Trace: A Tool for Logging Operating System Call Transactions*

Diomidis Spinellis
SENA S.A.
Kyprou 27
152 37 Filothei, Greece
e-mail: dspin@leon.nrcps.ariadne-t.gr

April 25, 1994

Abstract

A log of operating system calls made by a process can be used for debugging, profiling, verification and reverse engineering. Such a log can be created by acting as an intermediary between the traced process and the operating system. We describe the design and implementation of such a tool under the MS-DOS operating system environment, and provide some examples of its uses.

1 Introduction

In operating system based computing environments a significant amount of a process's behaviour is defined by its interface with the operating system. This interface typically defines the process's environment such as the current directory, the input/output operations including operations of files, the execution of sub-processes and inter-process communication. Logging and interpreting the transactions between a process and the operating system it runs on, can provide data that can be used for a variety of purposes. Some of them are:

Debugging A listing of a program's operating system calls allows the programmer to analyse its behaviour at a low but well defined level. Program errors can be explained in terms of the operating system calls that were (or were not) issued, and these can in turn point to the source of the error.

Profiling System calls can consume a significant amount of program run time, since the state of the machine must be saved and restored between calls. A listing of the system calls can provide hints on areas of a program that can be optimised (e.g. through buffering techniques) to enhance a program's speed.

Program verification A log of a program's transactions with the operating system can be used to verify a program against its specifications or a run of a previous version. In addition, the log can be used to detect the use of non-portable functions, or programs that have been infected by viruses.

Environment modelling and re-creation Revision control systems such as [Tic82] can use the log of a system's compilation process to determine the exact environment that was used to create the program, including the compilers and tools used and all included files and libraries.

Understanding and Documenting Interfaces The log of a program's operating system transactions can also be used to determine how an unknown program works, discover and understand undocumented system calls, and reverse engineer programs and environments[1]. The list of the operating system functions actually used by a program can be used as a guide for porting the program to an embedded environment lacking an operating system.

* *Operating Systems Review*, 28(4):56-63, October 1994.

[1] For a treatise on the legal implications of reverse engineering in the software domain see [Ign92].

1.1 Related Work

A tool called *trace* with similar functionality to the one described in this article, but hosted on the Unix environment is part of the SunOS [SUN90] operating system tools. In contrast to our approach that tool can not trace child processes. A similar tool is also described in [Rod86].

More mainstream approaches to debugging are tools like *gdb* [Sta89] and *VAX Debug* [Bea83]. Execution profiling under the Unix environment is described in [GKM83], while an MS-DOS profiling tool can be found in [Spi89]. All these tools require access to a program's source code or symbol information to provide meaningful information, while the tool described in this article does not. Methods for tracing operating system call transactions are described in [Bac85, LMKQ88] (*ptrace*) and [Kil84] (the */proc* filesystem).

2 Design

In the following sections we will describe the functionality of *trace*, its user interface (the important command-line options), and the structure of the implemented system.

2.1 Functional Description

The *trace* tool is a single program that intercepts system calls to the MS-DOS operating system and logs them in a readable way in file. These can then be browsed by the user, or processed by more specialised tools. *Trace* can monitor either a command passed as an argument, all the resident processes in the system, or a process with a given program segment prefix (PSP) address[2]. In all cases it creates a file (`trace.log` by default) where each system call and its arguments made by the process(es) monitored are printed. A number of options control the detail of information printed.

Figure 1 contains a sample listing of *trace* output when run on the MS-DOS `xcopy` command. Each line consists of the following fields:

- the system time,
- the PSP address of the calling process,
- the function call number,
- the address of the instruction that generated the function call,
- the function, its arguments and the return value and,
- additional information such as the error information or expanded strings for functions such as `read`, `write` and `getcwd`.

Trace can be used in four different ways:

1. trace a specific command, specified together with its arguments on the command line,
2. trace the system activity in general (this usually means TSRs[3]),
3. trace a given resident program and,
4. trace a sequence of commands.

[2] The PSP address is the MS-DOS equivalent of a process identification number.

[3] Terminate and Stay Resident programs; the MS-DOS equivalent of background running processes.

```
13:10:22 2c40 30 2D3B:178F get_version() = 3.31
13:10:22 2c40 25 2D3B:17B1 set_vector(0x23, 2D3B:0045)
13:10:22 2c40 35 2D3B:17BA get_vector(0x24) = 11C0:0556
13:10:22 2c40 25 2D3B:17D1 set_vector(0x24, 2D3B:0DE5)
13:10:22 2c40 62 2D3B:16A7 get_psp() = 2C40
13:10:22 2c40 19 2D3B:1698 get_current_disk() = C:
13:10:22 2c40 47 2D3B:16A2 getcwd(3, 2C97:01B2) = ok "SRC\TRACE"
13:10:22 2c40 47 2D3B:16A2 getcwd(3, 2C97:01F5) = ok "SRC\TRACE"
13:10:22 2c40 3b 2D3B:130E chdir("trace.c") = Error (Path not found)
13:10:22 2c40 3b 2D3B:1330 chdir(".") = ok
13:10:22 2c40 47 2D3B:16A2 getcwd(3, 2C97:0384) = ok "SRC\TRACE"
13:10:22 2c40 3d 2D3B:152C open("C:trace.c", 0) = 6
13:10:22 2c40 44 2D3B:1535 ioctl(GET_DEV_INFO, 6) =
          FILE: device=2 NOT_WRITTEN REMOVABLE UPDATE_DATE LOCAL
```

Figure 1: Output of `trace -v xcopy trace.c foo`

In order to trace a single command, that command and its arguments are specified on the *trace* command line following any *trace* options. As an example `trace -v xcopy trace.c foo` will trace the `xcopy trace.c foo` command and store the results in the default output file `trace.log`.

