Watchdogs – Extending the UNIX File System

Brian N. Bershad and C. Brian Pinkerton
University of Washington

ABSTRACT: The traditional UNIX file system provides operations whose semantics are fixed at file system implementation time. *Watchdogs* are user-level processes that can extend the file system to achieve user-defined file system semantics on a per-file basis. Watchdogs provide only those functions that need special handling. Other operations proceed through the normal file system, unimpeded by the existence of watchdogs. We describe a prototype implementation of watchdogs that has been used to build several useful applications. Although only a prototype, the system has acceptable performance.

This work took place while the authors were supported by NSF Grants No. DCR-8420945 and CCR-8611390. Bershad was also supported by a USENIX Association fellowship. Computing equipment was provided by Digital Equipment Corporation's External Research Program.

An earlier version of this paper was presented at the 1988 Winter USENIX Technical Conference (Dallas).

1. Introduction

The traditional UNIX file system serves as a repository for passive data objects – files. It provides facilities for naming, protecting, storing, and accessing these files. The file system defines the semantics in each of these areas, constraining users to the design decisions of the file system's implementors.

Watchdogs are extensions to the 4.3BSD UNIX file system that allow users to define and implement their own semantics for files. A watchdog is a user-level program associated with a file or directory. This program can provide alternative implementations of the file system's naming, protection, storage and access functions for that file or directory. The distinguishing characteristics of watchdogs include:

- the ability for unprivileged user-level programs to serve as watchdogs,
- added expense only for those functions that are redefined,
- complete transparency to programs accessing associated files, and
- interprocess communication (IPC) cost commensurate with that of the other major IPC mechanism – pipes.

Watchdogs can provide users with the types of features generally made impossible by the requirements that a file system be at once fast, simple and general. For example, a compaction watchdog can transparently compact and uncompact a file as it is being written and read. Users can specify their own file protection policies by creating a watchdog that redefines the *open* system call. Intelligent mailboxes, where actions are taken automatically upon the receipt of new mail, are also possible using watchdogs. Watchdogs make file versioning possible without burdening either the operating system (as in VMS [Levy & Eckhouse 1980]) or the user (as in SCCS [Rochkind 1975]) with the responsibility of maintaining past versions. Watchdogs also facilitate the creation of special-purpose pseudo file systems, where the files that comprise the file system need not exist within any single UNIX file system. To this end, watchdogs have been used as part of the Heterogeneous Computer Systems project at the University of Washington [Black et al. 1988] to allow UNIX machines to import files from heterogeneous file servers.

2. *Watchdog Operation*

The central object in the extended filesystem is the *guarded* file. A guarded file is an existing file or directory with which a watchdog has been associated. This watchdog is notified every time the file is opened and interacts with the kernel to provide extended semantics for the file.

A file consists of some data and some operations allowing processes to access that data. In the absence of watchdogs the implementation of any one file's operations are common to all files in the system. Watchdogs give each file the chance to provide its own implementation of procedures that implement the standard file system interface (open, close, read, write, etc.). Figure 1 diagrams the relationship between processes, the file system and watchdogs.

Watchdogs can interact with user-level processes only through the kernel because their involvement must be transparent to the user program. The strength of this transparency is demonstrated in two ways: user programs need not be recompiled to take advantage of watchdogs, and user programs cannot circumvent the watchdog system to access files directly.

