-ROM device. For robust scripts, the best option is the io provider; the others instrument a particular operating system and version and may require modifications to match the software version you are using.

cdrom.d

The `cdrom.d` script traces the `hs_mountfs()` call via the fbt provider, showing hsfs mounts along with the mount path, error status, and mount time.

Script

The `-Z` option is used on line 1 because the hsfs driver may not yet be loaded, and the functions to probe may not yet be in memory. Once a CD-ROM is inserted, the hsfs driver is automounted.

```
1   #!/usr/sbin/dtrace -Zs
2
3   #pragma D option quiet
4   #pragma D option switchrate=10hz
5
6   dtrace:::BEGIN
7   {
8           trace("Tracing hsfs (cdrom) mountfs...\n");
9   }
10
11  fbt::hs_mountfs:entry
12  {
13          printf("%Y:  Mounting %s... ", walltimestamp, stringof(arg2));
14          self->start = timestamp;
15  }
16
17  fbt::hs_mountfs:return
18  /self->start/
19  {
20          this->time = (timestamp - self->start) / 1000000;
21          printf("result: %d%s, time: %d ms\n", arg1,
22              arg1 ? "" : " (SUCCESS)", this->time);
```

continues

```
            self->start = 0;
    }

Script cdrom.d
```

Example

Here's a CD-ROM with the label "Photos001" inserted:

```
solaris# cdrom.d
Tracing hsfs (cdrom) mountfs...
2010 Jun 20 23:40:59:  Mounting /media/Photos001... result: 0 (SUCCESS), time: 157 ms
```

Several seconds passed between CD-ROM insertion and the mount initiating, as shown by `cdrom.d`. This time can be understood with more DTrace.

For example, the operation of volume management and hardware daemons can be traced (`vold(1M)`, `rmvolmgr(1M)`, `hald(1M)`, ...). Try starting this investigation with process execution:

```
solaris# dtrace -qn 'proc:::exec-success { printf("%Y %s\n", walltimestamp,
curpsinfo->pr_psargs); }'
2010 Jun 21 23:51:48 /usr/lib/hal/hald-probe-storage --only-check-for-media
2010 Jun 21 23:51:48 /usr/lib/hal/hald-probe-volume
2010 Jun 21 23:51:50 /usr/lib/hal/hal-storage-mount
2010 Jun 21 23:51:50 /sbin/mount -F hsfs -o nosuid,ro /dev/dsk/c0t0d0s2 /media/Photos0
01
2010 Jun 21 23:51:50 mount -o nosuid,ro /dev/dsk/c0t0d0s2 /media/Photos001
^C
```

The same CD-ROM was reinserted, and the HAL processes that executed to mount the CD-ROM can now be seen. DTrace can be used to further examine whether these events were triggered by a hardware interrupt (media insertion) or by polling.

UDFS Scripts

UDFS is the Universal Disk Format file system driver on Oracle Solaris, used by DVDs. This driver can be examined using DTrace in a similar way to HSFS.

dvd.d

Since the source code functions between hsfs and udfs are similar, only three lines need to be changed to `cdrom.d` for it to trace DVDs instead:

```
        trace("Tracing udfs (dvd) mountfs...\n");
fbt::ud_mountfs:entry
fbt::ud_mountfs:return
```

The output printed for mounts is the same as `cdrom.d`.

NFS Client Scripts

Chapter 7, Network Protocols, covers tracing from the NFS server. The NFS client can also be traced, which we will cover here in this chapter because the NFS mount from a client perspective behaves like any other file system. Because of this, physical (network device) I/O to serve that file system can be traced by the io provider (currently Oracle Solaris only), just like tracing physical (storage device) I/O for a local file system.

Physical I/O is not the only I/O we can use to analyze NFS client performance. Logical I/O to the NFS client driver is also interesting and may be served without performing network I/O to the NFS server—for example, when returning data from a local NFS client cache.

For kernel-based NFS drivers, all internals can be examined using the fbt provider. fbt instruments a particular operating system and version, so these scripts may therefore require modifications to match the software version you are using.

nfswizard.d

This script from the DTraceToolkit demonstrates using the io provider on Oracle Solaris to trace and summarize NFS client I/O. It traces back-end I/O only: those that trigger NFS network I/O. More I/O may be performed to the NFS share from the client, which is returned from the client cache only.

Script

This is a neat example of how you can produce a sophisticated report from basic D syntax:

```
#!/usr/sbin/dtrace -s
[...]
#pragma D option quiet

dtrace:::BEGIN
{
        printf("Tracing... Hit Ctrl-C to end.\n");
        scriptstart = walltimestamp;
        timestart = timestamp;
}
```

