wFIFO - a Filesystem-Based Solution to Unrestricted Log Growth in MINIX.

Jeffrey Absher, DePaul University
June 2003

Abstract: *In Unix environments, application logs often quickly grow unexpectedly large. This paper reviews various available solutions that attempt to manage that growth and proposes an elemental addition to filesystems that assists in managing unexpected growth. Additionally, a prototype of the filesystem element for MINIX is developed and development notes are presented. The filesystem element is a new type of file which grows to a specific size and then upon a block allocation beyond that threshold, it deletes the first block of the file. When this filetype is appended, it acts as a finite-length FIFO queue that has elements consumed when it is written to.*

I *Introduction and motivation for a solution*

Web servers, database servers, application servers, and LAN servers, all tend to log a large amount of data for security and for troubleshooting purposes. Part of the task of system administration is to allocate disk space for these logs and to manage these logs.[1] While infinite logging would be preferred by many, it is not practical and disk space becomes a limiting factor. Most system administrators turn to archival tools and to management schemes to prevent the growth of the logs from eventually consuming a filesystem's entire allocation of space.

These archival tools and schemes do tend to work well to prevent the eventual consumption of space due to log growth that stays within predicted parameters, but logging of data usually occurs per-transaction and not per time-unit. Hence, a higher-than-expected load of transactions will lead to faster log growth than predicted.

Also, we must note the case when an error state is encountered by the system. One of the purposes of logs is to diagnose errors. So, in an error state, applications tend to log more data than they do when they are healthy. Again, more data than expected are being logged per-unit-time and unexpected growth of logs occurs. Compounding the error state of the application, now, another threat exists to the application and to the hosting machine. Log files may consume their filesystem's allocated space and this can lead to further collapse of what may have been a partially-functioning application. It may also lead to additional problem-determination and diagnosis issues during a situation such as a website outage.

The Filesystem Hierarchy Standard[3] states that for UNIX systems, "most logs must be written to [/var/log] or an appropriate subdirectory." And while the operating system itself may comply with this standard, many applications do not. Apache, for example, by default places logs in a directory that is underneath its "ServerRoot."[4] Though this can be modified with configuration settings, the fact that the default settings do not comply with standards illustrates a 'fact of life' in the computing world. Non-standard-conforming applications and their effects must be dealt with.

While having logs is important, the owner of the system must balance logging needs with the uptime of the system and the risk of consuming a filesystem's space.

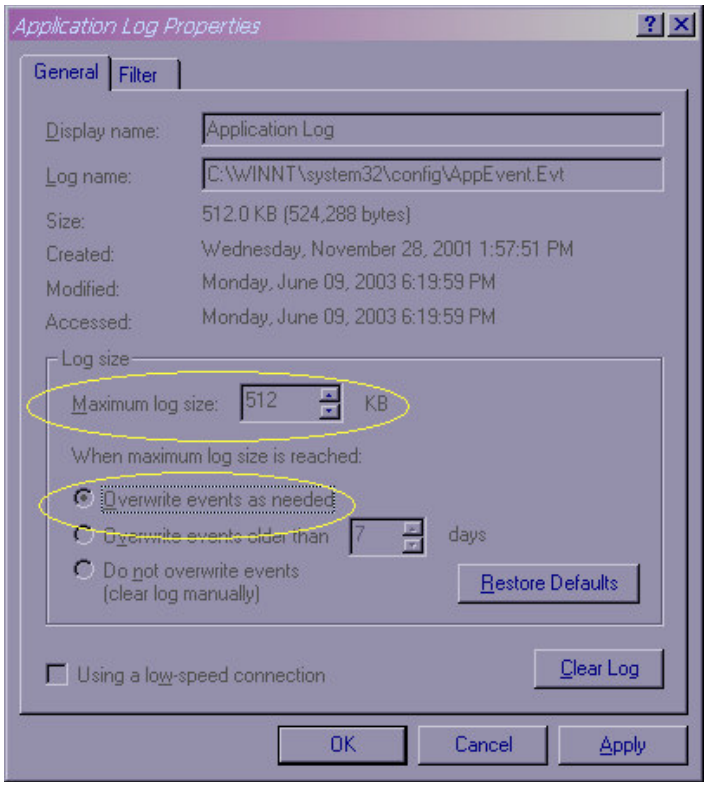
The new type of file, named ***wFIFO*** for "write-FIFOing," developed for this research can help with this problem. The owner of the file sets an attribute on the file that is the maximum amount of blocks that the file is to consume. When a block beyond that threshold is allocated to the file, a trigger in the filesystem code deletes the first block of the file and resets the file's block pointers accordingly. When the file is one that is only appended to (such as many log files) this has the effect of making the file a FIFO queue that is truncated on writes rather than on reads. Essentially, the file can be thought of as a large text-display device, but rather than being 1920 (80x24) bytes in length, its length is arbitrarily set by the file's owner. The wFIFO filetype allows system administrators to limit the unexpected growth of their logs at the cost of (presumably) the oldest data. They can still archive files using their current methods, but they will not run into a case where growth of a single file consumes the entire allocated space for a filesystem. If large enough thresholds are used, wFIFO can be used merely as a firewall-type feature in the system to prevent the compound problems described above, and truncation would only be invoked during a critical situation.