When a file guarded by a watchdog is opened, the opener is suspended and the watchdog is notified of the open by a message from the kernel. The message contains the arguments to the open call and the identity of the opener. With an acknowledgment to the kernel, the watchdog either denies or permits the requested

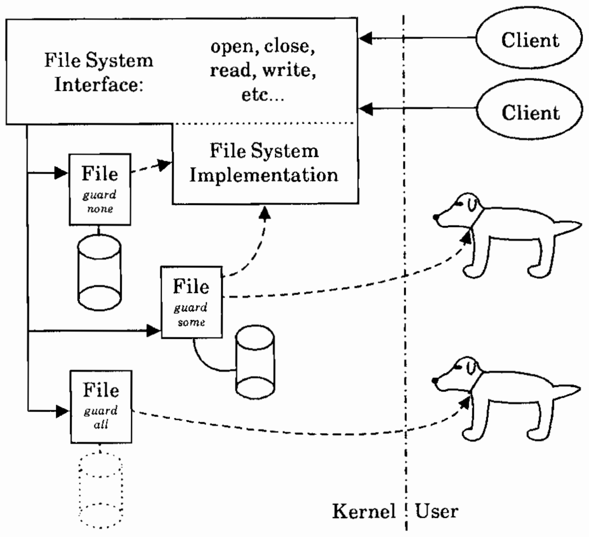

Figure 1: Processes, Watchdogs and the File System

access. Should access be granted, the watchdog informs the kernel of any other operations *on the opened file* that should be guarded by the watchdog. A watchdog may guard different instances of an open file in different ways. This allows a single file to have multiple views.

A request for a guarded operation on the opened file is relayed to the watchdog for processing. Upon being notified of the request, the watchdog must do one of the following:

1. perform the operation, providing or consuming any data normally associated with the operation. For example, a watchdog would satisfy a read request by returning a buffer of data to the reading process via the kernel. Whether the file originally opened actually sources or sinks this data depends on the implementation of the watchdog. To avoid loops, watchdogs are permitted direct access to the files they guard.
2. deny the operation, replying to the kernel with the UNIX error code that should be relayed to the process requesting the operation.

3. defer the operation, when possible, back to the kernel. In this case, the watchdog simply acknowledges the operation, but relies on the kernel to actually perform it. Deferment is appropriate when only the occurrence, but not the actual function, of an operation deserves attention (such as in accounting applications).

In all three cases, the process requesting the operation is unaware that its request is being evaluated by another process. Since only guarded operations are relayed to the watchdog, others proceed at their full speed, unimpeded by the existence of the watchdog.

3. Directory Watchdogs

Watchdogs can be associated with directories as well as with files. When a process opens a file, the process's access rights are checked for each directory in the file's pathname. If any of these directories is guarded by a watchdog, the watchdog is asked to validate the access attempt. This may require that several watchdogs be consulted during the resolution of a single pathname. If the parent directory of an opened file is guarded, and the file does not have an explicit watchdog, the parent directory's watchdog is used to negotiate access to the opened file. This arrangement allows one watchdog to collectively manage all files in a single directory.

Watchdogs permit a file's contents to be illusory. A process sees whatever the watchdog chooses to let that process see. Directories are no exception. A process can open a physically non-existent file in a physically non-existent directory as long as the last resolvable component of that file's name has an associated watchdog capable of handling the deception. Note that in this case all operations must be guarded, and the watchdog may not defer any activity back to the kernel.

To demonstrate the behavior of the different styles of watchdog attachment, consider the directory tree shown in Figure 2. A user trying to open */X/file1* would first be authenticated against *bowser*. If access to the directory is granted, *fido* is notified of the open attempt and can then inform the kernel if final access is approved. In the case of */X/file2*, *bowser* is responsible for both granting access to the file and assuming any of the file system operations that it chooses. This is because *file2*, though unguarded, lives in a guarded directory. A user trying to open */X/Z/file3* will only be checked for access when passing through */X*. The file itself is unguarded. Finally, attempts to access */Y/file4* or */Y/file5* would be handled by the watchdog *spot*.

4. Related Work

Many other systems have concentrated on moving a piece of the file system up to the user-level in order to obtain some new functionality. The principal difference between watchdogs and these efforts is that watchdogs provide a kernel framework for such extension while the others build a veneer on top of existing kernel facilities. For example, IBIS [Tichy & Raun 1984] and the Newcastle Connection [Brownbridge et al. 1982] implement remote file systems at user-level by modifying the standard subroutine libraries. The Apollo DOMAIN system [Rees et al. 1986] allows users to define typed objects (files) and operations (procedures) to manipulate these objects. The system relies on dynamic loading to bind an executing program to the appropriate implementation of an object's procedures.