continues

```
io:nfs::start
{
        /* tally file sizes */
        @file[args[2]->fi_pathname] = sum(args[0]->b_bcount);

        /* time response */
        start[args[0]->b_addr] = timestamp;

        /* overall stats */
        @rbytes = sum(args[0]->b_flags & B_READ ? args[0]->b_bcount : 0);
        @wbytes = sum(args[0]->b_flags & B_READ ? 0 : args[0]->b_bcount);
        @events = count();
}

io:nfs::done
/start[args[0]->b_addr]/
{
        /* calculate and save response time stats */
        this->elapsed = timestamp - start[args[0]->b_addr];
        @maxtime = max(this->elapsed);
        @avgtime = avg(this->elapsed);
        @qnztime = quantize(this->elapsed / 1000);
}

dtrace:::END
{
        /* print header */
        printf("NFS Client Wizard. %Y -> %Y\n\n", scriptstart, walltimestamp);

        /* print read/write stats */
        printa("Read:  %@d bytes ", @rbytes);
        normalize(@rbytes, 1000000);
        printa("(%@d Mb)\n", @rbytes);
        printa("Write: %@d bytes ", @wbytes);
        normalize(@wbytes, 1000000);
        printa("(%@d Mb)\n\n", @wbytes);

        /* print throughput stats */
        denormalize(@rbytes);
        normalize(@rbytes, (timestamp - timestart) / 1000000);
        printa("Read:  %@d Kb/sec\n", @rbytes);
        denormalize(@wbytes);
        normalize(@wbytes, (timestamp - timestart) / 1000000);
        printa("Write: %@d Kb/sec\n\n", @wbytes);

        /* print time stats */
        printa("NFS I/O events:    %@d\n", @events);
        normalize(@avgtime, 1000000);
        printa("Avg response time: %@d ms\n", @avgtime);
        normalize(@maxtime, 1000000);
        printa("Max response time: %@d ms\n\n", @maxtime);
        printa("Response times (us):%@d\n", @qnztime);

        /* print file stats */
        printf("Top 25 files accessed (bytes):\n");
        printf("   %-64s %s\n", "PATHNAME", "BYTES");
        trunc(@file, 25);
        printa("   %-64s %@d\n", @file);
}
```

Script nfswizard.d

The io provider is used to trace client NFS I/O only, by including `nfs` in the probe module field. This is technically an unstable field of the probe name, although it's also unlikely to be renamed any time soon. An alternate approach would be to trace all io probes and use a predicate to match when `args[1]->dev_name` was equal to `nfs`. See the io provider description in Chapter 4 for more discussion about matching this field for io probes.

Example

Here `nfswizard.d` was run for a few seconds while a `tar(1)` command archived files from an NFSv4 share:

```
client# nfswizard.d
Tracing... Hit Ctrl-C to end.
^C
NFS Client Wizard. 2010 Jun 22 05:32:23 -> 2010 Jun 22 05:32:26

Read:  56991744 bytes (56 Mb)
Write: 0 bytes (0 Mb)

Read:  18630 Kb/sec
Write: 0 Kb/sec

NFS I/O events:    1747
Avg response time: 2 ms
Max response time: 59 ms

Response times (us):
           value  ------------- Distribution ------------- count
             128 |                                         0
             256 |                                         1
             512 |@@@@@                                    221
            1024 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@         1405
            2048 |@                                        37
            4096 |                                         21
            8192 |@                                        31
           16384 |                                         19
           32768 |                                         12
           65536 |                                         0

Top 25 files accessed (bytes):
   PATHNAME                                                         BYTES
   /net/mars/export/home/brendan/Downloads/ping.tar                 40960
   /net/mars/export/home/brendan/Downloads/pkg_get.pkg              69632
   /net/mars/export/home/brendan/Downloads/procps-3.2.8.tar.gz      286720
   /net/mars/export/home/brendan/Downloads/psh-i386-40              2260992
   /net/mars/export/home/brendan/Downloads/proftpd-1.3.2c.tar.gz    3174400
   /net/mars/export/home/brendan/Downloads/perlsrc-5.8.8stable.tar  51159040
```

The output includes a distribution plot of response times, which includes network latency and NFS server latency—which may return from cache (fast) or disk (slow), depending on the I/O.

nfs3sizes.d

This script shows both logical (local) and physical (network) reads by an Oracle Solaris NFSv3 client, showing requested read size distributions and total bytes. It can be used as a starting point to investigate.

- **Client caching**: The nfs client driver performs caching (unless it is directed not to, such as with the `forcedirectio` mount option), meaning that many of the logical reads may return from the client's DRAM without performing a (slower) NFS read to the server.
- **Read size**: The nfs client driver read size may differ from the application read size on NFS files (this can be tuned to a degree using the `rsize` mount option).

Script

The `nfs3_read()` function is the VFS interface into the NFSv3 client driver, which is traced to show requested NFS reads. The `nfs3_getpage()` and `nfs3_directio_read()` functions perform NFSv3 network I/O.

```
#!/usr/sbin/dtrace -s

#pragma D option quiet

dtrace:::BEGIN
{
        trace("Tracing NFSv3 client file reads... Hit Ctrl-C to end.\n");
}

fbt::nfs3_read:entry
{
        @q["NFS read size (bytes)"] = quantize(args[1]->uio_resid);
        @s["NFS read (bytes)"] = sum(args[1]->uio_resid);
}

fbt::nfs3_directio_read:entry
{
        @q["NFS network read size (bytes)"] = quantize(args[1]->uio_resid);
        @s["NFS network read (bytes)"] = sum(args[1]->uio_resid);
}

fbt::nfs3_getpage:entry
{
        @q["NFS network read size (bytes)"] = quantize(arg2);
        @s["NFS network read (bytes)"] = sum(arg2);
}
```

Script nfs3sizes.d

This script traces the size of read requests. The size of the returned data may be smaller than was requested, or zero if the read failed; the script could be enhanced to trace the returned data size instead if desired.

Example

An application performed random 1KB reads on a file shared over NFSv3:

```
client# nfssizes.d
Tracing NFSv3 client file reads... Hit Ctrl-C to end.
^C

  NFS network read size (bytes)
           value  ------------- Distribution ------------- count
            2048 |                                         0
            4096 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 2564
            8192 |                                         2
           16384 |                                         0

  NFS read size (bytes)
           value  ------------- Distribution ------------- count
             128 |                                         0
             256 |                                         1
             512 |                                         0
            1024 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 147083
            2048 |                                         0

  NFS network read (bytes)                                    10518528
  NFS read (bytes)                                           150613423
```

In this example, there were many more logical NFS reads (147,084) than physical network reads (2,566) to the NFS server, suggesting that the NFS client cache is serving most of these logical reads (high client cache hit rate). The difference between logical and physical read size distribution can also be compared, which shows that the nfs client driver is requesting 4+KB reads to satisfy 1+KB requests. Both of these behaviors can be investigated further by DTracing more internals from the nfs client driver.

nfs3fileread.d

This script shows both logical and physical (network) reads by an Oracle Solaris NFSv3 client, showing the requested and network read bytes by filename. This is a variation of the `nfs3sizes.d` script explained previously.

