COLA: Customized Overlaying

Eduardo Krell[1] Balachander Krishnamurthy
(ekrell@ulysses.att.com) (bala@research.att.com)

AT&T Bell Laboratories
600 Mountain Avenue
Murray Hill, NJ 07974
USA

System calls are the basic building blocks for writing programs in the UNIX[2] operating system. From the canonical **read, write, open, close, seek, ...** to the more obscure ones, programs have been written to use system calls in a variety of ways. Often there is a need to *intercept* a few system calls to perform some special task. Given that it is hard to go below the level of system calls and still write portable programs, it is easy to see the need for intercepts at the system call level. A simple example of a useful interception facility is a library that watches for file creation and modifications.

In this paper we describe COLA, an elegant, customizable and dynamic facility to overlay a variety of system call intercepts. With COLA, users can specify an arbitrary number of system call filters, each of which may intercept different system calls and perform different actions upon interception. The set of overlaying filters can be modified at any time during the session. A program run under COLA will have any of the filtered system calls processed at each layer before control is passed on to the next layer. The final layer always is the standard UNIX system call layer. System calls not intercepted by any of the overlaying filters will execute transparently. No recompilation of programs or static relinking is necessary. It should be noted that this facility depends on availability of shared libraries.

1. Motivation

The motivation for COLA came from two separate projects in our department. The first project is the 3D File System [KK90] which intercepts some twenty odd system calls to provide the users with a file version facility and a transparent viewpathing mechanism.

The second is Yeast [KR91], an event-action tool, that watches for changes in attributes of objects such as files, directories, etc. The initial implementation of Yeast polled the file system periodically to watch for these events. Polling has the inherent trade-off of a low frequency polling interval leading to missed events, or a high frequency interval leading to a larger overhead. Automatic detection of file system events would eliminate this problem.

[1]Author's present address: Fundacion Chile, Casilla 773, Santiago, Chile. Email: ekrell@fundch.cl

[2]UNIX is a registered trademark of Unix System Laboratories, Inc.

We sought to generalize the system call interception mechanism and implement both the automatic detection facility and the 3D File System under this. As it turned out, we were not only able to do this, but were also able to create other useful applications thanks to the customizability of COLA. By layering these intercept libraries we can dynamically filter the system calls through as many layers as needed for each application.

Related work in this area appears to be sparse: apart from the 3-D File System we were also influenced by Presotto and Ritchie's *streams* mechanism [PR90]. Also, in the Silicon Graphics version of UNIX, there is an inode watching mechanism that is provided via the File Alteration Monitor (*fam*) facility. The facility is available at the kernel level. We believe that trying to achieve specific instances of intercepts by changing the kernel is not practical. A generic mechanism that works not only for file changes, but for any subset of system calls (e.g. non-file system related ones), is preferable. This is precisely the goal of COLA.

In the rest of the paper we look at an example COLA library in detail, discuss the implementation and a brief performance study. We conclude with the lessons learned thus far and some future plans.

2. Examples

In this section we present a sample COLA application in depth and mention other COLA applications.

Often, users are interested in knowing when some operating system phenomena have occurred. Chief among them are changes in the file system; e.g. when a file has been created, changed or removed. Currently, in UNIX, the common recourse is to poll the file system to look for changes in the file's status. With a COLA library, however, it is possible to intercept the **creat, close, open, unlink, write** etc. system calls and make precise deductions as to the change in the status of files.

A brief mention of the event-action specification tool Yeast is necessary. Users register specifications with Yeast consisting of patterns of events and an action; when event patterns are matched by Yeast, the action is triggered. Events can be simple temporal events (`at 9:30am wednesday`) or object events (`file foo mtime changed`), or user-defined events. Yeast has the capability to be notified of occurrences of events via its *announce* mechanism. An announcement packet includes the object, attribute and the new value of the attribute. Thus, when file `foo` is modified, the following announcement is generated and sent

announce file foo mtime = 12:53 November 20 1991

Without COLA, Yeast would have to poll the file system to check for file related events. With the *announce* COLA library, the relevant system calls are intercepted and file related events are announced to Yeast.

The detection itself is a simple matter of recording the **stat** information of a file that has been opened for writing, and checking it against the **stat** information at the time **close** is called. If the **stat** information differ, a Yeast announcement is generated. Similarly, announcements are generated when a file is created or removed (corresponding to **creat** and **unlink**). The announcements themselves are asynchronous writes to a well-known port on which the Yeast daemon is listening. Thus announcements of file changes are immediately handled by the Yeast daemon and any user action that has to be executed upon changes in the file system is triggered immediately, without any polling. The overhead on the file system is negligible and performance measurement shows that the automatic detection model is vastly superior to any polling model.

The *announce* COLA library itself is very simple since it intercepts half a dozen system calls and all but one of the corresponding routines are less than ten lines of code. The longest routine, **open**, is twenty lines long.

A library that permits users to use *emacs* or *vi* style of editing (like *ksh*) while interacting with any command, needs only to trap the **read** system call. Other libraries include a trivial system call tracing facility, a program profiler, a user level file system, a single library that enables moving away from hacks like /dev/fd and /dev/tcp etc. A user recently wrote a automatic file retrieval library simply by trapping *open*; if the file was not locally available a *ftp* connection was opened in background to retrieve the file from the remote site.

3. Implementation

The prototype implementation uses a shared library (which we will call the COLA library) to intercept all system calls. An initialization function in this library uses the **dlopen**() interface to the dynamic link-editor to load into the process' address space the COLA libraries wanted by the user. The user sets an environment variable called LD_COLA to a colon-separated list of libraries of system call traps.