The advantage of these other systems is that the interface and implementation of the operations are coresident in a single

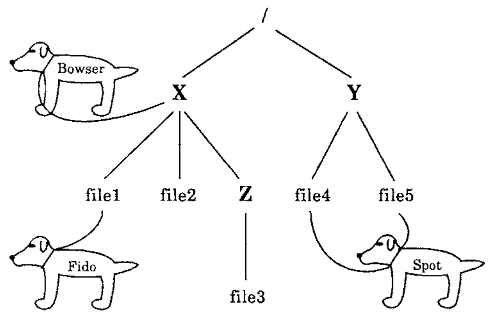

Figure 2: Watchdogs In The File System Tree

process so there is no context switch or IPC overhead. Unfortunately, users are still able to circumvent the extended interface, accessing files directly using the basic file system calls, or they may be prevented from accessing their files unless explicitly relinking their programs with special libraries. Hence, these systems are neither secure nor transparent. In contrast, watchdogs purposefully involve the kernel to obtain both security and transparency.

Operating systems that permit the entire file system to reside outside the kernel, such as Mach [Accetta et al. 1986] or Amoeba [Tanenbaum & Mullender 1981], are still not easily extendible. Although not kernel resident, these file systems are non-trivial programs unlikely to be modified by unsophisticated users. Because watchdogs are intended to handle isolated file system functions, they can be made simple enough so that even a novice programmer can master them.

The watchdog approach to directory management subsumes the unimplemented portal mechanism described in the original 4.2BSD UNIX documentation [Joy et al. 1984] and is similar to Apollo's extended naming facility in the DOMAIN system.

5. *Implementation*

The demands of watchdogs are unique among all facilities in the UNIX operating system. Watchdogs must be associated with files; the kernel must be able to transparently effect communication between the process using a file and the watchdog controlling it; there must be some mechanism for the creation and management of watchdog processes; and finally, the system must be robust enough so that error conditions arising from the users' and kernel's reliance on potentially unreliable watchdogs do not cripple the system. The software framework to provide these facilities has several principal components:

- a new system call to link watchdogs to files,
- a message-based kernel/watchdog communication mechanism, and
- a system-wide *chief* watchdog process, responsible for starting new watchdogs and managing ones already running.

This section describes the design and implementation of these components.

5.1 Binding the Watchdog to the File

The watchdog associated with a file is a characteristic of that file, much like its owner or protection mode. Consequently, the information belongs in the file's inode, where it can be modified only through secure system calls. The current implementation of 4.3BSD UNIX reserves 20 bytes in the inode for "future use." Since watchdogs are an experimental system, and since we did not want to make major changes to the inode subsystem (that is, reformat the disk), those 20 bytes are used to record the name of the executable image containing the watchdog. This imposes an annoying restriction on the length of a watchdog's name. This problem is circumvented by maintaining a public directory, */wdogs*, containing symbolic links to real watchdogs scattered throughout the system. Access control to this public directory is, of course, managed by a watchdog.

To bind a watchdog to a file, a program makes the *wdlink()* system call:

```
int wdlink(char *file, char *watchdog);
```

A user program by the same name provides this interface from the shell. A user must be the file's owner (or root) to link in a watchdog. *wdlink* with a null second argument unlinks any watchdog. The *ls* command has been augmented with yet another flag so that the name of a guarding watchdog can be seen on a directory listing.

```
%ls -lwd /wdogs
drwxrwxrwx  1 root  1024 Dec 25 11:43
                           /wdogs <- /wdogs/wd_mgr
```

Since the name of the watchdog is kept in the file's inode, there is no limit to the number of files that may be guarded by a single watchdog. The implementation does, however, impose a limit (one) on the number of watchdogs that may guard a file. Having multiple watchdogs share responsibility for a single file might be useful in certain situations, but the added utility did not seem to warrant the extra complexity.