Script

```
#!/usr/sbin/dtrace -s

#pragma D option quiet

dtrace:::BEGIN
{
        trace("Tracing NFSv3 client file reads... Hit Ctrl-C to end.\n");
}

```

continues

```
fbt::nfs3_read:entry
{
        this->path = args[0]->v_path;
        this->bytes = args[1]->uio_resid;
        @r[this->path ? stringof(this->path) : "<null>"] = sum(this->bytes);
}

fbt::nfs3_directio_read:entry
{
        this->path = args[0]->v_path;
        this->bytes = args[1]->uio_resid;
        @n[this->path ? stringof(this->path) : "<null>"] = sum(this->bytes);
}

fbt::nfs3_getpage:entry
{
        this->path = args[0]->v_path;
        this->bytes = arg2;
        @n[this->path ? stringof(this->path) : "<null>"] = sum(this->bytes);
}

dtrace:::END
{
        printf(" %-56s %10s %10s\n", "FILE", "READ(B)", "NET(B)");
        printa(" %-56s %@10d %@10d\n", @r, @n);
}
```

Script nfs3fileread.d

Example

All of the files read were 10MB in size and were read sequentially.

```
client# nfs3fileread.d
Tracing NFSv3 client file reads... Hit Ctrl-C to end.
^C
 FILE                                                        READ(B)     NET(B)
 /saury-data-0/10m_d                                         4182016    1265216
 /saury-data-0/10m_a                                        10493952          0
 /saury-data-0/10m_c                                        10493952   10485760
 /saury-data-0/10m_b                                        43753984   10485760
```

The difference between the `READ` (requested read bytes) and `NET` (network read bytes) columns are because of the following.

- `10m_d`: About 4MB was read from this file, which was partially cached.
- `10m_a`: This file was entirely cached in the client's DRAM and was read through once.
- `10m_c`: This file was entirely uncached and was read through once from the NFS server.
- `10m_b`: This file was entirely uncached and was read through multiple times—the first reading it from the NFS server.

TMPFS Scripts

tmpfs is a file system type for temporary files that attempts to reside in memory for fast access. It's used by Oracle Solaris for `/tmp` and other directories. The performance of `/tmp` can become a factor when tmpfs contains more data than can fit in memory, and it begins paging to the swap devices.

tmpfs activity can be traced at other levels such as the syscall interface and VFS. The scripts in this section demonstrate examining activity from the kernel tmpfs driver, using the fbt provider. fbt instruments a particular operating system and version, so these scripts may therefore require modifications to match the software version you are using. You shouldn't have too much difficulty rewriting them to trace at syscall or VFS instead if desired and to match only activity to `/tmp` or tmpfs.

tmpusers.d

This script shows who is using tmpfs on Oracle Solaris by tracing the user, process, and filename for tmpfs open calls.

Script

```
#!/usr/sbin/dtrace -s

#pragma D option quiet

dtrace:::BEGIN
{
        printf("%6s %6s %-16s %s\n", "UID", "PID", "PROCESS", "FILE");
}

fbt::tmp_open:entry
{
        printf("%6d %6d %-16s %s\n", uid, pid, execname,
            stringof((*args[0])->v_path));
}
```

Script tmpusers.d

Example

Here's an example:

```
solaris# tmpusers.d
   UID    PID PROCESS          FILE
     0     47 svc.configd      /etc/svc/volatile/svc_nonpersist.db-journal
     0     47 svc.configd      /etc/svc/volatile
     0     47 svc.configd      /etc/svc/volatile/sqlite_UokyAO1gmAy2L8H
     0     47 svc.configd      /etc/svc/volatile/svc_nonpersist.db-journal
```

continues

```
      0     47 svc.configd      /etc/svc/volatile
      0     47 svc.configd      /etc/svc/volatile/sqlite_Ws9dGwSvZRtutXk
      0     47 svc.configd      /etc/svc/volatile/svc_nonpersist.db-journal
      0     47 svc.configd      /etc/svc/volatile/sqlite_zGn0Ab6VUI6IFpr
[...]
      0   1367 sshd             /etc/svc/volatile/etc/ssh/sshd_config
      0   1368 sshd             /var/run/sshd.pid
```

tmpgetpage.d

This script shows which processes are actively reading from tmpfs files by tracing the tmpfs getpage routine, which is the interface to read pages of data. The time spent in `getpage` is shown as a distribution plot.

Script

```
#!/usr/sbin/dtrace -s

#pragma D option quiet

dtrace:::BEGIN
{
      trace("Tracing tmpfs disk read time (us):\n");
}

fbt::tmp_getpage:entry
{
      self->vp = args[0];
      self->start = timestamp;
}

fbt::tmp_getpage:return
/self->start/
{
      @[execname, stringof(self->vp->v_path)] =
          quantize((timestamp - self->start) / 1000);
      self->vp = 0;
      self->start = 0;
}
```

Script tmpgetpage.d

Example

Here the `cksum(1)` command was reading a file that was partially in memory. The time for `getpage` shows two features: fast I/O between 0 us and 4 us and slower I/O mostly between 128 us and 1024 us. These are likely to correspond to reads from DRAM or from disk (swap device). If desired, the script could be enhanced to trace disk I/O calls so that a separate distribution plot could be printed for DRAM reads and disk reads.

```
solaris# tmpgetpage.d
Tracing tmpfs disk read time (us):
^C

  cksum                                     /tmp/big0

           value  ------------- Distribution ------------- count
               0 |                                         0
               1 |@@@@@@@@@@@@@@@@@@@@                     9876
               2 |@@@@@@@@@@                               5114
               4 |                                         29
               8 |                                         48
              16 |@                                        354
              32 |                                         120
              64 |                                         19
             128 |@                                        317
             256 |@@@@@@@                                  3223
             512 |@                                        444
            1024 |                                         71
            2048 |                                         31
            4096 |                                         37
            8192 |                                         33
           16384 |                                         23
           32768 |                                         4
           65536 |                                         2
          131072 |                                         0
```

Case Study

Here we present the application of the DTrace commands, scripts, and methods discussed in this chapter.

