

USENIX Association

Proceedings of the Winter 1993 USENIX Conference

**January 25 – 29, 1993
San Diego, California, USA**

Amazon Ex. 1004 IPR Petition - USP 7,784,058
--

Oracle Ex. 1004
Oracle v. VirtaMove, IPR2025-00966

For additional copies of these proceedings contact

USENIX Association
2560 Ninth Street, Suite 215
Berkeley, CA 94710 USA

The price is \$33 for members and \$40 for nonmembers.

Outside the U.S.A and Canada, please add
\$25 per copy for postage (via air printed matter).

Past USENIX Technical Conferences

1992 Summer San Antonio	1987 Summer Phoenix
1992 Winter San Francisco	1987 Winter Washington, DC
1991 Summer Nashville	1986 Summer Atlanta
1991 Winter Dallas	1986 Winter Denver
1990 Summer Anaheim	1985 Summer Portland
1990 Winter Washington, DC	1985 Winter Dallas
1989 Summer Baltimore	1984 Summer Salt Lake City
1989 Winter San Diego	1984 Winter Washington, DC
1988 Summer San Francisco	1983 Summer Toronto
1988 Winter Dallas	1983 Winter San Diego

© Copyright 1993 by The USENIX Association
All rights reserved.

ISBN 1-880446-48-0

This volume is published as a collective work.

NIX acknowledges all trademarks appearing herein, including the following registered and unregistered marks:

Holder	Trademark(s)
Adobe Systems, Inc.	PostScript, Display PostScript System
Apple Computer Corp.	Macintosh
Brooktree Corporation	Brooktree, RAMDAC
Chorus Systemes	Chorus
Digital Equipment Corp.	VMS, Ultrix, Decstation, DECsystem, TURBOchannel, DECstation 5000, Alpha
Ingres, Inc.	INGRES
Intel	Paragon
International Business Machines Corp.	AIX, TCF, JFS
Locus Computing Corp.	TNC
MIPS Technologies, Inc.	MIPS, R2000, R3000, R4000
Massachusetts Institute of Technology	X Window System
Microsoft Corp.	MS-DOS
Open Software Foundation	OSF/1
SPARC International	SPARC
SUN Microsystems	SunOS, SparcStation, Solaris, SUN
StorageTek, Inc.	StorageTek
TransArc	AFS
UNIX Systems Laboratories, Inc.	UNIX
Xerox Corp.	Xerox
Xerox Corporation	GlobalView

Printed in the United States of America on 50% recycled paper, 10-15% post consumer waste.

Filesystems, I

Thursday (9:00-10:20)

Chair: Dan Geer

The Design and Implementation of the Inversion File System	205
<i>Michael A. Olson, University of California at Berkeley</i>	
Operating System Support for Portable Filesystem Extensions	219
<i>Neil Webber, Epoch Systems, Inc.</i>	
File Systems in User Space	229
<i>Paul R. Eggert, Twin Sun, Inc.; D. Stott Parker, UCLA Computer Science Dept.</i>	

Overhead

Thursday (10:45-12:05)

Chair: Rob Kolstad

UNIX Kernel Support for OLTP Performance	241
<i>Hyuck Yoo & Tom Rogers, Sun Microsystems, Inc.</i>	
Measurement, Analysis, and Improvement of UDP/IP Throughput for the DECstation 5000	249
<i>Jonathan Kay & Joseph Pasquale, University of California, San Diego</i>	
The BSD Packet Filter: A New Architecture for User-level Packet Capture	259
<i>Steven McCanne & Van Jacobson, Lawrence Berkeley Laboratory</i>	

I/O, I/O

Thursday (10:45-12:05)

Chair: Jeff Schwab

The Organization of Networks in Plan 9	271
<i>Dave Presotto & Phil Winterbottom, AT&T Bell Laboratories</i>	
Removable Media in Solaris	281
<i>Howard Alt, SunSoft, Incorporated</i>	
An Advanced Tape Cataloging System for UNIX Systems	289
<i>Christopher J. Calabrese, AT&T Bell Laboratories</i>	

Kernel Improvements

Thursday (1:30-2:50)

Chair: J. Kent Peacock

Efficient Kernel Memory Allocation on Shared-Memory Multiprocessors	295
<i>Paul E. McKenney & Jack Slingwine, Sequent Computer Systems, Inc.</i>	
An Implementation of a Log-Structured File System for UNIX	307
<i>Margo Seltzer, Harvard University; Keith Bostic, University of California, Berkeley; Marshall Kirk McKusick, University of California, Berkeley; Carl Staelin, Hewlett-Packard Laboratories</i>	
Exploiting In-Kernel Data Paths to Improve I/O Throughput and CPU Availability	327
<i>Kevin Fall & Joseph Pasquale, University of California, San Diego</i>	

File Systems in User Space

Paul R. Eggert – Twin Sun, Inc.
D. Stott Parker – UCLA Computer Science Dept.

ABSTRACT

Current methods for interfacing file systems to user programs suffer two major drawbacks: they require kernel modifications or root privileges, and they are too complicated to be given to ordinary users. In this paper we show alternative methods are possible. The recent rise of dynamic linking provides a new way for users to develop their own file systems: by interposing a layer of user code between a program and the system call interface, a user can alter or extend a file system's behavior.

For greatest power and reliability, such changes to file system behavior must be managed systematically. We present two user-extensible file systems that are loosely modeled on intensional logic. IFS0 is simple, and supports only extended pathname interpretation for files: it permits certain shell-like expressions as pathnames. To this, IFS1 adds directory lookup and an escape mechanism for interpreting pathnames that can be modified by the user at any point. These file systems operate by modifying the semantics of UNIX system calls that take pathname arguments.

With IFS1 a user can develop a wide range of useful file systems without writing a line of C code. We have developed a variety of sample file systems with IFS1, including *tar* image navigation and a software development file system resembling 3DFS. IFS1 can thus be thought of as a simple user-programmable file system toolkit.

Introduction

The past few years have seen the development of hundreds of file systems; everything from AFS (Andrew File System) to ZFS (Zebra File System). The high level of interest warranted a USENIX workshop[1] on the subject. By and large, these file systems have been developed to satisfy user needs not well addressed by traditional UNIX file systems.