II *Similar or Related Methods and Products*

Many similar methods and products exist, but most have limitations that do not serve end-users' exact needs at times. Sometimes applications that perform logging do not do it in a standardized manner, or sometimes they will not allow redirection of the log file. For example, early versions of the Netscape http server would not redirect the log file as modern apache servers do, and if the log file was deleted (with the rm command) Netscape server would hold the i-node open, and continue to write to it, making the rm command ineffective. To free the space used by logs, the Netscape server would have to be stopped, the log copied and then the server brought back up. This caused downtime for any site served by Netscape. While Netscape (iPlanet) has since repaired this issue in their modern servers, other products still can and do suffer from the same limitations.

Microsoft Logging

Microsoft includes with their Windows NT and 2000 lines of products a system logging function which acts similarly to the wFIFO filetype. It allows the end-user to limit the maximum log size based on the age of the entry or the size of the log file.



wFIFO functionality is similar to this, but wFIFO works on any file and does not know about "records."

While this log truncation is available for system logs (OS security logs) automatically, developers for Microsoft Products must use the logging function of their particular API to log to the system's event logs (App.LogEvent in Visual Basic, for example). A drawback is that the system event logs are not text-based in Windows systems either. If an application merely logs data to a text file, the event logging functionality is of little use to limit that file's size.

Shell history logging

Command-line shells also log the commands of their users in a history file for the purposes of command recall and security. Korn Shell, bash, and csh, in unix all log history to a text file. Windows-based systems maintain history, but do not log to a file. Unix systems have a HISTFILESIZE, HISTSIZE, or savehist environment variable which will limit the history file's size by the number of entries.[11] The developers of shells do recognize the usefulness of truncating logs, but this functionality is specific to shells. This would be a good case where the shell developers could use wFIFO files and remove the log-truncation code from their products.

GDG Files

The existence of Generational Data Groups proves that automatic file truncation is not a novel idea but one that has been around for a while in some form. Generational Data Groups are a feature of OS/390. Though it is foolish to compare mainframe-level storage

with UNIX filesystems, System-managed GDG's and the cataloging provided by OS/390 can prevent the unlimited growth of files by automatically truncating (scratching) the oldest generation.[7] The OS/390 storage system is much more structured than UNIX's This makes GDG's an obvious solution because OS/390's storage system can truncate entire records at appropriate record-delimiters. Since the UNIX operating system attempts to view almost all files as ordered and unstructured data, the operating system makes no assumptions about record delimiters and leaves any format assumptions to applications. Bar[6] succinctly points this comparison. "Many other operating systems have a notion of the format of the file's contents. For example, IBM OS/390 knows different formats for records, fixed and variable. Unix has always chosen not to let the file system know the format of the file's contents for the sake of simplicity."

IBM AIX's alog[8]

Alog will perform almost exactly the same function as wFIFO in a different manner. Alog is a **piped filter** for logs, meaning that the logs must be able to be redirected to the alog binary. Though this does work for many products, there are also many products that cannot redirect their logs to a pipe. The piped filter requirement is a limitation that wFIFO does not have. Unlike the current implementation of wFIFO, alog can dynamically shrink the files that it manages. "Alog is just a command-line utility that wraps the log file if it reaches it's (sic) maximum size."[2] If alog is viewed as a part of the filesystem (which is it **not**, it is a utility provided with the operating system) it does violate the tenet of format-free files for UNIX. This is evident because the alog binary must be used to read the resulting files. Contrastingly wFIFO files can be read merely by a "cat" command because the byte order is maintained. Alog must keep a separate pointer for the head, tail, and threshold settings outside of the actual file; wFIFO keeps its threshold settings in the file's i-node.