ZFS 8KB Mirror Reads

This case study looks at a ZFS workload doing 8KB reads from a mirrored zpool.

- **System**:
  - **7410**: 4 AMD Barcelona CPUs, 128GB DRAM, one 10Gb Ethernet port
  - **1 JBOD**: 22 1TB disks, 2 Logzillas, mirroring
  - **ZFS**: 10 shares, 8KB record size
- **Workload**:
  - NFSv3 streaming reads, 1MB I/O size
  - 100 threads total, across 10 clients (10 threads per client)
  - 200+GB working set, mostly uncached
- **Clients**:
  - 10 blades

Total throughput for this workload is 338MB/sec. The 10Gb Ethernet port has a theoretical maximum throughput of 1.16GB/sec, so what is holding us back? Disk I/O latency? CPU?

Basic Observability

Operating system tools commonly used to check system performance include `vmstat(1M)`, `mpstat(1M)`, and `iostat(1M)`. Running these

```
# vmstat 5
 kthr      memory            page            disk          faults      cpu
 r b w   swap  free  re  mf pi po fr de sr s6 s7 s1 s1   in   sy   cs us sy id
 0 0 0 129657948 126091808 13 13 0 0 0 0 2  4  4 19  3 3088 2223  990  0  1 99
 8 0 0 7527032 3974064 0 42  0  0  0  0  0  2  1  0 303 570205 2763 100141 0 62 37
 7 0 0 7527380 3974576 0  7  0  0  0  0  0  0  0  0 309 561541 2613 99200 0 62 38
 6 0 0 7526472 3973564 0  4  0  0  0  0  0  0  0  0 321 565225 2599 101515 0 62 37
 7 0 0 7522756 3970040 11 85 0  0  0  0  0  7  7  0 324 573568 2656 99129 0 63 37
[...]
```

`vmstat(1M)` shows high sys time (62 percent).

```
# mpstat 5
CPU minf mjf xcal  intr ithr  csw icsw migr smtx  srw syscl  usr sys  wt idl
[...summary since boot truncated...]
CPU minf mjf xcal  intr ithr  csw icsw migr smtx  srw syscl  usr sys  wt idl
  0    0   0 21242 34223  205 5482    2 1669 7249    0    28    0  58   0  42
  1    0   0 27446 30002  113 4574    2 1374 7029    0  1133    1  53   0  46
  2    0   0 198422 31967 2951 20938    3  213 2655    0    27    0  97   0   3
  4    0   0 16970 39203 3082 3866    9  829 6695    0    55    0  59   0  40
  5    4   0 24698 33998   10 5492    3 1066 7492    0    43    0  57   0  43
  6    0   0 26184 41790   11 7412    1 1568 6586    0    15    0  67   0  33
  7   14   0 17345 41765    9 4797    1  943 5764    1    98    0  65   0  35
  8    5   0 17756 36776   37 6110    4 1183 7698    0    62    0  58   0  41
  9    0   0 17265 31631    9 4608    2  877 7784    0    37    0  53   0  47
 10    2   0 24961 34622    7 5553    1 1022 7057    0   109    1  57   0  42
 11    3   0 33744 40631   11 8501    4 1742 6755    0    72    1  65   0  35
 12    2   0 27320 42180  468 7710   18 1620 7222    0   381    0  65   0  35
 13    1   0 20360 63074 15853 5154   28 1095 6099    0    36    1  72   0  27
 14    1   0 13996 31832    9 4277    8  878 7586    0    36    0  52   0  48
 15    8   0 19966 36656    5 5646    7 1054 6703    0   633    2  56   0  42
[...]
```