Unfortunately, existing file system interfaces discourage user experimentation. Changing file systems requires kernel modifications or root privileges; the task of writing a new file system is far too complicated to be given to most users. Current file system interfaces discourage adding even simple naming services. For example, users often complain that applications do not understand pathnames like “~” or “\$HOME” that use shell-interpretable expressions. Many applications have independently added support for these expressions, but this is a wasteful duplication of effort. It's hard to imagine solving this problem with current systems, since these abbreviations depend on information not readily available to either the kernel or the file system.

We propose intensional file systems as a simple, easy-to-explain abstraction for implementing one file system on top of another, an abstraction that provides motivating design principles for file system

designers, and that makes it easy for users to define new naming rules. In an intensional file system, a file system is stored as a description of how to name or produce files, instead of as the filenames or contents themselves. The term “intensional” comes from intensional logic, which concentrates on the problem of ascribing meaning to a logical expression or name. The fundamental design problem of an intensional file system is to specify the rules and mechanism for “extensionalizing” the files of an intensional file system, i.e., for computing the actual file that an IFS file denotes.

Our main motivation for intensional file systems is to provide a simple, systematic method for users to augment their file systems with naming services. Our design principles are heavily influenced by those of Satyanarayanan[12] for modern distributed file systems. We assume that clients have CPU cycles to burn. Our goals are to cache on clients whenever possible, to let users exploit usage properties, to minimize system-wide knowledge and change for easy administration, to minimize the security risk, and to have adequate performance.

Naming and Intensionality

Many file systems amount to either an *extended name service* or an *extended system call service*. The former provide a richer view of the underlying files, but little more than this; examples include the Semantic File System[5], 3DFS[8], and TFS[7]. The latter extend many system calls beyond their normal UNIX semantics.

This work was supported by a University of California/Twin Sun MICRO grant, and NSF grant IRI-8917907.

The central issue in the first kind of system is what a “name” is, and how it is to be interpreted. The issue in the second kind of system is the mapping between a particular system call and its semantics. Philosophers have tried to clarify issues in mapping between expressions and their meaning for several hundred years. This work is usually lumped under the subject “intensionality” or “intensional logic”.

Intensionality

Intensionality ascribes meaning to a sentence or an expression (such as a name). The *intension* of an expression is the procedure that finds the value (concrete meaning) of the expression in any given context. This value is also called an *extension* of the expression. Briefly, intensionality makes meaning be relative to context.

The meaning of expressions are often context-dependent: the meaning of the phrase *the current time* depends on when and where it is evaluated, and the sentence *The expression $\$(E)$ has the same meaning as the expression ‘E’.* has the meaning *true* for the Korn shell, but *false* for the traditional Bourne shell.

Intensional logics formalize these distinctions, and distinguish between expression syntax, intension, and extension. Syntax defines the expressions of a language; the *extension* of an expression is its value or what it denotes; and the *intension* of an expression is a function on contexts that yields the extension in the given context. The intension of an expression is a function from a context to a value. Figure 1 shows that an intension is the connection between a language expression and its extension.

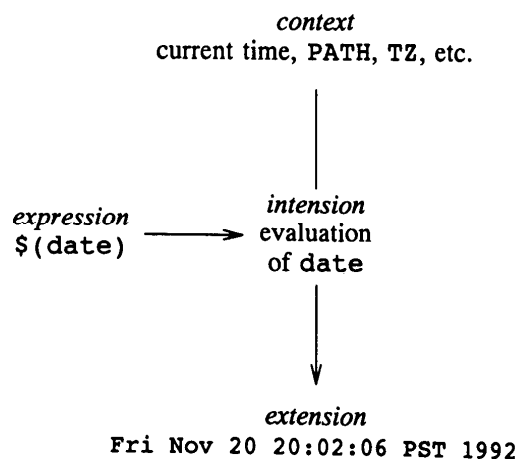


Figure 1: Intentions map contexts to extensions

For a clear, up-to-date discussion of the philosophical perspective on issues in intensionality, see Zalta [15].

Intensionality and File Systems

Naming mechanisms in file systems have been ad hoc, and could benefit from a more general foundation. A naming mechanism must somehow cover the distinctions just raised if it is to be general.

Intensional logic suggests a foundation. However it must be adapted for file system applications for a variety of reasons:

- Names in English (or any other natural language) are usually simple proper names, while names in file systems are complex paths with multiple atomic components, and can be highly time- and context-dependent.
- The concept of “evaluation” in English is limited, but is sophisticated and heavily used in an operating systems context.
- File systems raise many issues that are not of concern in processing English, such as performance, security, and conceptual simplicity.

Intensionality mechanisms used in file systems should reflect concerns like these. In IFS, we adapt the idea of intensionality in two ways:

1. The space of pathnames can be extended to include not just ordinary filenames, but also expressions. Any general name service can then be built by providing an interpreter that evaluates name expressions.
2. The same idea can be applied to system calls. Any system call (like `chmod("-", 0755)`) is an expression, whose interpretation can depend on context. We provide an interpreter that defines the context, i.e., defines what the system call does. This gives a general system call interface.

The second treatment is more general than the first, since name service is essentially determined by the open system call.

We have built two file systems that represent these two adaptations of intensionality in a user-programmable way. IFS0 is a name service that permits shell-like expressions as filenames, while IFS1 adds directory lookup and user-defined naming escapes.

IFS0

We initially conceived IFS0 [4] as a response to the evolution of proliferation of “buildable files”, taking advantage of the fact that today CPUs often have cycles to burn.

Today many files (formatted manual pages, diagnostics, binary executables, tar files, files copied from across the net, etc.) can be easily produced on demand whenever needed, rather than being stored on disk. Our idea was to somehow include a make-like capability into the file system, allowing it to rematerialize a buildable file whenever the file is needed. This idea grew into IFS0.

Overview of IFS0 Use

IFS0 extends the UNIX tree-structured pathnames by adding intensional pathnames, which are expressions evaluable with the shell. Intensional pathnames can evaluate to either file contents, or to names of other files. Pathname expressions evaluating to file contents are written with enclosing parentheses (...), while expressions evaluating to pathnames are written using the Korn shell notation \$(...). For example, the intensional pathname (date) evaluates to the current date, while the intensional pathname \$(whoami) evaluates to the file named by the output of the command whoami.