The COLA library looks into each library in this list for definitions of the system calls (**open(), stat(),** etc.). When it finds one, it stores a pointer to it in a jump matrix. The contents of the [**s**,**l**] element in this matrix is a pointer to system call number **s** in the library number **l**. If a library does not contain a definition for a system call, the corresponding pointer is NULL. The current prototype implements system calls only but the COLA concept really applies to all the functions in the C library. In that case, the numbering of the functions would be arbitrary and could be generated on the fly.

In normal operation, when a user-level program issues a system call (like a **open**()), it will be intercepted by the **open**() function in our COLA library. It will then call the first function whose address was stored in the fifth row of the jump matrix described above (the system call number for **open**() happens to be five), after storing the index of the column where that pointer was found. All NULL pointers are skipped.

When the **open**() function in the first library wants to call the **open()** in the next library, it does this by calling _COLA_OPEN(). The COLA library also defines these special names and so this call transfers control back to the COLA library. The job of the COLA library at this point is to call the **open**() function in the next library. Since we stored the index of the column we called last, we start looking for a non-null pointer from the column to the right of the last one searched, and the process continues.

When we run out of columns, it is time to call the actual **open**() system call using the **syscall**() mechanism, and returning the value we get back. System calls that are not trapped will proceed directly since the first entry in the matrix would be NULL requiring no further processing.

3.1. Writing COLA Libraries

A writer of a COLA library needs to follow certain guidelines. If the library implements the **open**() function, then it should have the same interface as the system call (same number and types of arguments, same return values, etc.). Moreover, the writer must make sure that a call is made to the next COLA library layer by calling the _COLA_OPEN() function, else the chain would not be completed.

3.2. COLA Support Library

The COLA support library provides some support functions for the writers of COLA libraries. For instance, the **printf**() function in the standard C library can not be used since it calls the **write**() system call. If a call to **printf**() is made from within a **write**() function in a COLA library, it will cause an infinite loop. We rewrote **printf**() to use the **syscall** primitive instead of calling **write**() directly. As more of these cases are found, they will be added to the support library.

In addition, we plan to provide a safe version of the **signal**() interface to handle interrupts in a transparent way.

4. Performance

Since COLA works by intercepting system calls we decided to measure this overhead. The implementation requires some initialization to be performed before the LD_COLA environment variable is checked. This is done regardless of the number of libraries through which system calls have to be filtered. The goal of the performance measurement was to measure the overhead introduced by the COLA library and to examine the extra time spent in a user's system call trap library. Since the work done inside the user's library varies with the application we considered a skeleton library, *libtrace*, which merely traps the system calls and prints out the name of the call and its arguments. For purposes of the study, the printing was suppressed.

The performance study was conducted on a 21 MIPS SPARCServer 490 with 32 MB memory. A simple C program that made three system calls in a loop was used as the benchmarking program. Three sets of measurements were made. The loop variable was set to be 1000, 10000 and 33333 thus generating 3000, 30000 and 100000 (almost) system calls. Each measurement figure presented is an average of multiple runs.

The first set of measurements did not have LD_PRELOAD set and is the base figure. The second set has the LD_PRELOAD variable set to the COLA library but with the LD_COLA environment variable unset (i.e. no system calls are actually trapped). The third set of measurements represents the figures with both LD_PRELOAD and LD_COLA environment variables set. The LD_COLA variable was set to be `libtrace`, the COLA library which traps all the system calls. Since the benchmark program referred to three system calls (`open, close, unlink`), the overhead involved in going through one layer (viz. `libtrace`) was measured in addition to the initialization overhead of the COLA library.

Table 1 shows the result of the study. As can be seen the overhead introduced is negligible.

Table 1: Overhead of libcola

# of Syscalls	Raw syscalls	LD_PRELOAD set	LD_PRELOAD + libtrace
3000	0.07 + 2.65	0.16 + 3.45	0.21 + 3.80
30000	0.97 + 29.35	1.63 + 32.88	2.14 + 33.26
100000	4.11 + 2m9.65	6.26 + 2m10.5	9.00 + 2m12.75

It must be stressed that if the COLA libraries were to do actual work the overhead would be necessarily greater. We picked the `libtrace` COLA library since it merely printed out the name of the system call along with its arguments. For the measurement period the printing was suppressed, since the goal of the measurement was to study the overhead in setting up and navigating through the layers rather than the work done inside the individual libraries.

5. Conclusion

As a result of COLA, we believe that intercepting system calls in a generic and customizable manner enables several applications that have hitherto not been implemented or implemented with difficulty and high overhead. Our implementation of COLA and the set of libraries we wrote has already demonstrated its usefulness to us. Rather than having a special facility to watch for file changes or making changes in the kernel, we were able to write a small COLA library that performs this task. This is very much in keeping with the UNIX philosophy of writing small tools and being able to combine them into larger ones. We believe that several system call filters will be written, spanning a rich class of applications.

There are some tricky aspects to writing COLA libraries - watching out for recursion, use of **malloc()**, etc. By moving some commonly used and potentially tricky routines into the COLA support library we can help minimize this problem. Users have to be careful about the order of layering the libraries but this is to be expected. The library writers have to adhere to the conventions mentioned in Section 3.1. Our future work will consist of improving the prototype and expanding the user community, apart from writing new COLA libraries.

6. References

[KK90] David G. Korn and Eduardo Krell. 'A New Dimension for the UNIX File System'. *Software Practice and Experience*, 20(S1):19–34, June 1990.

[KR91] Balachander Krishnamurthy and David Rosenblum. An event-action model of computer-supported cooperative work: Design and implementation. In *Proceedings of International Workshop on Computer Supported Co-operative Work, Berlin, FRG.* IFIP, April 1991.

[PR90] David L. Presotto and Dennis M. Ritchie. 'Interprocess Communication in the Ninth Edition UNIX System'. *Software Practice and Experience*, 20(S1):3–17, June 1990.