Apache log rotation

The apache http server has the ability to pipe its logs to a filter program such as alog or the other pipe filters mentioned below. It includes a binary called rotatelogs which will automatically archive its logs based on either the time elapsed since the last archival or the size of the current log. Unlike alog, which only works with one file, rotatelogs does not truncate the logs, but instead saves them to unique filenames when the size threshold is met. Rotatelogs is a piped filter program.

Other work.

DJ Bernstein's daemontools package has some similar functionality. "Multilog saves error messages to one or more logs. … It automatically rotates logs to limit the amount of disk space used. If the disk fills up, it pauses and tries again, without losing any data." Multilog is a filter and requires that the log files be fed to it via stdin.[9]

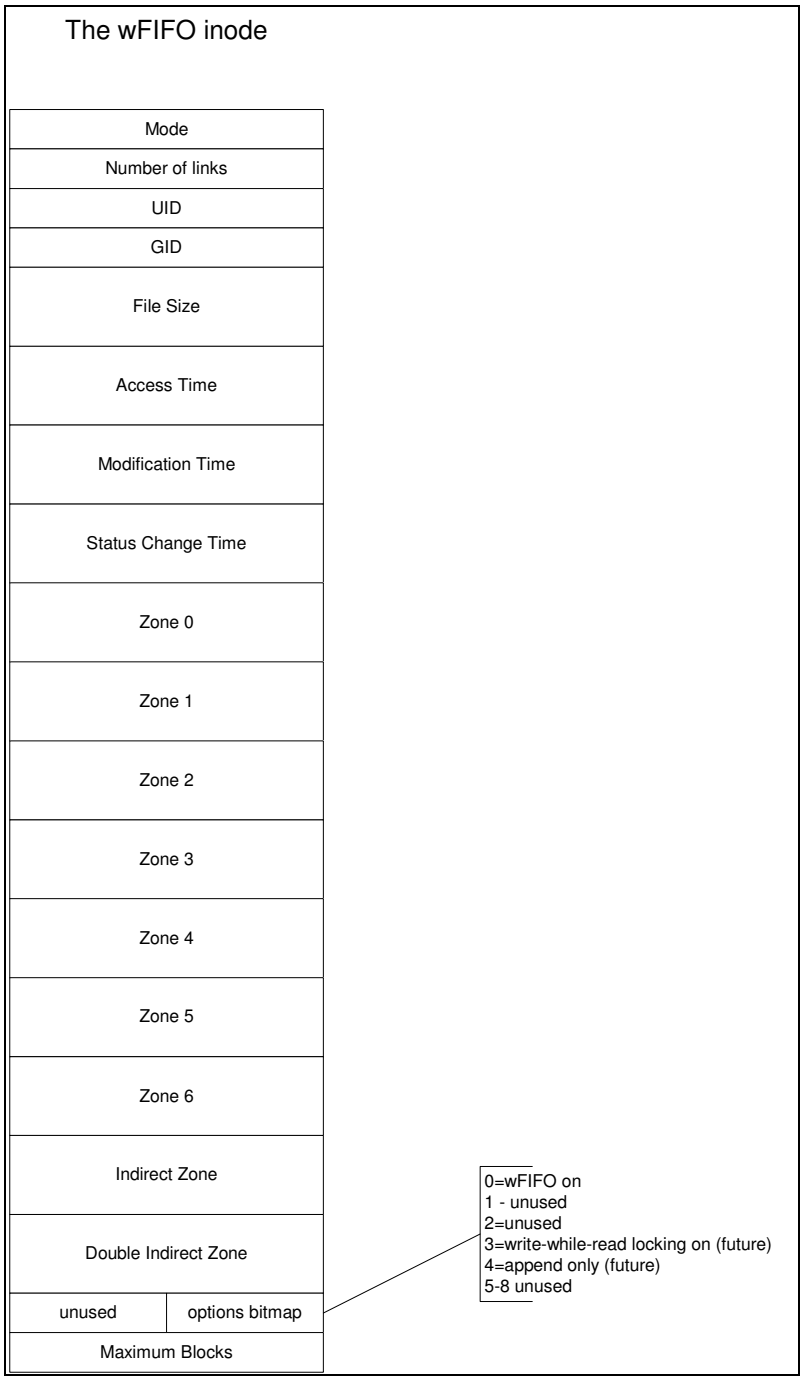
Andrew Ford has put together a log management system called cronolog based on rotatelogs from apache. It is intended specifically for web servers and like rotatelogs and like IBM's alog it is a piped filter for logs.[10]