IFS0 is quite simple from the user's viewpoint. With IFS0 the user creates an intensional file using the ordinary UNIX ln command, e.g.,

```
$ ln -s '(date)' now.txt
$ cat now.txt
Sun Mar  8 20:06:31 PST 1992
$ awk '{print $4}' now.txt
20:07:01
```

To see whether a file is intensional, you invoke ls -l in the usual way. The ownership, length, and date is that of the intension, not the extension. To see the extension, append the -L option; just as with symbolic links, it causes IFS0 to refer to the extension instead of the intension.

```
$ ls -l now.txt
lrwxrwxrwx 1 pre  6 Mar 8 20:06 now.txt -> (date)
$ ls -lL now.txt
-r----- 0 pre  29 Mar 8 20:14 now.txt
```

For a more practical example, consider the following make rules:

```
dist.tar.Z: dist.tar
    compress dist.tar >$@
dist.tar:
    tar cf - *.h *.c >$@
clean:
    rm -f dist.tar*
```

These can easily be replaced by intensional files that are always up-to-date, takes far less space than their extensional counterparts, and never need cleaning, as follows:

```
ln -s '(compress <dist.tar)' dist.tar.Z
ln -s '(tar cf - *.h *.c)' dist.tar
```

There is no problem with defining intensional files in terms of other intensional files this way. For example, to make a compressed tape, one can execute the command

```
dd <dist.tar.Z >/dev/rst1
```

- in the process of extensionalizing dist.tar.Z, IFS0 recursively extensionalizes dist.tar first.

Intensional Pathnames in IFS0

Any tree-oriented file system provides *extensional pathnames*, sequences of existing filenames specifying a path to a desired file from some initial directory. IFS0 extends this widely-used model by permitting *intensional pathnames*, which include shell-evaluable expressions.

Intensional pathnames are essentially shell programs that yield a pathname when evaluated. For example, the intensional pathname \$(sunp Xsun X) yields the name that is the output of the shell command sunp Xsun X. Suppose that the sunp command is defined as follows:

```
#!/bin/sh
if (sun) 2>/dev/null
then echo $1
else echo $2
fi
```

Then the pathname \$(sunp Xsun X) is equivalent to the pathname Xsun if interpreted on a Sun workstation, and otherwise is equivalent to the pathname X.

Extensional pathnames are annoyingly restrictive, and this has led to the creation of symbolic links, conditional symbolic links, and other similar extensions to UNIX. IFS0 lifts the restriction in a general way, letting pathnames include expressions. Although symbolic links were regarded with caution when they were introduced, they are now a popular mechanism in UNIX. IFS0 can be used as a natural extension of symbolic links.

Specifics of IFS0 Pathname Interpretation

IFS0 overloads the conventional notion of *pathname* with two specializations: *extensional pathname* and *intensional pathname*. An extensional pathname is a standard UNIX pathname.

In IFS0, an *intensional pathname* is of the form

D/\$ (command)

where *D* is an extensional pathname. When this pathname is interpreted, *command* is executed in the *D* directory, and the output is interpreted as a pathname. The syntax is borrowed from the Korn shell.

Mirroring these pathname types are file types. An *extensional file* corresponds to a conventional UNIX file. An *intensional file* is a symbolic link containing an *extensional pathname* or *intensional pathname*. When opened, intensional files cause evaluations that ultimately yield an extensional file.

IFS0 redefines *pathname system calls*, i.e., system calls like open and stat that have pathname arguments and require extensional files. IFS0's interpretation of these calls differs from that of the underlying file system in the following ways:

1. In IFS0, the pathname can be intensional. If so, it is evaluated before further processing.

2. Symbolic links can contain intensional pathnames. Therefore, pathname system calls first invoke `readlink` to see whether the file is intensional.
3. The context for an open includes not only the working directory, the user id, and so forth, but also includes anything accessible to the process that interprets the intensional pathname.
4. Accessing a file repeatedly uses the IFS0 evaluation mechanism until it yields an extensional file, or an error.
5. Extensionalization aborts if the total number of dereferences required exceeds the declared limit (currently 20). The limit of 20 prevents loops, and was inspired by the traditional limit of 20 symbolic link expansions in Berkeley UNIX.

IFS0 also supports one abbreviation. The pathname

D/(command)

is equivalent to the name of a temporary file containing the output of *command*.

IFS1

IFS0 showed us the potential for programmable file systems interfaces. Although IFS0 is satisfactory as a file system by itself, it has two major weaknesses.

First, IFS0 insists on having an intensional file (symbolic link) for any object file that one wants extensionalized. For example, with IFS0 we can store a large file *F* in compressed form *F.Z*, and use its extensional form:

```
$ ln -s '(zcat F)' F
$ ls -l F*
lrwxrwxrwx 1 pre      17 Nov 10 22:00 F -> (zcat F)
-rw-r--r-- 1 pre 788213 Nov 10 22:00 F.Z
$ ls -lL F
-r----- 0 pre 1297200 Nov 10 22:01 F
```

Unfortunately, with IFS0 we must manually create the symbolic link *F* in order to get access to the uncompressed file.

Second, IFS0 extensionalizes only the last filename component of a pathname. This simplifies the implementation, but it prevents intensional directories, which are highly desirable. We were reluctant to add intensional directories to IFS0, since it would require one extra `readlink` system call per pathname component, even for ordinary extensional paths.

When we considered fixes for these shortcomings, we were struck by the wealth of options available. When extensionalizing a directory, should the resulting directory temporary be placed in */tmp* or near the intensional directory? Should extensionalized directories be shared among several process invocations, or private to each process? Should the extensionalization rules depend only on intensional

files' names, or also on their contents? These and other questions quickly convinced us that we could not hope to simply add code and options to IFS0 to support directories and writable files – the resulting system would have been too inflexible and arbitrary.

Instead, we arrived at a design that uses a user space kernel together with an external extensionalizer specified by the user at run time via an environment variable. IFS1 improves on IFS0 by adding user-defined escapes, and by invoking the extensionalizer only for pathnames that do not exist extensionally, and which either are unusual (i.e., end in “*!*” or contain parentheses) or contain a symbolic link component that expands to an unusual pathname. The “*!*” pathnames invoke a user-defined escape mechanism; pathnames containing parentheses are interpreted much as in IFS0.

