Using the Sprite File System Traces

John H. Hartman

Computer Science Division
Electrical Engineering and Computer Sciences
University of California
Berkeley, CA 94720
jhh@sprite.berkeley.edu

Version 1.0
May 6, 1993

1 Introduction

This document describes the traces of file system activity collected for "Measurements of a Distributed File System", Mary Baker et al., *Proceedings of the 13th Symposium on Operating System Principles*, Oct. 1991., pp 198-212. These traces can be obtained via anonymous ftp from the directory `pub/sosp-traces` on "sprite.berkeley.edu". Included in that directory is a library for accessing records from the trace files, a sample program that uses the library to access records, and a collection of header files containing structure definitions for the records. This paper is intended to describe both the contents of the trace files and how to use these resources to extract information from the traces.

2 Environment

The traces were collected on a cluster of about 40 SparcStation 1, Sun 3, DECstation 3100, and DECStation 5000/200 workstations, all diskless and most with 24 to 32 Mbytes of memory. The system had four file servers during each of the trace intervals, although some of the servers changed from interval to interval. The main file server, "allspice", contained most of the files in the system and had 128 Mbytes of main memory. The entire cluster was connected by a single Ethernet network. There were about 30 every-day users of the system, and another 40 people who used the system occasionally.

3 Trace File Organization

The traces were collected during five separate intervals. Three of these intervals were 48 hours long; the other two are 24 hours long. Together these represent the eight 24-hour traces described in the paper. The traces for each interval are stored in a subdirectory of `pub/sosp-traces`, of the form `traces`*`N`*, where *N* is an integer from one to five.

Separate trace files were generated for each server in the system. The trace files for a particular server are stored in a subdirectory named after the server within each of the trace interval directories. For example, the trace files for "allspice" collected during the first trace interval can be found in `pub/sosp-traces/traces1/allspice.` In these directories there is one trace file per 15-minute interval (approximately) of the trace. The name of the file reflects the time at which the file was created, e.g `allspice.May.10.13.01` was created at 1:01pm on May 10, 1991 and it contains the trace records for the server "allspice". The trace files are currently compressed using the UNIX `compress` utility. The library provided for accessing the trace files will automatically uncompress them for you (see Section 10).

To make it easier to deal with the large number of trace files we've put a list of the files for each interval in a file called `files` in each of the trace interval directories. For some intervals not all trace files are listed in `files`; this is because in some intervals we collected more traces than we wanted, but couldn't bear to throw them out. The list contains only those files that were used in our analysis. For traces one through three the intervals were 48 hours long, which we analyzed as two separate 24-hour intervals. The files `files.a` and `files.b` contain the list of files for each of these sub-intervals.

4 Trace File Format

A trace file contains a magic number (`TRACELOG_MAGIC`), a file header, and a number of records. The file header is defined in `sysStats.h`, and is shown in Figure 1. If you use the provided library to access the trace files then the header contents shouldn't concern you.

```
typedef struct Sys_TracelogHeader {
     int  unused1;        /* Unused. */
     int  unused2;        /* Unused. */
     int  machineID;      /* ID of traced machine.*/
     char kernel[SYS_TRACELOG_KERNELLEN];
                          /* Kernel that was running. */
     char machineType[SYS_TRACELOG_TYPELEN];
                          /* Machine type. */
     int  bootTime[2];    /* Time machine was booted. */
     int  unused3;        /* Unused. */
     int  traceDir[4];    /* FileID of the trace dir.*/
} Sys_TracelogHeader;
```

Figure 1: Trace file header format.

5 Record Header

Each record in a trace file begins with a record header, shown in Figure 2. The two low-order bytes of `flags` field contain the size of the record. The macro `TRACELOG_BYTEMASK` can be used to mask off these bytes. The `time` field contains the time at which the event occurred. The first integer contains seconds, the second contains microseconds. **Important note**: the time in the trace files is relative to when the machine was booted. In order to get the absolute time that the event occurred you have to add on the time stored in the trace file header. Once again, the provided library will handle these details for you. The `type` field indicates the type of record that follows the header. You can use the type field to figure out which structure definition in `sosp.h` should be used to access the fields of the record.

```
typedef struct RecordHeader {
     int  flags;    /* Record size, misc. flags. */
     int  time[2];  /* Timestamp (relative).*/
     int  type;     /* Type of record.*/
} RecordHeader;
```

Figure 2: Trace record header format**.**

6 Record Format

Following the record header in a trace file is the record information itself. The following sections describe the different types of records and their contents.