III *Academic research into the area.*

While researching filesystems for NFS, Bennet, Bauer and Kinchlea[5] conclude that "it is imperative to implement some type of automatic archival scheme. The amount of space that can be easily freed is large and could postpone new disk and-or server acquisition." The wFIFO design does not provide archival, but it does prevent the same problems that they are trying to prevent; wFIFO allows system administers to make the decision to sacrifice the oldest data in their logs to prevent a situation of a full disk.

The experience of implementing the wFIFO filesystem into MINIX and the requirements to modify portions of the code which have little to do with filesystems support the concept of the **Virtual File System**. Unfortunately, the VFS in Linux is poorly documented, and any implementation of a filesystem requires poring over the actual code to utilize. While the purpose of VFS is to "generalize block device access," [6] it also helps to generalize the related aspects of a filesystem such as superblock maintenance and i-node requests. With VFS, many system calls sent to the filesystem are translated "into a pointer to a function specific to each filesystem handling the file in question." [6] VFS's abstraction layer is a good idea and any production-level implementation of a wFIFO filesystem should be written for systems with VFS interfaces.

IV *wFIFO solution package detail*



The existence of multiple programs and methods for log restriction, rotation and archival indicates that this problem is not yet fully solved. By moving an element of log restriction into the filesystem, wFIFO offers a small step toward a generally acceptable solution without sacrificing the concept of unstructured files in UNIX.

wFIFO provides an automatic truncation setting on a file at the filesystem level with the truncation occurring at the beginning of the file (head) as new data is written to the end of the file (tail). A portion of the file's i-node is allocated to hold a value representing the

number of blocks that the file will grow to. Every time the file is written to, the filesystem driver checks to see if the writing position pointer is pointing to a position beyond the maximum threshold represented by the block limit in that file's i-node. If this is the case, the filesystem allocates the new block, and writes to it as it normally would, but prior to returning control to the calling function, the filesystem frees the first block of the file and rotates all of the block pointers within the file's i-node and its indirect blocks 'down' one block location. This effectively makes the file a fixed-size queue of blocks.

wFIFO filesystem pseudocode

```
within read-write call, AFTER new allocation and write of block:
if (inode.checkwfifo is set and filepositionpointer > BLOCKSIZE *
inode.Maximum Blocks)
        walkerindex = 0
        throwaway = inode.getblockaddr(0)
        while( walkerindex++ < Maximum Blocks )
                inode.setblock( walkerindex,  inode.getblockaddr(walkerindex–1))
        freeblock(throwaway)
        inode.filesize = inode.filesize – blocksize
        filepositionpointer = filepositionpointer – blocksize
end
```

Other additions needed for wfifo

wFIFO needs a supporting new filesystem type and the various tools for filesystems need to be modified (mount, fsck, mkfs, partitioning, etc). A new system call to set and check the wFIFO bytes in the inode is needed. A new simple binary to invoke that system call is needed.

do_wFIFO systemcall (file, on, size) pseudocode

```
        if file.filesystem.readonly() return( err )
        if file.filesystem != wfifo return (err)
        if (caller != super_user && caller != file.owner) return( err )
        if (file.size > size * BLOCKSIZE) return( err )
        file.inode.chwfifobit = on;
        file.inode.MaximumBlocks = size
        file.lastupdatetime = now
        file.inode.dirty = true
        return( OK )
```

V *Results*

For the implementation, MINIX is used. Following are some images of the key functionality of the wFIFO filesystem in MINIX.