Users define a pathname escape mechanism with the environment variable `IFS_COMMAND`. For example, we can define `IFS_COMMAND` to interpret pathnames like *F!*, where *F* does not exist but *F.Z* does. The following script copies *F* to the standard output, and caches the extensional uncompressed file in */tmp*.

```
#!/bin/sh
- zcat <$1.Z || exit 2 # ENOENT
```

The program “*-*” runs its arguments as a command, directs standard output to a temporary file in an IFS-managed directory in */tmp*, and yields the name of the temporary file. Suppose the above script is named */bin/zcat.ext*, and we set `IFS_COMMAND` to */bin/zcat.ext*. Afterwards, references to *F!* force *F* to be extensionalized:

```
$ ls -l F*
-rw-r--r-- 1 pre      788213 Nov 10 22:00 F.Z
$ ls -lL F
F not found
$ ls -lL F!
-r----- 0 pre 1297200 Nov 10 22:01 F!
```

The `IFS_COMMAND` program can specify further actions for IFS1 to take by outputting other information. It tells IFS1 that the new file is a temporary by prefixing its pathname with `(unlink)`; IFS1 then unlinks the file after doing the system call on it, and before returning to the user program. For example, the IFS0 abbreviation

D/(command)

is implemented in IFS1 by defining it to be equivalent to the pathname

D/\$(- command)

where “*-*” is the temporary file command described above. Here is a simple implementation of the “*-*” command:

```
#!/bin/sh
echo -n "(unlink)/tmp/ifs$$"
exec "$@" >/tmp/ifs$$
```

The user-defined escape feature permits something like make processing to be performed directly by IFS1. The `zcat.ext` script implements the makefile rule

```
.SUFFIXES: .Z
.Z:
    uncompress $*
```

but works anywhere in the file system, handles compressed files in read-only directories, does not require explicit invocation of a special program like `make`, and automatically cleans up the results.

The “`!`” escape character is required to force IFS1 extensionalization. This convention avoids surprises; extensionalization happens only for pathnames containing explicit IFS0 intensional expressions, pathnames containing symbolic links that expand to such expressions, and pathnames ending in “`!`”. This makes it possible for the user (and IFS1) to determine easily whether a pathname will involve extensionalization. Also, with `cs`h this convention works well, since the command line abbreviation “`!!!`” means “do the previous command, but this time extensionalize its last argument”.

Specifics of IFS1 Pathname Interpretation

The wrapper that IFS1 interposes around a pathname system call has no extra effect if the system call at first succeeds, or if it fails for reasons other than the file's lack of existence. However, if the call fails because of a missing file (i.e., `errno=ENOENT`), and if the environment variable `IFS_COMMAND` is set, IFS1 examines the pathname looking for parenthesized commands, or for symbolic links that expand to parenthesized commands. If it finds any, it extensionalizes them using rules similar to IFS0; otherwise, if `IFS_COMMAND` is nonempty and the pathname ended with “`!`”, IFS1 invokes `IFS_COMMAND`, passing it the pathname (minus “`!`”), the name of the system call, and some extra information to tell the program whether the file is being accessed for reading or writing. If the program fails, IFS1 returns `-1` and sets `errno` to the exit status of the program; otherwise, IFS1 retries the system call with the new pathname output by the program, and immediately unlinks the new pathname if the program preceded it with the string `(unlink)`.

Sample IFS Applications

Previous sections have shown small examples like conditional symbolic links and date time stamps. This section shows some more elaborate examples of the IFS mechanisms in use.

FTP Storage

With IFS we can store a large file `F` on another site, and use FTP to retrieve the file whenever needed. For example:

```
$ ln -s '(catftp alex.sp.cs.cmu.edu\'
> 'cs-techreports/list.of.papers)' \
> internetTRs
$ ls -l internetTRs | fold -45
lrwxrwxrwx 1 stott      57 Nov 14 15:40
internetTRs -> (catftp alex.sp.cs.cmu.edu cs-
techreports/list.of.papers)
$ wc internetTRs
4638      4638 260659 internetTRs
```

where `catftp` is a shell script that invokes the `ftp` command with anonymous login.

File Fingerprinting and Encryption

Suppose we trust our local workstation, but are using networks or file servers that are less reliable, and we want to check against inadvertent or malicious modification of a file whenever we access it. We have written a program called `md4cp` that is like `cp`, but the copy that results is an intensional file that verifies the original against a fingerprint generated by the MD4 message digest algorithm [11]. Here is the source code for `md4cp`: it uses `md4` to generate the fingerprint, and creates an intensional file that will use `md4verify` to check the fingerprint.

```
#!/bin/sh
if [ -d $2 ]
then F=$2/'basename $1'
else F=$2
fi
ln -s "\$(md4verify $1 'md4 <$1')\" $F
```

With this we can create fingerprinted files. For example, suppose `.` is reliable, but `/usr/local` is not. We make a new self-checking X server as follows:

```
$ md4cp /usr/local/bin/X11/X .
$ ls -l X | fold -45
lrwxrwxrwx 1 stott      66 Nov 20 17:18
X -> $(md4verify /usr/local/bin/X11/X 54fbda3
4ee46771858401f2494f72b1d)
```

Before invocation, `X` first verifies that `/usr/local/bin/X11/X` still has the indicated checksum. Of course, IFS isn't necessary for this particular example: since the X server is a program, we could have just written a small shell script that verified and invoked the actual X server. But the IFS approach is more general than the shell script approach, because it works for non-executable files as well.

The source code for `md4verify` is as follows:

```
#!/bin/sh
if checksum='md4 <$1'
then
    case $checksum in
        $2) echo -n $1 ;;
        *) exit 22 # EINVAL
    esac
else exit 2 # ENOENT
fi
```

It should also be possible to store a file in encrypted form using IFS; reading the file would invoke a window-based password requester that demands the password directly from the user. (The user interface of the traditional UNIX `crypt` command is inappropriate for IFS, because it disturbs the application screen layout.) This could be done with a command like `ln -s '(xcrypt F.x)' F`.

A Versioning File System

We have implemented a simple versioning file system with function reminiscent of 3DFS[8]. Our file system is based on RCS[13]. It is implemented with an extensionalizer called `rscs.ext` that has two major features: access to old versions, and automatic checkout of the latest version.