`mpstat(1M)` shows CPU 2 is hot at 97 percent sys, and we have frequent cross calls (`xcals`), especially on CPU 2.

```
# iostat -xnz 5
                    extended device statistics
    r/s    w/s   kr/s   kw/s wait actv wsvc_t asvc_t  %w  %b device
[...summary since boot truncated...]
                    extended device statistics
    r/s    w/s   kr/s   kw/s wait actv wsvc_t asvc_t  %w  %b device
    0.2   23.4   12.8 1392.7  0.5  0.1   20.3    2.3   6   5 c3t0d0
```

```
    0.0   22.4    0.0 1392.7  0.5  0.0   22.3    1.7   6   4 c3t1d0
  324.8    0.0 21946.8    0.0  0.0  4.7    0.0   14.4   1  79 c4t5000C5001073ECF5d0
  303.8    0.0 19980.0    0.0  0.0  4.0    0.0   13.1   1  75 c4t5000C50010741BF9d0
  309.8    0.0 22036.5    0.0  0.0  5.3    0.0   17.0   1  82 c4t5000C5001073ED34d0
  299.6    0.0 19944.1    0.0  0.0  4.4    0.0   14.7   1  76 c4t5000C5000D416FFEd0
  302.6    0.0 20229.0    0.0  0.0  4.4    0.0   14.4   1  77 c4t5000C50010741A8Ad0
  292.2    0.0 19198.3    0.0  0.0  4.0    0.0   13.8   1  74 c4t5000C5000D416E2Ed0
  305.6    0.0 21203.4    0.0  0.0  4.5    0.0   14.8   1  80 c4t5000C5001073DEB9d0
  280.8    0.0 18160.5    0.0  0.0  4.0    0.0   14.3   1  75 c4t5000C5001073E602d0
  304.2    0.0 19574.9    0.0  0.0  4.3    0.0   14.2   1  77 c4t5000C50010743CFAd0
  322.0    0.0 21906.5    0.0  0.0  5.1    0.0   15.8   1  80 c4t5000C5001073F2F8d0
  295.8    0.0 20115.4    0.0  0.0  4.6    0.0   15.7   1  77 c4t5000C5001073F440d0
  289.2    0.0 20836.0    0.0  0.0  4.6    0.0   16.0   1  75 c4t5000C5001073E2F4d0
  278.6    0.0 18159.2    0.0  0.0  3.8    0.0   13.6   1  73 c4t5000C5001073D840d0
  286.4    0.0 21366.9    0.0  0.0  5.0    0.0   17.5   1  79 c4t5000C5001073ED40d0
  307.6    0.0 19198.1    0.0  0.0  4.2    0.0   13.5   1  74 c4t5000C5000D416F21d0
  292.4    0.0 19045.3    0.0  0.0  4.2    0.0   14.2   1  76 c4t5000C5001073E593d0
  293.2    0.0 20590.0    0.0  0.0  5.2    0.0   17.7   1  81 c4t5000C50010743BD1d0
  317.2    0.0 21036.5    0.0  0.0  3.9    0.0   12.4   1  74 c4t5000C5000D416E76d0
  295.6    0.0 19540.1    0.0  0.0  4.0    0.0   13.5   1  72 c4t5000C5001073DDB4d0
  332.6    0.0 21610.2    0.0  0.0  4.2    0.0   12.5   1  75 c4t5000C500106CF55Cd0
[...]
```

iostat(1M) shows the disks are fairly busy (77 percent).

Just based on this information, there is little we can do to improve performance except upgrade to faster CPUs and faster disks. We could also check kernel tuning parameters to prevent CPU 2 from running hot, but at this point we don't even know *why* it is hot. It could be the cross cals, but we can't tell for certain that they are responsible for the high sys time. Without DTrace, we've hit a brick wall.

Enter DTrace

First we'll use DTrace to check high system time by profiling kernel stacks on-CPU and for the hot CPU 2:

```
# dtrace -n 'profile-1234 { @[stack()] = count(); } tick-5sec { exit(0); }'
dtrace: description 'profile-1234 ' matched 2 probes
CPU     ID                    FUNCTION:NAME
 11  85688                       :tick-5sec
[...output truncated...]


              unix`0xfffffffffb84fd8a
              zfs`zio_done+0x383
              zfs`zio_execute+0x89
              genunix`taskq_thread+0x1b7
              unix`thread_start+0x8
             2870

              unix`do_splx+0x80
              unix`xc_common+0x231
              unix`xc_call+0x46
              unix`hat_tlb_inval+0x283
              unix`x86pte_inval+0xaa
              unix`hat_pte_unmap+0xfd
              unix`hat_unload_callback+0x193
```

continues

```
              unix`hat_unload+0x41
              unix`segkmem_free_vn+0x6f
              unix`segkmem_free+0x27
              genunix`vmem_xfree+0x104
              genunix`vmem_free+0x29
              genunix`kmem_free+0x20b
              genunix`dblk_lastfree_oversize+0x69
              genunix`dblk_decref+0x64
              genunix`freeb+0x80
              ip`tcp_rput_data+0x25a6
              ip`squeue_enter+0x330
              ip`ip_input+0xe31
              mac`mac_rx_soft_ring_drain+0xdf
             3636

              unix`mach_cpu_idle+0x6
              unix`cpu_idle+0xaf
              unix`cpu_idle_adaptive+0x19
              unix`idle+0x114
              unix`thread_start+0x8
            30741
```

This shows that we are hottest in `do_splx()`, a function used to process cross calls (see `xc_call()` further down the stack).

Now we check the hot stacks for CPU 2, by matching it in a predicate:

```
# dtrace -n 'profile-1234 /cpu == 2/ { @[stack()] = count(); }
tick-5sec { exit(0); }'
dtrace: description 'profile-1234 ' matched 2 probes
CPU     ID                    FUNCTION:NAME
  8  85688                       :tick-5sec
[...output truncated...]

              unix`do_splx+0x80
              unix`xc_common+0x231
              unix`xc_call+0x46
              unix`hat_tlb_inval+0x283
              unix`x86pte_inval+0xaa
              unix`hat_pte_unmap+0xfd
              unix`hat_unload_callback+0x193
              unix`hat_unload+0x41
              unix`segkmem_free_vn+0x6f
              unix`segkmem_free+0x27
              genunix`vmem_xfree+0x104
              genunix`vmem_free+0x29
              genunix`kmem_free+0x20b
              genunix`dblk_lastfree_oversize+0x69
              genunix`dblk_decref+0x64
              genunix`freeb+0x80
              ip`tcp_rput_data+0x25a6
              ip`squeue_enter+0x330
              ip`ip_input+0xe31
              mac`mac_rx_soft_ring_drain+0xdf
             1370
```

This shows that CPU 2 is indeed hot in cross calls. To quantify the problem, we could postprocess this output to add up which stacks are cross calls and which aren't, to calculate the percentage of time spent in cross calls.