6.1 Open

Record type: `SOSP_OPEN`

Structure: `SospOpen`

An open event is generated each time a file is opened. This includes opens by the kernel as well as application programs. The former happens when the kernel opens binaries for execution and files for paging. The format of an open record is shown in Figure 3. `CurrentHostId` identifies the client machine that is opening the file. `HomeHostID` identifies the home host of the process that is opening the file. The home host of a process differs from its current host if the process has been migrated. Logically it is running on its home host, but physically it is running elsewhere. `Fid` identifies the file that is being opened, and `sid` identifies the stream that was created as a result of the open. The format of a `FileID` is described in Section 7. `EffID` and `realID` provide the effective and real user ids, respectively. `Mode` contains flag bits provided to the open routine. These flag bits are defined in `fs.h` and `kernel/fs.h.` Of particular importance is the `FS_USER` bit, which indicates that a user process is opening the file as opposed to the kernel opening a swap file, directory, etc. The `mode` flags also tell you if the file was opened for reading, writing, truncate, etc. `NumNowReading` indicates the number of clients that now have the file open for reading, including the client doing the current open. Note that this is the number of client machines that have the file open for reading, not the number of processes. `NumNowWriting` provides the number of clients that have the file open for writing. `Create` is the creation time of the file, and `modify` is the time of last modification, both measured in seconds since the epoch. `Size` is the size of the file in bytes. `Type` indicates the type of file that was opened, e.g. regular file, directory, symbolic link, etc. File types are defined in `fs.h`. `Cacheable` indicates whether or not the file can be cached by the client, or if all read and write requests must go through to the server.

```
typedef struct {
     RecordHeader   rec;      /* Record info.*/
     int       currentHostID; /* Host doing open.*/
     int       homeHostID;    /* Home host if migrated.*/
     FileID    fid;           /* File ID being opened.*/
     FileID    sid;           /* Stream created.*/
     int       effID;         /* Effective user ID.*/
     int       realID;        /* Real user ID.*/
     int       mode;          /* Open mode.*/
     int       numNowReading; /* # of clients with file
                               * open for reading after
                               * the open.*/
     int       numNowWriting; /* # of clients with file
                               * open for writing after
                               * the open.*/
     int       create;        /* Create time of file.*/
     int       size;          /* File size in bytes.*/
     int       modify;        /* Modify time of file. */
     int       type;          /* FS_FILE, FS_DIRECTORY, etc.
                               * See /usr/include/fs.h.*/
     int       cacheable;     /* Is file cacheable? */
} SospOpen;
```

Figure 3: Open record format.

6.2 Delete (Unlink)

Record type: `SOSP_DELETE`

Structure: `SospDelete`

This record is generated when a file is unlinked. This does not mean that the file was actually deleted from disk, since it may have been open at the time it was unlinked. See Section 6.15 for information on file deletion. X shows the format of an unlink record.

```
typedef struct {
     RecordHeader   rec;      /* Record info. */
     int       currentHostID; /* Host doing open. */
     int       homeHostID;    /* Home host if migrated. */
     FileID    fid;           /* File ID. */
     int       modify;        /* Modify time of file. */
     int       create;        /* Create time of file. */
     int       size;          /* Size of file to delete. */
} SospDelete;
```

Figure 4: Delete record format.

6.3 Make Link

Record type: `SOSP_MKLINK`

Structure: `SospMkLink`

This record is generated when a hard link is created. The format of a mklink record is shown in Figure 5. `Fid` indentifies the existing file that is being linked to.

```
typedef struct {
     RecordHeader   rec;       /* Record info. */
     int       currentHostID; /* Host doing mklink. */
     int       homeHostID;    /* Home host if migrated.*/
     FileID    fid;           /* File ID of link. */
} SospMkLink;
```

Figure 5: Mklink record format.