Access to Old Versions

With `rscs.ext`, the name `p,rn!` evaluates to a temporary file containing a copy of pathname `p`'s version `n`. For example, the shell command

```
diff ifs1.h,r1.3! ifs1.h,r1.4!
```

compares version 1.3 to version 1.4 of the file `ifs1.h`. This behaves much like the traditional RCS command

```
rscsdiff -r1.3 -r1.4 ifs1.h
```

but IFS has the advantage that old versions are available to all commands, not just those which have a special RCS wrapper program. For example,

```
grep extension ifs1.h,r1.3!
```

searches for the string `extension` in version 1.3 of `ifs1.h`. With the traditional approach, a new `rscsgrep` wrapper program would have to be written, but with IFS we can use ordinary `grep`.

Automatic Checkout

Developers often test small changes to a set of source files in a private area that contains copies of the files. In such an environment, it is convenient to have unchanged files checked out automatically when they are needed. Modern versions of `make` have automatic checkout built in; `rscs.ext` implements automatic checkout for all programs, not just `make`. If `rscs.ext` is invoked on a file `F` corresponding to an ordinary RCS file, it extensionalizes the file by invoking the RCS checkout command `co -q F`. The following example shows how `rscs.ext` extensionalizes the file `OK` the first time that it is referenced.

```
$ ls -l
total 2
drwxrwsr-x 3 p 512 Nov 20 14:11 RCS
lrwxrwxrwx 1 p 17 Nov 20 14:10 OK -> $(rscs.ext OK)
$ head -1 OK
Here are the detailed results of the first round:
$ ls -l
total 3
drwxrwsr-x 3 p 512 Nov 20 14:11 RCS
-r--r--r-- 1 p 1170 Nov 20 14:12 OK
```

Lazy Recursive Checkout of Directories

If `rscs.ext` is invoked on a directory `D` corresponding to a subdirectory `RCS/D` of the RCS hierarchy, it lazily extensionalizes `D` by `mkdir`ing it and populating it with intensional files, one for each entry in `RCS/D`. The following scenario shows one step of `rscs.ext`'s lazy recursive directory extensionalization prompted by a user's `cd` command.

```
$ ls -l RCS/ifs
total 21
drwxrwsr-x 2 p 512 Nov 20 14:17 doc
-r--r--r-- 2 p 18418 Oct 31 15:55 i.c,v
-r--r--r-- 2 p 1758 Oct 31 15:55 i.h,v
$ ls -l
total 2
drwxrwsr-x 4 p 512 Nov 20 14:17 RCS
lrwxrwxrwx 1 p 14 Nov 20 14:19 ifs -> $(rscs.ext ifs)
$ cd ifs
$ ls -l
total 4
lrwxrwxrwx 1 p 10 Nov 20 14:19 RCS -> ../RCS/ifs
lrwxrwxrwx 1 p 14 Nov 20 14:19 doc -> $(rscs.ext doc)
lrwxrwxrwx 1 p 16 Nov 20 14:19 i.c -> $(rscs.ext i.c)
lrwxrwxrwx 1 p 16 Nov 20 14:19 i.h -> $(rscs.ext i.h)
```

Implementation

The RCS file system extensionalizer `rscs.ext` is a perl script containing about fifty lines, including commentary. A listing appears in Appendix 1.

A tar File System

Although `tar` archive files have become a standard way to store collections of files, particularly compressed archives, they are hard to inspect. De-archiving them takes time, uses disk space, and must be cleaned up.

We can avoid archive file problems using IFS1. As an example, here is a simple `IFS1_COMMAND` shell script that automatically extensionalizes `tar` files into their corresponding directories.

```
#!/bin/sh
if [ -f $1.tar ]
then = tar xf - <$1.tar
elif [ -f $1.tar.Z ]
then zcat $1.tar.Z | = tar xf -
else exit 2 # ENOENT
fi
```

With this, we can `cd` directly into `.tar` files or `.tar.Z` files. The program `"="` is an IFS1 utility that extensionalizes a command in a temporary directory in `/tmp`, which is eventually removed by a cleanup daemon.

This approach works well for small archives. Large archives can cause problems, however, because the de-archival will store megabytes of stuff that we mostly don't care about in `/tmp`. Often there is not enough space left to do this safely.

Using the IFS1 approach, we implemented a lazy de-archiver that produces intensional files that refer back to the archive for more de-archival, if this turns out to be wanted. We did this by extending `tar` itself to support intensionality. In about 200

lines of C we added two options to Gnu tar 1.11.1:

- The H option indicates that the pathname arguments are actually numbers of specific records in the archive, each of which is the archive header record for a file to be extracted. For example, the command `tar xfh foo.tar 35 43` extracts the two files whose headers are at record numbers 35 and 43 (after verifying these headers correct). Access is fast for archives that are ordinary files (not fifos). Compressed archive access is much slower, since the entire archive must be sequentially uncompressed.
- The I option causes tar to extract files as intensional files that invoke tar with the H option, selecting the appropriate header record in the archive for extraction.

The Gnu tar program also provides three other useful options: Z announces that the input file should be uncompressed first, and O announces that extracted files should be routed directly to stdout. These extra options make it possible to satisfy all our de-archival needs with tar itself.

The scenario in Appendix 2 was obtained with the Gnu Chess tar file (which was compressed). After `cd'ing` into the compressed tar file, we poked around. The `chdir` system call handles the escaped pathname `gnuchess-4.0.pl60!`, causing a lazy intensional version of the tar file directory structure to be created in a subdirectory of `/tmp`. Although the original tar file is 400 Kbytes, the image in `/tmp` is only a few thousand bytes. It consists of the directories in the original and intensional files pointing at their header records into the tar file. E.g., `gnuchess.book` is an intensional file equivalent to

```
(tar xfhOZ /tmp/gnuchess-4.0.pl60.tar.Z 334)
```

which announces: to get this file, extract from the original compressed archive, putting the file with header record 334 to stdout. Since any dynamically linked program automatically incorporates `IFS`, `ls`, `head`, and `diff` all get the files you would expect.

The lazy tar extensionalizer is as follows:

```
#!/bin/sh
if [ -f $1.tar ]
then = tar xI 'realpath $1.tar'
elif [ -f $1.tar.Z ]
then = tar xIZ 'realpath $1.tar.Z'
else exit 2 # ENOENT
fi
```

This is much like the eager script shown earlier, except that provision is made to save the full pathname of the original tar file, since it may need to be referred to later. The `realpath` utility produces an absolute pathname from a given

pathname. The key feature is the new I option of tar: when run in the extensional directory built by "=", it produces the intensional files seen in Appendix 2.