5.2 Kernel/Watchdog Communication: Watchdog Message Channels

Typically, communication between processes and the UNIX kernel is asymmetric and haphazard, relying on a large number of not very orthogonal system calls and an asynchronous signalling mechanism. The former allows processes to manipulate kernel facilities, while the latter provides for very simple message passing from the kernel to a process. Watchdogs require a richer form of communication. A process makes a system call requesting that some operation be performed on a file. The kernel, acting as a switch, routes that request either to the file system, normally, or to the watchdog, for a guarded operation. In the latter case, the watchdog eventually responds to the kernel with results to be relayed back to the user, or with a response instructing the kernel to take action on the request.

The ability for the kernel and a process to treat one another as peers in a message-based communication environment exists in many other operating systems [Accetta et al. 1986, Rashid & Robertson 1981, Baskett et al. 1977], but is not present in 4.3BSD UNIX. There were three possible ways to fill this void: implement general message passing between processes and the kernel, exploit an existing communication mechanism, or construct a message system specifically tailored for watchdogs.

The first approach would have been to implement a completely general message passing mechanism, similar to those present in other systems, replacing or augmenting the existing procedure call interface with a message-based one. Because this would involve fundamental changes to the system, because we wished to retain compatibility with the original 4.3BSD system, and because the implications of such a major redesign effort were beyond the requirements of the project, this choice was rejected.

A second approach would have been to use existing socket code so that the kernel and a watchdog would communicate via standard UNIX sockets. Concern for performance kept us from adopting this approach. The characteristics of kernel/watchdog communication did not appear to require the generality provided by sockets.

The third choice, and the one taken, was to build a special-purpose message passing system. With this, optimizing for the anticipated message characteristics was possible. A watchdog communicates with the kernel via a Watchdog Message Channel (WMC). There is one WMC per watchdog. A WMC is created with the *createwmc()* call and referenced by a standard UNIX file descriptor using the operations read, write, close, etc. A new kernel descriptor type, *DTYPE_WDCHANNEL*, exists to support these operations on the channel. The kernel sends messages to the watchdog on a WMC to relay user requests and data for guarded operations, and the watchdog uses its channel to respond to those requests.

Messages contain a type field, a session identifier and the message contents. The session identifier permits the kernel and the watchdog to multiplex the activity of multiple files over a watchdog's single WMC. When a file is first opened, the kernel assigns to it the session identifier used in the open message. All subsequent messages between the kernel and watchdog referencing that open file contain the same identifier.

Messages may be sent by either the kernel or the watchdog. For example, the kernel can send to the watchdog a message of type *Read_Request*. The message includes a file's session identifier, current offset, and number of bytes to read. The watchdog may choose to respond with a simple message in the form of an acknowledgment, denying or deferring the operation, or it may respond with the actual data to be returned to the reading process.

A watchdog has dynamic control over where and how its address space is utilized on behalf of processes accessing files. In the case of a *Write_Request*, a watchdog's positive, non-deferring acknowledgment includes a pointer to where the written data should be placed in the watchdog's address space. For example, a process may write 40k bytes to a file in a single operation, but the watchdog may only be equipped to handle 4k bytes at a time. This restriction is enforced and the process's write system call always returns the number of bytes transferred into the watchdog's address space. Programs that do not properly check the return value of the write call, instead relying on the fact that the file system may impose no limit on the length of a written buffer, are technically in error and may be so exposed if they access a file guarded by a buffer-limited watchdog.