Sometimes frequency counting the kernel function name that is on-CPU is sufficient to identify the activity, instead of counting the entire stack:

```
# dtrace -n 'profile-1234 /cpu == 2/ { @[func(arg0)] = count(); }
tick-5sec { exit(0); }'
dtrace: description 'profile-1234 ' matched 2 probes
CPU     ID                    FUNCTION:NAME
  1  85688                       :tick-5sec

  mac`mac_hwring_tx                                                   1
  mac`mac_soft_ring_worker_wakeup                                     1
  mac`mac_soft_ring_intr_disable                                      1
  rootnex`rootnex_init_win                                            1
  scsi_vhci`vhci_scsi_init_pkt                                        1
[...output truncated...]
  unix`setbackdq                                                     31
  ip`ip_input                                                        33
  unix`atomic_add_64                                                 33
  unix`membar_enter                                                  38
  unix`page_numtopp_nolock                                           47
  unix`0xfffffffffb84fd8a                                            50
  unix`splr                                                          56
  genunix`ddi_dma_addr_bind_handle                                   56
  unix`i_ddi_vaddr_get64                                             62
  unix`ddi_get32                                                     81
  rootnex`rootnex_coredma_bindhdl                                    83
  nxge`nxge_start                                                    92
  unix`mutex_delay_default                                           93
  unix`mach_cpu_idle                                                106
  unix`hat_tlb_inval                                                126
  genunix`biodone                                                   157
  unix`mutex_enter                                                  410
  unix`do_splx                                                     2597
```

This output is easier to examine and still identifies the cross call samples as the hottest CPU activity (`do_splx()` function). By postprocessing the sample counts (summing the count column using `awk(1)`), we found that CPU 2 spent 46 percent of its time in `do_splx()`, which is a significant percentage of time.

Investigating Cross Calls

CPU cross calls can be probed using DTrace directly:

```
# dtrace -n 'sysinfo:::xcalls { @[stack()] = count(); } tick-5sec { exit(0); }'
dtrace: description 'sysinfo:::xcalls ' matched 2 probes
CPU     ID                    FUNCTION:NAME
 10  85688                       :tick-5sec
[...output truncated...]

              unix`xc_call+0x46
              unix`hat_tlb_inval+0x283
              unix`x86pte_inval+0xaa
              unix`hat_pte_unmap+0xfd
              unix`hat_unload_callback+0x193
              unix`hat_unload+0x41
```

continues

```
              unix`segkmem_free_vn+0x6f
              unix`segkmem_free+0x27
              genunix`vmem_xfree+0x104
              genunix`vmem_free+0x29
              genunix`kmem_free+0x20b
              genunix`dblk_lastfree_oversize+0x69
              genunix`dblk_decref+0x64
              genunix`freemsg+0x84
              nxge`nxge_txdma_reclaim+0x396
              nxge`nxge_start+0x327
              nxge`nxge_tx_ring_send+0x69
              mac`mac_hwring_tx+0x20
              mac`mac_tx_send+0x262
              mac`mac_tx_soft_ring_drain+0xac
           264667

              unix`xc_call+0x46
              unix`hat_tlb_inval+0x283
              unix`x86pte_inval+0xaa
              unix`hat_pte_unmap+0xfd
              unix`hat_unload_callback+0x193
              unix`hat_unload+0x41
              unix`segkmem_free_vn+0x6f
              unix`segkmem_free+0x27
              genunix`vmem_xfree+0x104
              genunix`vmem_free+0x29
              genunix`kmem_free+0x20b
              genunix`dblk_lastfree_oversize+0x69
              genunix`dblk_decref+0x64
              genunix`freeb+0x80
              ip`tcp_rput_data+0x25a6
              ip`squeue_enter+0x330
              ip`ip_input+0xe31
              mac`mac_rx_soft_ring_drain+0xdf
              mac`mac_soft_ring_worker+0x111
              unix`thread_start+0x8
           579607
```

The most frequent stacks originate in either `ip` (the IP and TCP module) or `nxge` (which is the 10GbE network interface driver). Filtering on CPU 2 (`/cpu == 2/`) showed the same hottest stacks for these cross calls. Reading up the stack to understand the nature of these cross calls shows that they enter the kernel memory subsystem (*Solaris Internals* [McDougall and Mauro, 2006] is a good reference for understanding these).

Perhaps the most interesting stack line is `dblk_lastfree_oversize()`—oversize is the kernel memory allocator slab for large buffers. Although it is performing well enough, the other fixed-size slabs (8KB, 64KB, 128KB, and so on) perform better, so usage of oversize is undesirable if it can be avoided.

The cross call itself originates from a code path that is freeing memory, including functions such as `kmem_free()`. To better understand this cross call, the `kmem_free()` function is traced so that the size freed can be examined if this becomes a cross call on CPU 2:

```
# dtrace -n 'fbt::kmem_free:entry /cpu == 2/ { self->size = arg1; }
sysinfo:::xcalls /cpu == 2/ { @ = quantize(self->size); }'
dtrace: description 'fbt::kmem_free:entry ' matched 2 probes
^C


           value  ------------- Distribution ------------- count
          524288 |                                         0
         1048576 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 37391
         2097152 |                                         0
```

The output shows that the frees that become cross calls are in the 1MB to 2MB range.

This rings a bell. The clients are using a 1MB I/O size for their sequential reads, on the assumption that 1MB would be optimal. Perhaps it is these 1MB I/Os that are causing the use of the oversize kmem cache and the cross calls.