The "part" screen with a large new filesystem created:
(3F was chosen as the numeric type because a 3 is a rotated "w".)

```
  Select device       ----first----   --geom/last--   ------sectors-----
    Device             Cyl Head Sec    Cyl Head Sec      Base      Size       Kb
    /dev/hd0                           520   32  63
                         0    0   0    519   31  62         0   1048320   524160
Num Sort   Type
 1* hd1  81 MINIX        0    1   0     63   31  62        63    128961    64480
 2  hd2  81 MINIX       64    0   0    190   31  62    129024    256032   128016
 3  hd3  81 MINIX      191    0   0    317   31  62    385056    256032   128016
 4  hd4  3F wFIFO      318    0   0    519   31  62    641088    407232   203616

Type '+' or '-' to change, 'r' to read, '?' for more help, '!' for advice
```

fsck and fsck3 written to accommodate the new filesystem:

```
# fsck /dev/hd4
Please use fsck1, fsck2 or fsck3.
# fsck3 /dev/hd4
warning: huge directory: / (ino = 1)
warning: . has offset 6144 in / (ino = 1, . is linked to 1)
warning: .. has offset 6160 in / (ino = 1, .. is linked to 1)

Checking zone map
Checking inode map
Checking inode list

blocksize =  1024        zonesize  =  1024

     4    Regular files
     2    Directories
     0    Block special files
     0    Character special files
 25449    Free inodes
     0    Named pipes
     0    Symbolic links
201833    Free zones
# _
```

Mounting the new filesystem and the df (display filesystem) command:

```
# mount
/dev/hd1a is root device
/dev/hd1c is mounted on /usr
# mount /dev/hd4 /newfs
/dev/hd4 is read-write mounted on /newfs
# mount
/dev/hd1a is root device
/dev/hd1c is mounted on /usr
/dev/hd4 is mounted on /newfs
# df

Device     Inodes  Inodes  Inodes     Blocks  Blocks  Blocks  Mounted      V Pr
           total   used    free       total   used    free      on
           -----   -----   -----      -----   -----   -----   -------      - --
/dev/hd1a    480     204     276       1440     438    1002   /            2 rw
/dev/hd1c  10512    3325    7187      63040   27879   35161   /usr         2 rw
/dev/hd4   25455       6   25449     203616    1783  201833   /newfs       3F rw
# _
```

The “simple” chwfifo command to invoke the system call, no arguments gives syntax.: The current implementation has only 1 option, but future implementations may have options such as “force append only” or “read while write blocking.”

```
# chwfifo
Usage: chwfifo opts blocks file
for opts, "or" the following bitmaps together:
        00000001 = pruning on
use "0" in blocks for current status
        1-65536 for blocks (zones)

Dont try on dirs or special files
# _
```

Chwifo again: Note that it fails on a non-3F filesystem with a return value of -1.

```
# pwd
/oldfs/wfifo_test
# ls -la
total 6
drwxr-xr-x  2 root     operator       48 Jun 12 12:30 .
drwxr-xr-x  3 root     operator       48 Jun 12 12:30 ..
-rw-r--r--  1 root     operator     4067 Jun 12 12:30 oldfsfile
# chwfifo 0 0 oldfsfile

Name: oldfsfile
Checking file settings:RO: 0 FSVer: 2 own: 0  type: OK zones = 4
Systemcall returned -1
# _
```

The full functionality: Change to a directory on the 3F filesystem and create 4 blank files. Then turn on truncation at the 9 zone level for “on9z”. Nine-zone truncation, though short, will allow one indirect zone to be accessed, (MINIX has 7 direct zones within the i-node.)

```
# pwd
/newfs/wfifo_test
# > off
# > on9z
# > onSI40z
# > onDI300z
# chwfifo 0 0 off

Name: off
Checking file settings:RO: 0 FSVer: 9 own: 0  type: OK zones = 1
Enable:0 zones:0
Systemcall returned 0
# chwfifo 1 9 on9z

Name: on9z
Checking file settings:RO: 0 FSVer: 9 own: 0  type: OK zones = 1
Enable:1 zones:9
Systemcall returned 0
# _
```

Now turn on truncation at various levels for the other two files, forcing multiple single indirect (SI) zones and double-indirect (DI) zones to be used:

```
# chwfifo 1 40 onSI40z

Name: onSI40z
Checking file settings:RO: 0 FSVer: 9 own: 0  type: OK zones = 1
Enable:1 zones:40
Systemcall returned 0
# chwfifo 1 300 onDI300z

Name: onDI300z
Checking file settings:RO: 0 FSVer: 9 own: 0  type: OK zones = 1
Enable:1 zones:300
Systemcall returned 0
# _
```