6.4 Set Attributes

Record type: `SOSP_SET_ATTR`

Structure: `SospSetAttr`

This record is created when a process sets the attributes of a file. Figure 6 illustrates the format of the set attribute record. `UserID` is the effective user ID of the process setting the attributes.

```
typedef struct {
     RecordHeader   rec;       /* Record info. */
     int       currentHostID; /* Host doing setattr. */
     int       homeHostID;    /* Home host if migrated.*/
     FileID    fid;           /* File being set attr'd.*/
     int       userID;        /* User doing setattr. */
} SospSetAttr;
```

Figure 6: Set attributes record format.

6.5 Get Attributes

Record type: `SOSP_GET_ATTR`

Structure: `SospGetAttr`

This record is created when a process gets the attributes of a file. Figure 7 illustrates the format of the get attribute record.

6.6 Lseek

Record type: `SOSP_LSEEK`

Structure: `SospLseek`

An lseek record is generated when a process changes a stream offset via an `lseek` system call. The format of the lseek record is shown in Figure 8. `Sid` indentifies the

```
typedef struct {
     RecordHeader   rec;      /* Record info. */
     int       currentHostID; /* Host doing setattr. */
     int       homeHostID;    /* Home host if migrated.*/
     FileID    fid;           /* File being get attr'd.*/
     int       userID;        /* User doing getattr. */
} SospSetAttr;
```

Figure 7: Get attributes record format.

stream whose offset was changed. `Start` is the offset of the stream prior to the lseek, and `end` is the offset of the stream after the lseek. `Rwflags` records whether the stream was read or written since the last lseek record (or the stream was opened). If the `FS_READ` bit is set (defined in `fs.h`) then the stream has been read, and if the `FS_WRITE` bit is set the stream has been written.

```
typedef struct {
     RecordHeader   rec; /* Record info. */
     FileID    sid;      /* Stream ID. */
     int       start;    /* Starting offset. */
     int       end;      /* Ending offset. */
     int       rwflags;  /* R/W flags from stream. */
} SospLseek;
```

Figure 8: Lseek record format.

6.7 Close

Record type: `SOSP_CLOSE`

Structure: `SospClose`

A close record is generated when a process closes a stream. The format of the close record is shown in Figure 9. `Sid` indentifies the stream that was closed. `Offset` is the offset of the stream when it was closed, and `size` is the size of the underlying file. `Flags` contains the stream flags, as defined in `kernel/fs.h.Rwflags` has the same meaning as it does in an lseek record (see Section 6.6). `RefCount` indicates the number of references to the stream when it was closed (see next paragraph for details on what this means). `Consist` is set to 1 if the close caused a change in the system's cache consistency state (see Section 6.10).

Due to process migration it is possible for multiple close records for the same stream to appear in the traces. Multiple references to the same stream occur when a process forks. The resulting processes both have a reference to any streams open at the time of the fork. Each of these processes will subsequently close their streams, resulting in multiple closes on the same streams. In the non-migrated case the kernel will only issue a close request to the file server when the last process closes a stream. Process migration, however, makes it possible for two processes on two different clients to share the same stream. In this case the file server keeps track of the stream's reference count, and the clients send all close requests through to the file server. A stream that is shared in this manner is indicated by

```
typedef struct {
     RecordHeader   rec; /* Record info. */
     FileID    sid;      /* Stream ID. */
     int       offset;   /* Offset at close time. */
     int       size;     /* Size at close time. */
     int       flags;    /* Flags for stream. */
     int       rwflags;  /* R/W flags for stream. */
     int       refCount; /* Current refs to file. */
     int       consist;  /* Did consistency on close. */
} SospClose;
```

Figure 9: Close record format.

the presence of the `FS_RMT_SHARED` bit in the `flags` field. If this bit is set then the stream is “really” closed only when the `refCount` field equals zero.

6.8 Migrate

Record type: `SOSP_MIGRATE`

Structure: `SospMigrate`

A migrate record is generated when a process with an open stream migrates. The format of the migrate record is shown in Figure 10. `Id` indentifies the stream that was migrated. `Offset` is the current offset of the stream. `FromHost` is the host on which the process was running prior to the migration, and `toHost` is the host on which the process was running after the migration. Processes always migrate from their home (initial) host to a remote host or vice versa; they never migrate directly between two remote hosts without returning home first.

```
typedef struct {
     RecordHeader   rec; /* Record info. */
     int       fromHost; /* Source host of migration. */
     int       toHost;   /* Dest. host of migration. */
     FileID    id;       /* Stream ID migrated. */
     int       offset;   /* Current offset of stream. */
} SospMigrate;
```

Figure 10: Migrate record format.