The structures required to associate a process with a watchdog during an open file session are illustrated in Figure 3. The solid lines indicate links needed to forward a process's access request on a guarded file to the actual watchdog, while the dashed lines reflect those needed to relay a watchdog's response back to the process. The user's per-process open file table points into the system-wide open file table. Watched entries in the system's open file table include a pointer into the watchdog session table. Each session table entry references the kernel's end of the WMC, where unread messages are kept. The WMC entry also contains a pointer to the process representing the watchdog. The watchdog's WMC indirectly references the entry in the WMC table through the system's open file table. A message arriving from the watchdog includes the session identifier, which maps into the session table. Each session table entry has a back-pointer to the appropriate entry in the system's open file table, allowing watchdog responses to be returned to the correct process.

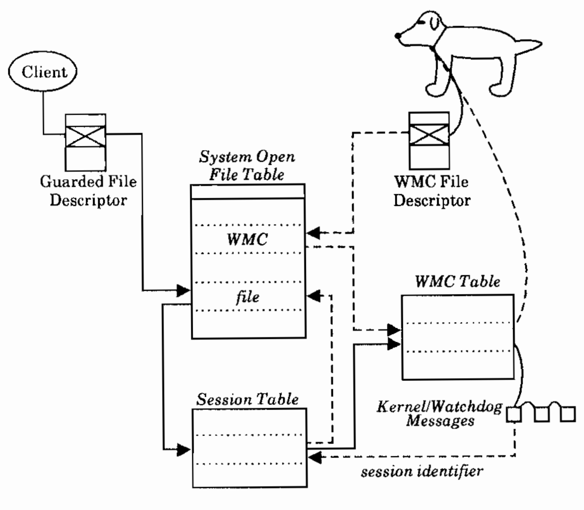

Figure 3: Watchdog Data Paths

5.3 *Managing Watchdog Processes*

The kernel assumes a minimal role in the management of watchdog processes. It must be able to map from a watchdog's name onto that watchdog's WMC. This is done by remembering the device and inode number of a watchdog as it requests a WMC. Later, when a guarded file is opened, the device and inode number of the watchdog named in the opened file's inode are compared against those of the currently active WMCs. If a match is found, the watchdog is running and a new session is begun. Otherwise, the watchdog is not yet running and must be started.

Since a large part of the watchdog framework includes a kernel to process communication mechanism, this structure is used to allow a normal user-level process to manage all watchdogs, much like *inetd* manages network daemons. This manager process, known as the chief watchdog, communicates with the kernel through a normal WMC, having first announced itself as the chief by sending an *I-am-chief* message on the channel. On the opening of a guarded file for which no watchdog is currently running, the kernel sends a message to the chief indicating that a new watchdog should be created. The chief *execs* the requested watchdog which then opens its own WMC. The watchdog's first read on its new WMC returns the message describing the outstanding open request. If the chief is unable to start the watchdog, or if the watchdog terminates abnormally, the chief notifies the kernel of the failure, returning to it the UNIX error code that should be relayed back to the opening process. Once running, a watchdog should not terminate until it has no active sessions. If a watchdog dies in the middle of a file session, processes referencing the file will, on their next access attempt, receive either an EOF or an error depending on the type of access method (i.e. *read* returns EOF, while *fstat* returns an error).

Creating a new watchdog process on every open is expensive, at least two orders of magnitude slower than utilizing an existing watchdog. In situations where a watchdog is frequently used, the overhead of involving the chief and constructing a new process each time is unacceptable. By monitoring watchdog creation requests from the kernel, the chief may allow a heavily used watchdog to continue to run even though it might not be currently active. The chief's benevolence is based on the expectation that a frequently opened file is likely to be opened again in the near future. Thus, when the watchdog is required again there is no need to start a new process or even involve the chief.

Although the chief is normally responsible for unleashing most watchdogs, there is no restriction that prevents users from running them directly. Users can debug watchdogs by running them from within the shell or their favorite debugger. This is the normal mode for watchdog development.