The lazy approach taken to tar navigation taken here is space efficient, but of course charges a high price in performance. The response time for compressed archives can be irritatingly slow, since the archive is uncompressed and scanned sequentially on each access. However, it would not be difficult to change the extensionalizer here to check how much space is left in `/tmp`, and use lazy extensionalization only if this is necessary.

Design Issues for Intensional File Systems

Basic Design Principles

Our design principles for IFS were influenced by those of Satyanarayanan [12], and included the following:

- *Clients have cycles to burn.*
More and more, file client processors are nearly as fast as (and a much less scarce resource than) file server processors.
- *Cache whenever possible.*
Like CPU cycles, client caches are becoming relatively abundant and parallelism is maximized by moving as much work as possible onto the clients.
- *Let users exploit usage properties.*
The people who actually use intensional files are more likely to know their usage properties than even the best file system designer. Keeping the IFS system design simple makes it easier for users to gain intuition about how to set up their intensional files efficiently.
- *Minimize system-wide knowledge and change.*
The more local an intensional file's specification is, the easier it is to build and scale the file system containing that file, particularly to large distributed systems.
- *Minimize security risk.*
A primary danger of intensional files is that they invite abuse. Malicious users could replace extensional user files with intensional files that log keystrokes or perform other unspeakable acts, for example. This danger cannot entirely be eliminated; it is analogous to the problem of letting users set their PATH environment variable to malicious users' directories. However, the danger should not be unnecessarily enhanced, or made so subtle that users will not be able to avoid it easily.
- *Have adequate performance.*
File systems must provide adequate performance, or they will not be used. Of particular importance is *interactive* performance.

Since IFS0 and IFS1 are just first prototypes, high performance is not essential; but enough is needed to get adequate hands-on experience.

Introduction of Intensional Files

The first design issue in implementing intensional files is how the user is supposed to conceptualize them. Intensional files must be mapped into the existing file system model in a coherent way.

Even IFS's modest implementation exposes several fundamental design considerations for intensional file systems, including the following:

- *How can we add intensional files gracefully?* It should be easy for users to see exactly which files are intensional, and what rules are being used to extensionalize them. A principal goal of our work was to come up with simple, intuitive rules governing intensional files. This is in addition to the obvious goal that unmodified traditional programs should transparently access the extensional counterparts of intensional files.

- *How heavyweight should intensional file systems be?* Traditional UNIX file systems contain a large number of files and directories, and (for security reasons) can be manipulated only by the superuser. At first, we followed this model, but eventually we rejected it because it is inappropriate for a design meant for personal use and tailoring.

- *Should extensionalization be done by the file system client or by the server?* Doing the work on the server means that we don't have to touch the clients; but doing it on the clients increases parallelism and decreases network overhead.

These issues are not new to intensional files. The introduction of symbolic links in Berkeley UNIX raised most of these issues, albeit in somewhat simpler ways. Symbolic links are an important special case of intensional files: the intensional counterpart of a symbolic link is the name of the linked-to file, and the extensional counterpart is the contents of the linked-to file. Once we saw that symbolic links were special cases of intensional files, it became obvious that one way to implement intensional files is to generalize symbolic links.

A more radical analogy between intensional and traditional file systems is that a traditional file system can be modeled as an IFS backed by a traditional UNIX special device. Thus, modern microkernels with file systems migrated out of the kernel have much in common with intensional file systems.

Related Work in File Systems

There has been an enormous amount of interest recently in extending file systems to support new functionality, as indicated by the recent USENIX workshop on file systems [1]. New approaches strike a compromise between flexibility, performance, and completeness. For example, operating systems like

Mach permit the entire file system to reside outside the kernel, yet few users would be capable of writing a new Mach file system or even making ad hoc file system modifications. UNIX System V Streams [2] are much less flexible, providing a service that is user-programmable but not really user-extensible. (An extensible variant [10] has been proposed, however.)

Stackable file systems like Ficus [6, 9] are a promising way to implement new file system concepts atop traditional systems. They provide a disciplined scheme for incorporating features into a file system by modeling the file system design as a protocol stack, and letting one insert new protocols onto the stack. This can greatly ease implementation by file system experts, but it's not a task for novices, and it requires root permissions.

One general approach that has been proposed for extensible file systems is *file system interface mapping*, in which basic system calls like `open` and `read` can be replaced by user code. This approach can encourage exploration, but a key issue is how the mapped interface is specified by the user.

One approach for file system interface mapping allows users to associate a process with a given file that implements the appropriate mapping. The kernel is modified to route system calls for the file to this process. Bershad and Pinkerton [3] describe *watchdogs*, which are user-defined processes that the user can attach to a file. Watchdogs are notified about each system call affecting the file. They have been used to implement access control, file compaction, mail biffing, directory views, transparent remote file access, etc. Welch and Ousterhout [14] propose *pseudo-devices*, basically processes that can be treated like files, again by interface mapping. Various sticky programming jobs such as interacting with device drivers, X servers, and TCP/IP services can be simplified by treating them as pseudo-devices.

Another general approach is to encourage user redefinition of the basic system call interface in `libc`. Any such redefinitions are automatically adopted by all dynamically linked programs. This approach requires no kernel modifications, and works entirely in user space. COLA [8] is a system call interception scheme developed at Bell Labs. Basically, COLA is to system calls at the user level as Ficus is to file system protocols at the kernel level. With COLA, a programmer can insert a new definition of, say, the `read` system call; the programmer-inserted code operates as a layer between the actual system calls and the virtual `read` call that is visible to the rest of the program. We had independently developed techniques similar to COLA in a memory management profiler, and were thus familiar with its implementation strategy. It is the approach used in IFS.

IFS1 Implementation

IFS1 is currently implemented atop SunOS 4.1.x, and operates by intercepting the few system calls that refer to pathnames, notably `open` and `stat`. By manipulating an environment variable, a user can tell the dynamic linker to preload IFS1 before linking the standard C library; the `IFS_COMMAND` environment variable enables and disables IFS1 itself.

Aside from a few subtle issues about which system calls to intercept, and how to prevent interception self-loops, IFS1 itself is surprisingly simple: its kernel is only 700 lines of C code. Although there are some obvious ways to improve performance, little attempt has been made to tune the code.