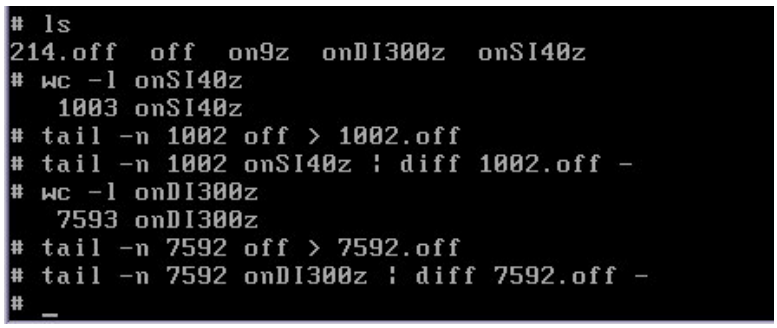
Now start filling up all of the files: Garbageman is a program written to assist with testing; garbageman generates 10000 lines of random text, each line is of random length but with a maximum length of 80. 432 is the random seed. Tee causes the command to write to all four files simultaneously. Note the files' sizes after the "fill" data is tee'd into them.
A zone is 1024 bytes. 8 zones < on9z < 9 zones, 39 zones < onSI40z < 40 zones, and 299 zones < onDI300z < 300 zones.
Another check is that because partial zones are written at the latest period in time, all three wFIFO files are exactly 3 bytes over a zone-boundary.
This makes sense because the head of the queue is truncated at zone boundaries, but the tail of the queue is written to with an arbitrary amount of data.

```
# garbageman 432 10000 80 ¦ tee off on9z onDI300z onSI40z > /dev/null
# ls -la
total 753
drwxr-xr-x  2 root     operator       96 Jun 12 12:38 .
drwxrwxrwx  3 bin      bin          6176 Jun 12 12:05 ..
-rw-r--r--  1 root     operator   404483 Jun 12 12:45 off
-rw-r--r--  1 root     operator     8195 Jun 12 12:45 on9z
-rw-r--r--  1 root     operator   306179 Jun 12 12:45 onDI300z
-rw-r--r--  1 root     operator    39939 Jun 12 12:45 onSI40z
# _
```

Now check that the last "n-1" lines of the files match: The first line of text may have included a zone (block) boundary and hence will probably not match as the truncation was likely to consume part of it. Wc counts the number of lines and tail extracts the last 214 lines from the file. Diff will print to the screen if differences are found, here it does not find any.

```
# ls
off  on9z  onDI300z  onSI40z
# wc -l on9z
    215 on9z
# tail -n 214 off > 214.off
# tail -n 214 on9z ¦ diff 214.off -
# _
```

Check that integrity is maintained on the files that stress the indirect blocks:



The functionality of chwfifo is demonstrated and testing is complete.

VI *Conclusion, Limitations, Further Research*

Filesystem truncation for a FIFO file at the file's head upon writes is something that can be done successfully and without excess effort. This technique could offer another option to system administrators to protect their systems from downtime and from compounded problems. If currently-existing i-nodes have space, this functionality could be 'shoehorned' into current filesystems with minimal effort. Other than the related functions and system calls, the actual code is straightforward. Upon a zone allocation, one additional check is needed. If the file needs to be truncated, the truncation is efficient in terms of disk-accesses because only 2 zones are actually written, the rest of the processing is performed in the i-node. Then the kernel must perform some dynamic updating of position pointers (these new pointers may have to cross back 'up' the VFS abstraction layer.) Complexity of the solution is not a significant barrier to implementation.

Other solutions to the problem of unchecked growth exist, and some have been reviewed here. The larger problem is the appropriate manner to add structure to a file so that the OS can view the file as either an ordered stream of bytes or as something with structure such as a set of records. wFIFO sidesteps this problem by adding data to the i-node with a system call, but an i-node is still some form of structure. Since this structure is already present in unix, wFIFO does not violate the tenet of unstructured files in unix any more than the OS itself does. But, the more information that we put into an i-node, the more we violate the structure-free-file tenet. Recent Linux's have also have found a nice way to add structure by allowing the user to "attach arbitrary name/value pairs as properties of a filename."[12] NTFS has alternate data streams that would allow such data to be stored and accessed. ADS's are used for attributes such as thumbnails and version information. The problem then becomes one of standardization of attribute names. If we were to create an attribute called wFIFO and then the filesystem truncates the file based on that, as long as no other programs "step on" that attribute then wFIFO would work well. But if, as a matter of course, the filesystem manipulates files based on a hidden attribute that is not protected by a system call, that attribute could be manipulated by users or by other programs and then the filesystem will behave unpredictably. Since wFIFO is integral to the filesystem and is not an application-level action, I recommend protecting the wFIFO settings from other programs by keeping its functionality wrapped in a FS-specific system call.