Trying the Solution

As an experiment, we changed I/O size on the clients to 128KB. Let's return to system tools for comparison:

```
# mpstat 5
CPU minf mjf xcal  intr ithr  csw icsw migr smtx  srw syscl  usr sys  wt idl
[...summary since boot truncated...]
CPU minf mjf xcal  intr ithr  csw icsw migr smtx  srw syscl  usr sys  wt idl
  0    0   0 2478  7196  205 10189    2 2998 3934    0    41    0  47   0  53
  1    0   0  139  6175  111 9367    2 2713 3714    0    84    0  44   0  56
  2    0   0 10107 11434 3610 54281   11 1476 2329    0   465    1  79   0  20
  4    7   0   36  7924 3703 6027   11 1412 5085    0   146    1  54   0  46
  5    0   0    4  5647   10 8028    3 1793 4347    0    28    0  53   0  47
  6    1   0   49  6984   12 12248    2 2863 4374    0    38    0  56   0  44
  7    0   0   11  4472   10 7687    3 1730 3730    0    33    0  49   0  51
  8    0   0   82  5686   42 9783    2 2132 5116    0   735    1  49   0  49
  9    0   0   39  4339    7 7308    1 1511 4898    0   396    1  43   0  57
 10    0   0    3  5256    4 8997    1 1831 4399    0    22    0  43   0  57
 11    0   0    5  7865   12 13900    1 3080 4365    1    43    0  55   0  45
 12    0   0   58  6990  143 12108   12 2889 5199    0   408    1  56   0  43
 13    1   0    0 35884 32388 6724   48 1536 4032    0    77    0  73   0  27
 14    1   0   14  3936    9 6473    6 1472 4822    0   102    1  42   0  58
 15    3   0    8  5025    8 8460    8 1784 4360    0   105    2  42   0  56
[...]
```

The cross calls have mostly vanished, and throughput is 503MB/sec—a 49 percent improvement!

```
# iostat -xnz 5
                    extended device statistics
    r/s    w/s   kr/s   kw/s wait actv wsvc_t asvc_t  %w  %b device
[...summary since boot truncated...
                    extended device statistics
```

continues

```
    r/s    w/s   kr/s   kw/s wait actv wsvc_t asvc_t  %w  %b device
    0.2   45.6   12.8 3982.2  1.7  0.2   37.6    4.3  19  20 c3t0d0
    0.4   45.2   25.6 3982.2  1.3  0.1   28.7    2.9  15  13 c3t1d0
  381.8    0.0 21210.8    0.0  0.0  5.2    0.0   13.7   1  92 c4t5000C5001073ECF5d0
  377.2    0.0 21914.8    0.0  0.0  5.5    0.0   14.6   1  87 c4t5000C50010741BF9d0
  330.2    0.0 21334.7    0.0  0.0  6.4    0.0   19.3   1  89 c4t5000C5001073ED34d0
  379.8    0.0 21294.8    0.0  0.0  5.4    0.0   14.3   1  92 c4t5000C5000D416FFEd0
  345.8    0.0 21823.1    0.0  0.0  6.1    0.0   17.6   1  90 c4t5000C50010741A8Ad0
  360.6    0.0 20126.3    0.0  0.0  5.2    0.0   14.5   1  85 c4t5000C5000D416E2Ed0
  352.2    0.0 23318.3    0.0  0.0  6.9    0.0   19.7   1  93 c4t5000C5001073DEB9d0
  253.8    0.0 21745.3    0.0  0.0 10.0    0.0   39.3   0 100 c4t5000C5001073E602d0
  337.4    0.0 22797.5    0.0  0.0  7.1    0.0   20.9   1  96 c4t5000C50010743CFAd0
  346.0    0.0 22145.4    0.0  0.0  6.7    0.0   19.3   1  87 c4t5000C5001073F2F8d0
  350.0    0.0 20946.2    0.0  0.0  5.3    0.0   15.2   1  89 c4t5000C5001073F440d0
  383.6    0.0 22688.1    0.0  0.0  6.5    0.0   17.0   1  94 c4t5000C5001073E2F4d0
  333.4    0.0 24451.0    0.0  0.0  8.2    0.0   24.6   1  98 c4t5000C5001073D840d0
  337.6    0.0 21057.5    0.0  0.0  5.9    0.0   17.4   1  90 c4t5000C5001073ED40d0
  370.8    0.0 21949.1    0.0  0.0  5.3    0.0   14.2   1  88 c4t5000C5000D416F21d0
  393.2    0.0 22121.6    0.0  0.0  5.6    0.0   14.3   1  90 c4t5000C5001073E593d0
  354.4    0.0 22323.5    0.0  0.0  6.4    0.0   18.1   1  93 c4t5000C50010743BD1d0
  382.2    0.0 23451.7    0.0  0.0  5.9    0.0   15.3   1  95 c4t5000C5000D416E76d0
  357.4    0.0 22791.5    0.0  0.0  6.8    0.0   19.0   1  93 c4t5000C5001073DDB4d0
  338.8    0.0 22762.6    0.0  0.0  7.3    0.0   21.6   1  92 c4t5000C500106CF55Cd0
[...]
```

The disks are now reaching 100 percent busy and have become the new bottleneck (one disk in particular). This often happens with performance investigations: As soon as one problem has been fixed, another one becomes apparent.

Analysis Continued

From the previous `iostat(1M)` output, it can be calculated that the average I/O size is fairly large (~60KB), yet this results in low throughput per disk (20MB/sec) for disks that can pull more than 80MB/sec. This could indicate a random component to the I/O. However, with DTrace, we can measure it directly.