IFS1 Performance

Despite this rather academic attitude towards performance, IFS1 is useful in practice. In ordinary use it's not even noticed. For example, starting up a new `cs`h ordinarily takes about 500 system calls in our environment; IFS1 adds just seven system calls to this total, five to dynamically prelink the IFS1 library, and two to check whether `~/.history` (a nonexistent file) is intensional. On a Sparcstation ELC running SunOS 4.1.2, IFS1 adds only 32 μ s overhead to each pathname system call in the usual case of an existing extensional file. There is no overhead for system calls like `read` and `write` that do not interpret pathnames. Since intensional files are cached in `/tmp`, the normal SunOS `tmpfs` cacheing means that one can even construct examples where IFS1 is faster than ordinary file system access. Another situation where IFS1 compares favorably with conventional file systems is the not unusual combination of a slow remote file server, a fast client, and large local extensionalized files.

The key reason for IFS1's speed is that it acts only in unusual circumstances, when a pathname does not exist, and has an unusual form or a symbolic link. When IFS1 does not act, the overhead is low – generates no extra system calls in the usual case. When IFS1 does act, the overhead of the IFS1 mechanism itself is low even with the current inefficient prototype; under a tenth of a second on a Sparcstation ELC as measured by `cs`h time. This is perfectly adequate for interactive applications.

Security

IFS1 takes a naive approach to security. The process that extensionalizes a pathname has almost all the privileges of the process that is accessing the pathname. However, all the extensionalizer's file descriptors are closed except for standard output; this exception is more to avoid inadvertent side effects than to close security holes. Also, for safety's sake, the dynamic linker does not prelink IFS1 in setuid programs.

IFS uses only facilities already available to ordinary users, and does not require superuser access either for installation or use, so in some sense IFS1 does not introduce new security issues. In ordinary use, we expect that avoiding Trojan horse extensionalizers will be about as difficult as avoiding ordinary Trojan horse programs in one's `PATH`. Also, as the MD4 example showed, IFS can be used to improve security in an insecure environment.

Future Work

We consider IFS0 and IFS1 as successful experiments. Their success suggests some obvious extensions for the next version.

- *Writeable intensional files*

One would like the ability to not only read intensional files, but also write them. For example, writing to the extensional counterpart of a compressed file should cause the `compress` program to be invoked to store the actual file on disk. Even more than intensional directories, this requires careful resource allocation; for example, we must be careful that inadvertent process death does not undo the effects of `write`.

- *Security*

IFS1's naive approach to security is simple and easy to explain, which is an important virtue in security matters, but some potential users have expressed fears about bugaboos like Trojan horse extensionalizers. IFS1 could be extended to run extensionalizers in a more restricted environment.

- *Cacheing*

IFS does little cacheing of extensionalizations of files: even if the same process opens a file twice, IFS dutifully does the full work of extensionalizing the file twice. The “-” and “=” utilities could be made much more intelligent. Although extensional caches would improve performance, they would come at the cost of complicating the implementation with the usual questions of cache management.

- *Managing imaginary file systems*

Some tools need directory information that may not exist in an intensional system. For example, consider the use of `du` on an intensional directory, which would ordinarily discover how much storage the (extensional version of the) directory occupies on disk. Should IFS materialize the entire directory just so that `du` can see how big the result is? Such questions show there is lots of work to do here.

- *When is it worthwhile to make files intensional?*

It is not normally worth making the executable version of a large software system an intensional file, since its materialization can

be costly. Currently it seems that the main payoff of intensionality comes with files that are relatively easy to materialize, or that are relatively large, or both.

Conclusion

It is high time to open file systems up to direct user manipulation, rather than leaving them closed systems that only qualified wizards are permitted to design. Mundane tasks like versioning, encryption, and compression should be handled by a file system directly, instead of by the current multitude of ad hoc approaches, one per application. Current dynamic linking technology (and the object-oriented technology of the not-too-distant future) should make it easy for users to add new file naming conventions in a systematic way.

IFS builds on the idea of intensionality to give users a flexible way to extend their file systems' naming conventions. The examples in this paper show that intensionality can replace and extend many ad hoc file system techniques.

Over the past year we have tried to isolate a part of intensionality that can form a practical basis for file systems development. The current design of IFS1 reflects both a basic "expressions as names" mechanism and a "name escape" mechanism. We hope further experience with IFS will produce a important mechanism for day-to-day use. If nothing more, however, IFS represents the kind of file system interface that lets a hundred flowers blossom and a hundred schools of thought contend.

References

1. *Proceedings of the First USENIX File Systems Workshop*, Ann Arbor, MI, May 1992. P. Honeyman (program chair).
2. AT&T, *UNIX System V Release 3.2 Streams Programmers Guide*, Prentice-Hall, 1989.
3. Bershad, Brian N. and C. Brian Pinkerton, "Watchdogs - Extending the UNIX File System," *Computing Systems*, vol. 1, no. 2, pp. 169-188, 1988.
4. Eggert, Paul R. and D. Stott Parker, "An Intensional File System," *Proc. First USENIX File Systems Workshop*, pp. 145-146, Ann Arbor, May 1992.
5. Gifford, D.K., P. Jouvelot, M.A. Sheldon, and J.W. O'Toole Jr, "Semantic File Systems," *Proc. of the 13th ACM Symposium on Operating Systems Principles*, pp. 16-25, October 1991.
6. Heidemann, J.S. and G.J. Popek, "An Extensible, Stackable Interface for File System Development," Technical Report CSD-900044, UCLA Computer Science Department, Los Angeles CA 90024-1596, December 1990.

7. Hendricks, D., "A Filesystem for Software Development," *Proc. USENIX Summer Conference*, pp. 333-340, Anaheim, June 1990.
8. Korn, D. and E. Krell, "The 3-D File System," *Proc. USENIX Summer Conference*, pp. 147-156, Baltimore, June 1989.
9. Page, T. and R. Guy, "The Ficus Scalable File System," *IEEE TCOS Newsletter*, vol. 5, no. 3, pp. 19-20, IEEE Computer Society Technical Committee on Operating Systems and Application Environments, Fall 1991.
10. Rees, Jim, Margaret Olson, and J. Sasidhar, "A Dynamically Extensible Streams Implementation," *USENIX Conference Proceedings*, pp. 199-207, Phoenix, AZ, Summer 1987.
11. Rivest, R., "The MD4 message digest algorithm," RFC 1186, Network Working Group, October 1990.
12. Satyanarayanan, M., "The Influence of Scale on Distributed File System Design," *IEEE Transactions on Software Engineering*, vol. 18, no. 1, pp. 1-8, January 1992.
13. Tichy, Walter F., "RCS - a system for version control," *Software—Practice & Experience*, vol. 15, no. 7, pp. 637-654, July 1985.
14. Welch, Brent B. and John K. Ousterhout, "Pseudo-Devices: User-Level Extensions to the Sprite File System," *USENIX Conference Proceedings*, pp. 37-49, San Francisco, Summer 1988.
15. Zalta, E.N., *Intensional logic and the metaphysics of intentionality*, MIT Press, Cambridge, MA, 1988.