Should we add structure to Unix files? That has already happened in some cases and it appears to be inevitable that such attempts will continue. The interesting questions are where should the structure be added and at what level should that structure be accessed and preserved. The obvious answer to these questions, for now, is that structure which is required by the operating system either for security, caching, truncation, timestamping, archival, and other features of the operating system should end up in the i-node and be protected by system calls. Structure which is optional or structure which is used by applications should be maintained in properties or attributes which are accessible to common programs. Unix itself should not move away from viewing a file as an ordered stream of bytes. Attributes instead could be considered internal (i-node) and external (attached name-value pairs) and placed appropriately with the majority of them being external.

A different future direction that may be of interest for wFIFO in particular is to make it more intelligent and allow it to recognize record delimiters. An additional attribute such as a delimiter byte could be specified. With some modification of the read and write position-pointer arithmetic, the filesystem could present the illusion that the truncated file starts at a record delimiter rather than at the start of a disk block. An example is to use <CR> as a delimiter. If, after truncation, the first <CR> occurs 32 bytes after the start of the first block, position pointers could be manipulated to start 32 bytes later than they used to. There would be some significant engineering and arithmetic needed to make this work, but it is feasible.

The purpose of this research is to test the viability of a wFIFO system by implementing a prototype and to understand the costs of adding this functionality at the filesystem level. I find that it is viable and that the prototype is successful. An argument can be made that since there is already archive information (date information) kept in i-nodes, archival is a valid feature for the operating system to support. If archival is a valid feature for the OS to support, truncation is part of archival and may be supported as well at the OS level without violating design concepts. Any wFIFO implementation in a production-level environment should obey stricter software engineering rules and should offer more robust argument-checking and error checking than this implementation does.

References:

1. Limoncelli, T., Hogan C., *The Practice of System and Network Administration*, Addison Wesley 2002 ISBN 0-201-70271
2. Email Communication with James Shaffer of IBM, responsible developer for alog. May 9 2002. jabsher@us.ibm.com and jjs@austin.ibm.com
3. Filesystem Heirarchy Standard Group eds. Russel R., and Quinlan D. "Filesystem Heirarchy Standard – Version 2.2" May 2001
4. Apache Log files documentation
   http://httpd.apache.org/docs-2.1/logs.html
   http://httpd.apache.org/docs-2.1/mod/core.html#errorlog
   referenced 10th June 2003.
5. Bennett, J., Bauer, M., Kinchlea, D. "Characteristics of Files in NFS Environments" 1991 ACM 089791-396-5/91/0006/0033.
6. Bar, M, *Linux File Systems*, Osborne/McGraw Hill 2001 ISBN 0-07-212977-7
7. IBM, *OS/390 V2R10.0 DFSMS Using Data Sets* Document Number SC26-7339-00.
8. IBM, *AIX Commands Reference Volume 1*
   http://publibn.boulder.ibm.com/doc_link/en_US/a_doc_lib/cmds/aixcmds1/alog.htm
   referenced 10th June 2003.
9. daemontools homepage
   http://cr.yp.to/daemontools.html
   referenced 10th June 2003
10. Ford & Mason Ltd, cronolog usage document
    http://www.cronolog.org/usage.html
    referenced 10th June 2003
11. Peek,J., O'Reilly, T., Loukides, M., *Unix Power Tools* 2nd edition, O'Reilly, August 1997 (chapter 11) ISBN 1-56592-260-3
12. Raymond, E.S. *The Art of Unix Programming*, Prentice Hall 2003. ISBN: 0-13-142901-9 (chap 20)

Other Key Reading (not referenced)

1. Ruffin, M.. "A Survey of Logging Uses" Broadcast Technical Report 36 Espirit Basic Research Project 6360 February 21, 1995.
2. Oxman, G. "The extended-2 filesystem overview" Aug 3 1995

3. Card, R., Ts'o T., Tweedi S. "Design and Implementation of the Second Extended Filesystem" *Proceedings of the First Dutch International Symposium on Linux*, ISBN 90-367-0385-9
4. Rosenblum M., OusterHout J. "The Design and Implementation of a Log-Structured File System"
5. Tanenbaum, A, Woodhull, A, *Operating Systems Design and Implementaion* – Second Edition, Prentice Hall 1997