5.4 *Writing Watchdogs*

The internal structure of a watchdog is similar to a server in a networked environment. The watchdog first creates a rendezvous port (WMC), and then listens patiently on that port awaiting requests. The following code fragment highlights the structure of a watchdog's implementation.

```
struct wdmsg wdmsg;
intcc;
intwmc = createwmc();
for (;;) {
    cc = read(wmc, &wdmsg, sizeof(struct wdmsg));
    if (cc != sizeof(struct wdmsg)) {
        if (cc < 0)
            perror("read"), exit(-1);
        else
            abort();  /* should NEVER happen */
    }
    switch (wdmsg.wm_type) {
    case WDMSG_OPENREQ:
        do_open(wmc, &wdmsg); break; /* open */
    case WDMSG_READREQ:
        do_read(wmc, &wdmsg); break; /* read */
    case WDMSG_STATREQ:
        do_stat(wmc, &wdmsg); break; /* stat */
    case WDMSG_CLOSEREQ:
        do_close(wmc, &wdmsg);break; /* close */
    /* other guarded operations go here */
    default:    /* should NEVER happen */
        abort();
    }
}
```

Each of the functions in the switch takes care of one file operation and then responds back via the WMC with an acknowledgment directing the kernel to take an appropriate action on behalf of the process wishing to access the file.

A watchdog emulating the special file */dev/null* trivially implements the read and write functions as:

```
do_read(wmc, wdmsgp)
    intwmc;
    struct wdmsg*   wdmsgp;
{
    /* cast message components into something short * /
    struct wdrwmsg  *wi =     /* read/write msg */
        (struct wdrwmsg*)(&wdmsgp->w_un.wi_rw);
    struct wdackmsg *wa =     /* ack msg */
        (struct wdackmsg*)(&wdmsgp->w_un.wa_ack);
    struct wdrr  wdrr; /* req/reply ack including data */
    interror = 0;
    char   maxbuf[BUFSIZ];
                /* set return session id */
    wdrr.wdrr_sid = wdmsgp->wm_sid;
    if (wdmsgp->wm_type == WDMSG_READREQ){
            /* /dev/null read returns 0 bytes */
        wdrr.wdrr_len = 0;
        wdrr.wdrr_data = (char*)NULL;
        /*
         * acknowledge with data to return to process
         */
            /* Put Data */
        error = ioctl(wmc, WDIOCPDATA, &wdrr);
    } else{
        /* beware programs that don't
           check return values! */
        wdrr.wdrr_len = MAX(wi->wi_size, BUFSIZ);
        wdrr.wdrr_data = maxbuf;
        /*
         * ack with reference to where data should be put
         */
                /* Get Data */
        error = ioctl(wmc, WDIOCGDATA, &wdrr);
    }
    if (error)   {
        /* Data transfer failed.  Must ack explicitly */
        wdmsgp->wm_type = WDMSG_WDACK;
        wa->wa_status = error;    /* returned to user */
        write(wmc, wdmsgp, sizeof(struct wdmsg));
    }
}
do_write(wmc, wdmsgp)
    intwmc;
    struct wdmsg*   wdmsgp;
{
    do_read(wmc, wdmsgp);
}
```

6. *Applications*

This section describes several of the watchdogs that have been implemented since the system became operational. In general, these watchdogs extend the file system by either providing new functions not previously possible or replacing existing functions.

wdacl

A file access controller that arbitrates over all opens of a file by verifying that the opener is mentioned in an access control list associated with the file being opened. Since each open request is accompanied by the name of the file to be opened, a single watchdog can be used to control access to many files.

wdcompact

An on-the-fly compaction watchdog that allows files to be stored on the disk in compacted form, but viewed normally. Without watchdogs, this could only be done by manually inserting a pipe element to do the (un)compaction between every read and write.

wdbiff

A *biff* watchdog that watches a user's mailbox for new mail and notifies the owner of its arrival. Without watchdogs, *biff*ing can only be done through the combination of several ad-hoc mechanisms.

wdview

A directory watchdog that presents different views of the same directory depending on the user doing the query.

wdhfs

A remote file system watchdog that guards a directory and provides heterogeneous remote access to files. Pathnames mentioning the guarded directory are resolved to the directory and then passed to the watchdog which remotely accesses the file. Information on the location of the remote file is obtained from a file similar to */etc/mtab*. The remote file system is part of the HCS project at the University of Washington.

wddate

A simple *date* watchdog that allows users to read the current time and date from a file. The file itself contains no data; all bytes emanate from a process reading the system clock.