6.9 Truncate

Record type: `SOSP_TRUNCATE`

Structure: `SospTrunc`

This record indicates that a stream (file) has been truncated. The format of the truncate record is shown in Figure 11. `Sid` indentifies the stream that was truncated. `OldLength` is the size of the file prior to the truncation, and `newLength` is the size of the

```
typedef struct {
     RecordHeader   rec;      /* Record info. */
     FileID    sid;           /* Stream ID. */
     int       oldLength;     /* Length before trunc. */
     int       newLength;     /* Length after trunc. */
     int       modify;        /* Modify time of file. */
     int       create;        /* Create time of file. */
} SospTrunc;
```

Figure 11: Truncate record format.

file after the truncation. `Modify` and `create` contain the modification and create times of the file, respectively.

6.10 Consistency Change

Record type: `SOSP_CONSIST_CHANGE`

Structure: `SospConsist`

A consistency change record is generated when there is a change in the cache consistency state of the system. The format of the consistency change record is shown in Figure 12. `HostID`, `fid`, and `op` identify the host, file and operation that caused the consistency change. `Op` can be one of `SOSP_OPEN` or `SOSP_CLOSE`, as defined in `sosp.h`. If `op` is `SOSP_OPEN` then `writeOp` indicates whether or not the change involves a file that was open for writing. If `op` is `SOSP_CLOSE` then `writeOp` indicates whether or not the file was cached, either for reading or writing.

```
typedef struct {
     RecordHeader   rec;      /* Record info. */
     int            hostID;   /* Host causing consistency
                               * change. */
     FileID         fid;      /* File ID causing consist.*/
     int            op;       /* Operation causing the
                               * consist change: SOSP_OPEN,
                               * etc. */
     int            writeOp;  /* Whether operation is for
                               * writing (TRUE) */
} SospConsist;
```

Figure 12: Consistency change record format.

6.11 Read/Write

Record type: `SOSP_READ`

Structure: `SospRead`

A read record is generated when a non-cacheable read or write is done to a file. Uncacheable accesses only happen when a file is not cacheable due to concurrent writesharing. Accesses to non-shared files do not create read or write records; reads and writes to non-shared files must be inferred from the lseek and close records.

The format of a read record is shown in Figure 13. `HostID` identifies the host doing the read. `Fid` and `sid` identify the file and stream that is being read. `ReadIt` is set to 1 for read events and 0 for write events. `Offset` and `size` define the range of bytes accessed. `NumNowReading` and `numNowWriting` contain the number of clients with the file open for reading and writing, respectively.

```
typedef struct {
     RecordHeader   rec;        /* Record info. */
     int       hostID;          /* Host doing read/write. */
     FileID    fid;             /* File ID being read. */
     FileID    sid;             /* Stream ID being read. */
     int       readIt;          /* Whether it's a read (TRUE)
                                 * or write (FALSE). */
     int       offset;          /* File offset to start
                                 * operation. */
     int       numBytes;        /* Size in bytes to read or
                                 * write. */
     int       numNowReading;   /* Number of clients with
                                 * file open for reading.*/
     int       numNowWriting;   /* Number of clients with
                                 * file open for writing.*/
} SospRead;
```

Figure 13: Read record format.

6.12 Lookup

Record type: `SOSP_LOOKUP`

Structure: `SospLookup`

A lookup record is generated when a symbolic file name is used to find a file. For example, a lookup is done prior to opening a file to convert the symbolic file name provided by the process into the `FileID` for the file. The format of a lookup record is shown in Figure 14. `CurrentHostId` and `homeHostId` identify the current and home hosts of the process doing the lookup. `Result` is the file that resulted from the lookup. `Status` indicates the status of the lookup operation. 0 indicates that the lookup succeeded (see `status.h` for a complete list of status indications). `Op` indicates the type of lookup operation (see `kernel/fsNameOps.h`). Immediately following the lookup record in the trace file are several file IDs of the directories traversed during the lookup. `Size` indicates the number of file IDs that follow the lookup record.