Author Information

Paul Eggert is Director of R&D at Twin Sun, Inc. He got his Ph.D. in Computer Science at UCLA in 1981, and is currently working on CASE tools, groupware, and software reliability. He can be reached as eggert@twinsun.com, or by U.S. mail at Twin Sun, Inc., 360 N. Sepulveda Blvd., El Segundo, CA 90245.

Stott Parker is Professor in the Computer Science Department at UCLA. He joined the department in 1979 after getting his Ph.D. at the University of Illinois. Contact him at stott@cs.ucla.edu, or by U.S. mail at Computer Science Department, University of California, Los Angeles, CA 90024-1596.

Appendix 1: RCS Intensional File System Source Code

```
#!/usr/local/bin/perl

# IRCS extensionalizer
# $Id: IFS.usenix,v 1.3 1992/11/22 04:06:21 eggert Exp eggert $

# Disable IFS internally.
delete $ENV{'IFS_COMMAND'};

$pathname = @ARGV[0];

if ($pathname =~ /^(.*)r([^\$,\/:;@]*)$/) {
    # Map "foo,rN" to temporary containing revision N of file foo.
    $tmp = '/tmp/ifs'.getppid;
    exit if
        (print "(unlink)$tmp")
        && open(STDOUT, ">$tmp")
        && (system 'co', '-q', "-p$2", $1) == 0;
} elsif (print $pathname) {
    if ($pathname =~ /^(.*\/)(([^\./]*)$)/) {
        $dirl = $1;
        $file = $2;
    } else {
        $dirl = '';
        $file = $pathname;
    }

    if (! -d "${dirl}RCS/$file") {
        # Map to latest revision on default branch.
        exit if (system 'co', '-q', $pathname) == 0;
    } elsif (opendir(DIR, "${dirl}RCS/$file")) {
        # Map to corresponding directory.

        # Unlink any symbolic link; it's probably what invoked us.
        unlink($pathname) if -l $pathname;

        # Then create dir and subsidiary symlinks. */
        if (mkdir($pathname, -1) && symlink("../RCS/$file", "$pathname/RCS")) {
            $ok = 1;
            for (readdir(DIR)) {
                if ($_ ne '.' && $_ ne '..') {
                    s/,v$//;
                    $ok &= symlink("\$(rcs.ext $_)", "$pathname/$_");
                }
            }
            exit if $ok;
        }
    }
}

exit 2 # ENOENT
```

Appendix 2: Intensional tar File System Scenario

```

$ ls -l gnuchess-4.0.pl60.tar.Z
-rw----- 1 stott      416029 Nov 16 20:03 gnuchess-4.0.pl60.tar.Z
$ cd gnuchess-4.0.pl60
$ ls
gnuchess-4.0.pl60/
$ cd gnuchess-4.0.pl60
$ ls -l
total 2
drwx--x--x  2 stott      528 Nov 16 20:13 doc/
drwx--x--x  2 stott      168 Nov 16 20:13 misc/
drwx--x--x  2 stott     1120 Nov 16 20:13 src/
drwx--x--x  2 stott      444 Nov 16 20:13 test/
$ du .
1      ./doc
0      ./misc
1      ./src
0      ./test
2      .
$ ls -l misc | cut -c1-12,46-
total 0
lrwxrwxrwx  ChessFont -> (tar xfHOZ /tmp/gnuchess-4.0.pl60.tar.Z 260)
lrwxrwxrwx  gnuchess.book -> (tar xfHOZ /tmp/gnuchess-4.0.pl60.tar.Z 334)
lrwxrwxrwx  gnuchess.lang -> (tar xfHOZ /tmp/gnuchess-4.0.pl60.tar.Z 319)
lrwxrwxrwx  gnuchess.nunn.book.Z -> (tar xfHOZ /tmp/gnuchess-4.0.pl60.tar.Z 426)
lrwxrwxrwx  match -> (tar xfHOZ /tmp/gnuchess-4.0.pl60.tar.Z 316)
$ head -7 misc/gnuchess.book
!
! Opening Library for CHESS
!
! Copyright (c) 1988,1989,1990 John Stanback
! Copyright (c) 1992 Free Software Foundation
!
! This file is part of GNU CHESS.
$ cd src
$ ls
Makefile@      debug12.h@      dspcom.c@      gdxbkstats.c@  nuxdsp.c@
Makefile~@     debug13.h@     eval.c@        genmoves.c@    postprint.c@
README@        debug16.h@     extern.h@      gnuan.c@       search.c@
ataks.c@       debug256.h@    game.c@        gnuchess@      systems.h@
ataks.h@       debug4.h@      gbookdist.c@   gnuchess.h@    util.c@
bincheckr.c@   debug40.h@     gdbm.h@        init.c@        uxdsp.c@
book.c@        debug41.h@     gdbmbkstats.c@ main.c@        version.h@
checkbook.c@   debug512.h@    gdbmconst.h@   match@
checkgame.c@   debug64.h@     gdbmdefs.h@    membkstats.c@
debug10.h@     debug8.h@      gdbmerrno.h@   nondsp.c@
$ ls -lL *book*
-rw----- 1 stott      42755 Nov 16 20:24 book.c
-rw----- 1 stott      21787 Nov 16 20:24 checkbook.c
-rw----- 1 stott       1963 Nov 16 20:24 gbookdist.c
$ diff Makefile*
39c39
< DISTDIR=/tmp_mnt/home/fsf/cracraft/gnuchess-4.0
---
> DISTDIR=/tmp_mnt/home/fsf/cracraft/g4p60
$ pwd
/tmp/.ifs/7763/7802304/gnuchess-4.0.pl60/src
$

```