The last example demonstrates that watchdogs can be used to provide a single, very simple, interface to serve many system functions. In this example, the UNIX *date* command has been eliminated.

7. *Performance*

For many users, the question of whether or not to use watchdogs will be decided by their performance. Although individual watchdogs have costs related to their complexity, all guarded operations impose a minimum overhead. This overhead is due to the cost of message passing and context switching incurred on each guarded operation. This section summarizes these basic costs for the most common file system operations: read, write and open.

Table 1 summarizes the read and write costs in terms of the average elapsed time to perform a single operation in the context of accessing a large file. First, we measured the cost of deferred operations where the only overhead is communication to and from the watchdog. From the standpoint of the process accessing the file, a deferred read or write adds about ten percent to the operation's elapsed time. Second, we looked at the time required for a process to read and write a guarded file when the watchdog does the actual reads and writes on behalf of that process. Two sets of figures are given for the reads: one with file read-ahead

	unguarded file	guarded deferred	guarded not deferred	guarded watchdog cache
read 1K (read-ahead)	5.21	5.38	5.58	2.88
read 1K (no read-ahead)	6.85	8.96	9.13	2.88
write 1K	8.09	8.17	8.30	2.92

Table 1: Average Elapsed Time for *read* and *write* (milliseconds)

enabled and the other without. The reason that the elapsed time for guarded operations increases so much without read-ahead is that the watchdog processing for any one buffer can not be overlapped with the read-ahead of the next. Lastly, to mask the overhead of the disk access, we measured the cost of issuing a guarded read (or write) and having the watchdog return the data from an in-core buffer (essentially a cached file). The cost here is significantly less than any call involving the disk, which shows that watchdogs are a viable means for caching frequently accessed files and that the overhead of the watchdog need not be excessive.

Table 2 compares three different types of open: a normal, unguarded open, a guarded open where the watchdog is already running, and a guarded open where the watchdog must be created.

	unguarded file	guarded file (alive watchdog)	guarded file (dead watchdog)
open (absolute name)	3.07	9.59	108.0
open (relative name)	1.40	21.36	117.0

Table 2: Average Elapsed Time to Open a File (milliseconds)

Creating a new watchdog process is costly because it involves a *fork()* and *exec()* by the chief watchdog. Such expense is acceptable for watchdogs that are not frequently used, but cannot be tolerated for those that may be invoked several times every second. The latter may be kept alive even during short periods of inactivity so that they can be referenced quickly. In the best case, opens involving these watchdogs are only three times slower than their unguarded counterparts. The worst case occurs when a file named by a relative path is opened and the kernel must determine the absolute pathname of the file for the watchdog. Since the name of the current working directory is not kept in the kernel, determining the full pathname of the file is expensive.

In situations where a file's contents must always be filtered before their examination (such as in compressed or encrypted files), watchdogs serve as a natural replacement for pipes. To compare the overhead of watchdogs to that of pipes, we conducted two experiments with pipes. The results are summarized in Table 3 (again, averaged over a large number of reads). In the first test, we sent a large amount of data through a pipe to measure the cost of reading one kilobyte from a sending process. This cost is very close to the cost of reading data from a cached file and suggests that watchdogs have data transfer times similar to pipes. To test this hypothesis in a more realistic setting, we had a process read data from a file and write it to a pipe. We measured the time required for each read on the pipe and compared it to the time required to do a non-deferred read from a guarded file. Again we found that watchdogs have speed similar to that of pipes.