The use of a lookup record is the only way to determine if a file was created when it was opened. Prior to the open record in the trace will be a lookup record with an `op` of `FS_DOMAIN_OPEN` and a `status` of 0. Compare the file ID in the open record with the last file ID in the list of file IDs following the lookup record. If the IDs do not match then the file was created. If they do match then the file was found during the lookup and hence it already existed.

```
typedef struct {
     RecordHeader   rec;      /* Record info. */
     int       currentHostID; /* Host doing lookup. */
     int       homeHostID;    /* Home host if migrated. */
     FileID    result;        /* FileID result. */
     int       status;        /* Status of lookup:
                               * See FslclLookup. */
     int       size;          /* Number of fileIDs in
                               * buffer space following. */
     int       op;            /* Operation causing lookup:
                               * FS_DOMAIN_OPEN, etc.
                               * See kernel fsNameOps.h. */
     /*
      * Note: the file IDs follow as if in an array
      * immediately after this header. The size field
      * gives the number of the file IDs.
      */
} SospLookup;
```

Figure 14: Lookup record format.

6.13 Consistency Action

Record type: `SOSP_CONSIST_ACTION`

Structure: `SospConsistAction`

A consistency action record is generated when the file server has to invoke a consistency action on a client in order to preserve the cache consistency of the system.The format of the consistency change record is shown in Figure 15. `CausingHostID` identifies the host that did something to cause the consistency action; `affectedHostID` is the host that was affected by the action. `Fid` is the file that is the target of the consistency action. `Action` identifies the type of consistency action, as defined in `kernel/fsconsist.h`.

6.14 Prefix

Record type: `SOSP_PREFIX`

Structure: `SospAddPrefix`

The prefix record is used primarily to synchronize trace files from multiple servers. During the tracing a process was run that tried to access a non-existent prefix (file system) at regular intervals. Doing so generated a prefix record in the trace files of all of the servers, allowing the files to be synchronized at that point. The format of a prefix record is shown in Figure 16, although you'll probably have little use for it.

```
typedef struct {
     RecordHeader   rec; /* Record info. */
     int  causingHostID; /* Host causing consistency
                          * change. */
     int  affectedHostID;/* Host affected by
                          * consistency change. */
     FileID    fid;      /* File ID causing consist. */
     int       action;   /* Whether the action to take
                          * on the client is
                          * FSCONSIST_WRITE_BACK_BLOCKS
                          * and/or
                          * FSCONSIST_INVALIDATE_BLOCKS,
                          * etc. */
} SospConsistAction;
```

Figure 15: Consistency action record format.

```
typedef struct {
     RecordHeader   rec;      /* Record info. */
     int            hostID;   /* ID of host. */
     int            rpcID;    /* RPC sequence number. */
} SospAddPrefix;
```

Figure 16: Prefix record format.

6.15 Delete Descriptor

Record type: `SOSP_DELETE_DESC`

Structure: `SospDeleteDesc`

This record is created when a file descriptor is deleted. Delete descriptor differs from the delete record described in Section 6.2 in that the latter only records the fact that a process unlinked the file. If the file is still open it will not actually be deleted until all open streams to it are closed, at which time a delete descriptor record is generated. The format of a delete descriptor record is shown in Figure 17. `Fid` identifies the file that was deleted. `Modify` and `create` are the file's modification and creation dates. `Size` is the size of the file.

```
typedef struct {
     RecordHeader   rec;      /* Record info. */
     FileID         fid;      /* FileID deleted. */
     int            modify;   /* Modify time for file. */
     int            create;   /* Create time for file. */
     int            size;     /* Size of the file. */
} SospDeleteDesc;
```

Figure 17: Delete descriptor record format.

6.16 Lost Records

Record type: SOSP_LOST_TYPE

Structure: SospLost

Due to buffer size limitations on the servers it is possible for trace records to be lost. If this happens a “lost” record is inserted into the trace that indicates how many trace records were lost. The format of a “lost” record is shown in Figure 18.

```
typedef struct {
     RecordHeader   rec;       /* Record info. */
     int            count;     /* Number of records lost. */
} SospLost;
```

Figure 18: Lost record format.