Running `bitesize.d` from Chapter 4 (and the DTraceToolkit) yields the following:

```
# bitesize.d
Tracing... Hit Ctrl-C to end.

     PID  CMD
    1040  /usr/lib/nfs/nfsd -s /var/ak/rm/pool-0/ak/nas/nfs4\0

           value  ------------- Distribution ------------- count
            4096 |                                         0
            8192 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 8296
           16384 |                                         0

       0  sched\0

           value  ------------- Distribution ------------- count
             256 |                                         0
             512 |                                         8
            1024 |                                         51
            2048 |                                         65
            4096 |                                         25
            8192 |@@@@@@@@                                 5060
```

```
           16384 |@@@@                                     2610
           32768 |@@@@                                     2881
           65536 |@@@@@@@@@@@@@                            8576
          131072 |@@@@@@@@@@@                              7389
          262144 |                                         0
```

This shows I/O from 8KB through to 128KB. 8KB I/O is expected because of the ZFS record size and when nfsd responds to a request by reading 8KB I/O. Doing this sequentially will trigger ZFS prefetch, which will read ahead in the file asynchronously to the nfsd thread (`sched`). The vdev layer can aggregate these reads up to 128KB before they are sent to disk. All of these internals can be examined using DTrace.

Running `seeksize.d` from Chapter 4 (and the DTraceToolkit) yields the following:

```
# seeksize.d
Tracing... Hit Ctrl-C to end.

     PID  CMD
    1040  /usr/lib/nfs/nfsd -s /var/ak/rm/pool-0/ak/nas/nfs4\0

           value  ------------- Distribution ------------- count
              -1 |                                         0
               0 |@@@@@@@@@@@@@@@@@@@@@@@@                 5387
               1 |                                         1
               2 |                                         53
               4 |                                         3
               8 |                                         7
              16 |@@                                       450
              32 |@@                                       430
              64 |@                                        175
             128 |@                                        161
             256 |@                                        144
             512 |                                         97
            1024 |                                         49
            2048 |                                         10
            4096 |                                         19
            8192 |                                         34
           16384 |                                         84
           32768 |@                                        154
           65536 |@                                        307
          131072 |@@                                       528
          262144 |@@@                                      598
          524288 |@                                        266
         1048576 |                                         23
         2097152 |                                         0

       0  sched\0

           value  ------------- Distribution ------------- count
              -1 |                                         0
               0 |@@@@@                                    3160
               1 |                                         2
               2 |                                         38
               4 |                                         11
               8 |                                         3
              16 |                                         265
              32 |@                                        309
```

continues

```
              64 |@                                        442
             128 |@                                        528
             256 |@                                        767
             512 |@                                        749
            1024 |@                                        427
            2048 |                                         165
            4096 |                                         250
            8192 |@                                        406
           16384 |@                                        870
           32768 |@@@                                      1623
           65536 |@@@@@                                    2801
          131072 |@@@@@@@                                  4113
          262144 |@@@@@@@                                  4167
          524288 |@@@                                      1787
         1048576 |                                         141
         2097152 |                                         7
         4194304 |@                                        718
         8388608 |@                                        354
        16777216 |                                         0
```

This shows that the disks are often seeking to perform I/O. From this, we could look at how the files were created and what file system parameters existed to optimize placement in order to reduce seeking.

Running iopattern from Chapter 4 (and the DTraceToolkit) yields the following:

```
# iopattern
%RAN %SEQ  COUNT    MIN    MAX    AVG     KR     KW
  72   28  72996     36 131072  59152 4130835  85875
  70   30  71971    512 131072  61299 4217260  91147
  67   33  68096    512 131072  59652 3872788  94092
  63   37  72490     36 131072  60248 4173898  91155
  66   34  73607    512 131072  60835 4285085  95988
[...]
```

iopattern confirms the previous findings.

Finally, an `iolatency.d` script was written to show overall device latency as a distribution plot:

```
#!/usr/sbin/dtrace -s

io:::start
{
        start[arg0] = timestamp;
}

io:::done
/start[arg0]/
{
        @time["disk I/O latency (ns)"] = quantize(timestamp - start[arg0]);
      start[arg0] = 0;
}
```

Script iolatency.d

```
# iolatency.d -n 'tick-5sec { exit(0); }'
dtrace: script 'io-latency.d' matched 10 probes
dtrace: description 'tick-5sec ' matched 1 probe
CPU     ID                    FUNCTION:NAME
 15  85688                       :tick-5sec

  disk I/O latency (ns)
           value  ------------- Distribution ------------- count
           32768 |                                         0
           65536 |                                         1
          131072 |                                         259
          262144 |@                                        457
          524288 |@@                                       1330
         1048576 |@@@@                                     2838
         2097152 |@@@@@                                    4095
         4194304 |@@@@@@@                                  5303
         8388608 |@@@@@@@@@                                7460
        16777216 |@@@@@@@                                  5538
        33554432 |@@@@                                     3480
        67108864 |@@                                       1338
       134217728 |                                         147
       268435456 |                                         3
       536870912 |                                         0
```

The latency for these disk I/Os is fairly large, often exceeding 8 ms.
There are a few ways we might improve performance here.

- Tuning file system on-disk placement to promote sequential access, which should take I/O latency closer to 1 ms.
- Upgrading (or improving) caches by increasing the size of the Level 1 cache (the ARC, which is DRAM-based) or using a level-two cache (the ZFS L2ARC, which is SSD-based) to span more of the working set. The internal workings of these caches can also be examined.
- Faster disks.

Conclusion

In this case study, we've demonstrated using DTrace to solve one problem and gather data on the next. This isn't the end of the road for DTrace—we can continue to study the internals of file system on-disk placement using DTrace, as well as the workings of the level-one file system cache to hunt for suboptimalities.

Summary

In this chapter, DTrace was used to examine file system usage and internals. This was performed from different perspectives: at the system call layer, at the virtual file system (VFS) layer, and from the file system software itself. For performance investigations, at the ability to measure I/O latency from these different layers can be crucial for pinpointing the source of latency—whether that's from the file system or underlying devices. Characteristics of the file system workload were also measured, such as I/O types and filenames, to provide context for understanding what the file system is doing and why.