	data cached	data from file
read 1K, pipe	2.77	6.05
read 1K, guarded	2.88	5.35

Table 3: Average Elapsed Time of Guarded Reads versus Reads from a Pipe (milliseconds)

Watchdogs' performance can be improved in many ways. Opens can be enhanced by providing a more efficient determination of a file's full pathname. Several parts of the system rely on linear scans of kernel data structures to associate watchdogs with currently open files. These scans could be eliminated by using hash tables. Although the system is designed to allow page remapping between user and kernel space for transferring large blocks of data, this facility is currently unimplemented. Data is simply copied byte for byte between user and kernel. For reads and writes going through watchdogs to actual files, this means that any single byte of data will have to be copied across kernel boundaries three times, twice more than for normal file accesses, and once more than for pipes.

8. Conclusions

Watchdogs allow arbitrary redefinition of file system operations. Because watchdogs are implemented as user-level processes and interact with the kernel, they provide a means to transparently and securely change file system semantics. These qualities allow watchdogs to simplify the user's perspective of the file system by binding a file's contents to its operations. For example, the current date may be read directly from a file, the line printer daemon (*/usr/lib/lpd*) may guard */dev/printer* to arbitrate access to the hardware, and a mailbox may automatically *biff* its owner upon receipt of new mail. We have demonstrated that watchdogs can incur low system overhead and exhibit good performance. Nevertheless, we expect that performance can be further improved as the system matures. We believe that the main concept demonstrated by watchdogs, namely the ability to easily redefine single operating system functions (as opposed to entire subsystems) at the user level, is an important one and should be included in modern operating systems.

Acknowledgements

We'd like to thank Ed Lazowska, Hank Levy, David Notkin and Ellen Ratajak for their helpful comments on earlier drafts of this paper.

References

M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young, Mach: A New Kernel Foundation For UNIX Development, *Proceedings of the 1986 Summer USENIX Technical Conference*, pages 93-111 (June 1986).

F. Baskett, J. Howard, and J. Montague, Task Communication in DEMOS, *Proceedings of the 6th ACM Symposium on Operating System Principles* **14,5**, pages 23-31 (November 1977).

A. P. Black, E. D. Lazowska, H. M. Levy, D. Notkin, J. Sanislo, and J. Zahorjan, Interconnecting Heterogeneous Computer Systems, To appear in *Communications of the ACM,* University of Washington, Dept. of Computer Science (March 1988).

A. Brownbridge, A. Marshall, and A. Randell, The Newcastle Connection – or UNIXes of the World Unite, *Software – Practice and Experience* **12**(2) pages 1147-1162 (December 1982).

W. Joy, E. Cooper, R. Fabry, S. Leffler, K. McKusick, and D. Mosher, 4.2BSD UNIX System Manual, *4.2BSD UNIX Programmer's Manual* (July 1984).

H. M. Levy and R. H. Eckhouse, *Computer Programming and Architecture: The VAX-11,* Digital Press, Bedford, Mass (1980).

R. F. Rashid and G. G. Robertson, Accent: A Communication Oriented Network Operating System, *Proceedings of the 8th ACM Symposium on Operating System Principles*, pages 64-75 (December 1981).

J. Rees, E. Shienbrood, and P. Levine, An Extensible I/O System, *Proceedings of the 1986 Summer USENIX Technical Conference*, pages 114-125 (June 1986).

M. Rochkind, The Source Code Control System, *IEEE Transactions on Software Engineering* **SE-1,4**, (December 1975).

A. S. Tanenbaum and S. J. Mullender, An Overview Of The Amoeba Distributed Operating System, *Operating Systems Review* **15**, pages 51-64 (July 1981).

W. F. Tichy and Z. Ruan, Towards a Distributed File System, *Proceedings of the 1984 Summer USENIX Technical Conference*, pages 87-97 (June 1984).

[*submitted Feb. 22, 1988; accepted March 14, 1988*]