7 File Identifiers

Files and streams in the traces are identified by the FileID structure. This structure consists of four fields, of which three represent a unique identifier. These three are defined by the structure ID in sosp.h, which has the fields host, domain, and inode. Host identifies the server for the file, domain identifies the file system on which the file resides, and inode is the file number within the domain. Domain is only unique within a particular server -- file systems on different servers may have the same domain number. The domain and inode of a file corresponds to the file major and minor numbers in the UNIX file system. The format of the FileID and ID is shown in Figure 19. When comparing two FileIDs only compare the ID field.

```
typedef struct {
     int  host;     /* Server for file. */
     int  domain;   /* File system (major number). */
     int  inode;    /* File number (minor number). */
} ID;
typedef struct {
     int       notused;  /* Ignore this. */
     ID        id;       /* Unique id. */
} FileID;
```

Figure 19: ID and FileID formats.

8 Filtering Unwanted Activity

The traces capture more system activity than you probably want to know about. This section tells you how to filter out things you probably don’t want.

8.1 Backups

Each morning an incremental backup is done of all of the Sprite file systems. These backups show up in the traces. Their effects can be eliminated by ignoring events generated by the user “dumper”, which is user ID 28.

8.2 Trace Collection

The act of collecting the trace data generated events that show up in the traces. For the first three traces there is a file called `ignore` in the trace directory that lists the IDs of the trace files themselves. You can eliminate the effect of collecting the traces by ignoring events related to these files. During the last two traces we got smart and collected the traces using the "dumper" user ID (28). Ignore events caused by this user to ignore the effect of collecting the traces.

8.3 Kernel File Accesses

Some of the records in the traces were caused by the operating system. The kernel accesses binaries in order to execute them, and it also accesses swap files that act as backing store for running processes. Opens caused by the kernel will not have the `FS_USER` bit set in the open record. Kernel activity can be ignored by ignoring any streams created by an open without this bit set. The kernel does not use the standard read/write/lseek file abstraction to access swap files. Because of this the kernel does not generate lseek records for swap files, so the traces cannot be used to measure paging traffic.

9 Warnings

There are a number of problems with the traces that we never resolved. Fortunately the problems show up for a small number of records. Here's a list of things to watch out for:

1. Time can go backwards. Be prepared. It jumps back for a while, then jumps forward to the correct time.

2. Sometimes a "delete descriptor" record appears before a "close" record.

10 Trace Library

We've provided a library for accessing the trace files. The library will decompress the trace files (if necessary) and return trace records in chronological order[1], even if the trace files are from multiple servers. The following routines are provided by the trace library.

1. The trace library assumes that time of each record in a trace file is greater than the records before it. Records with incorrect time fields (see Section 9) can not be ordered correctly.

10.1 initRead

Syntax:
```
int initRead(myName, argc, argv, hdrPtrPtr)
        char      *myName;
        int       argc;
        char      **argv;
        Sys_TracelogHeader *hdrPtrPtr;
```

Returns: 0 for success, -1 for failure

Use this routine to initialize the trace package. `MyName` is the name of the process, and is used for printing error messages. `Argc` and `argv` contain a list of trace file names. `HdrPtrPtr`, if it is not NULL, returns a pointer to a representative trace file header. If the a trace file name ends in “`.Z`” it is assumed to contain compressed trace data. The `zcat` command is invoked to uncompress the trace data.

10.2 getNextRecordMerge

Syntax:
```
int getNextRecordMerge(buf)

        char      **buf;
```

Returns: Machine ID if successful, -1 otherwise

This routine is used to return the next record in chronological order from the list of files given to `initRead. Buf` is set to point to a static buffer that contains the record. `GetNextRecordMerge` also sets the global variables `startSec` and `startUSec.` The values of these variables must be added to the contents of the `time` field in each record to get the absolute time at which the record was created. The return value of this routine is the Sprite ID of the server on which the trace record was collected.

11 SospExample

We’ve provided a sample program for processing the traces. You should look at it before writing your own to see how to use the trace library, ignore unwanted events, and deal with time jumping backwards. The sample program can be found in `src/cmds/sospexample.`

12 Questions/Comments

If you have any questions or comments about the traces or this document, please send them to jhh@sprite.berkeley.edu.