When no command is given in the command line, *trace* will run waiting for a keystroke. Until that keystroke is received all system activity is monitored. This allows one to start a TSR by activating its hot-key and monitor its activity.

Sometimes one may wish to monitor a specific program that is already in memory. This usually means that it has been invoked as a TSR. By using a memory inspection utility such as the MS-DOS **mem** command, or `trace -i` one can find the program's PSP address. The PSP address of a program is used as a unique program identifier that can then be passed to *trace* using the `-p` flag in order to specify the program to monitor.

Another use of *trace* can be the monitoring of a sequence of commands, or the functioning of commands internal to a command interpreter. To do that one needs to run the command interpreter under *trace*.

2.2 Command-line Options

The operation of *trace* can be modified by specifying various command line options. Command line options are passed and parsed by *trace* following the standard Unix conventions.

Some of the more interesting command line options that modify the behaviour of *trace* are the following:

-a Monitor all system calls. By default only the documented MS-DOS functions are traced and interpreted. Using the `-a` flag, other functions are logged uninterpreted by listing their function number and the register contents.

`-b` Print the interrupt branch address. Each line is preceded by the address on which the MS-DOS interrupt was generated. The address can be used to pinpoint the routine that issued the function call from a linker listing.

`-c` Only a summary count of all calls is produced at the end of the program run. No detailed information is given. One line is produced for each function used (figure 2). The line contains the function number in hexadecimal, the symbolic function name and the number of times the function was called. This option can be useful for profiling purposes.

```
Function number    Function name    Number of calls
...
3F                   read           497
40                  write             5
41                 delete             2
42                  lseek           320
43                  chmod            10
44                  ioctl            75
45                    dup             2
...
```

Figure 2: Sample output from the execution of `trace -c`

- `-e` Trace between `exec` calls. Unless this option is given, when a program performs an `exec` call, monitoring is disabled until the child process terminates.
- `-f` Calls are prefixed with the MS-DOS function call number.
- `-h` A short help list on the program options is displayed on the standard output.
- `-i` Calls are prefixed with the process-id of the process that performed them. This under MS-DOS is the PSP address of the program.
- `-o` *filename* The output file for tracing information is *filename* instead of the default `trace.log`. The *filename* can also be a device name such as `con` or `prn`.
- `-p` *psp* Trace a process with process-id (PSP address) *psp*. This can be used for tracing a resident program.
- `-r` Produce a register dump on functions that do not have their arguments printed.
- `-s` Functions that take or return string parameters have their parameters printed as strings.
- `-t` Prefix all system calls with the current time in the form of `hh:mm:ss`.
- `-v` Verbose option. This option will produce the greatest amount of data. It is equivalent to specifying the `-abefinrstwx` options. An example of *trace* output produced with this option is given in figure 1. The meaning of each field is explained in section 2.1.
- `-w` Errors from MS-DOS functions are printed in word form (i.e. symbolically) rather than as error codes.
- `-x` Data printed under the -s option will be printed even if it is not ASCII in hexadecimal form.
- `-y` Commit all transaction logs to the file after every call. This is done by closing the file at every call intercept. Although this option decreases performance it can be useful for obtaining a log for programs that never terminate or crash.

2.3 System Structure

Trace works by acting as an intermediary between a process and the operating system as illustrated in figure 3. All system calls are intercepted by *trace*. The input parameters are interpreted and written to the log file. Then, the operating system call is executed by *trace* on behalf of the process that performed the original call. Any results returned by the operating system are also logged into the file, and then passed back to the originating process as if returned by the operating system.

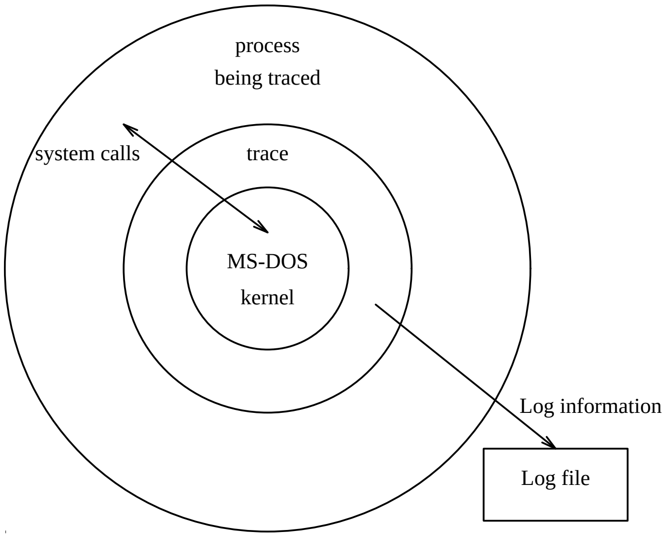

Figure 3: Structure of the system running *trace*.

Functionally *trace* consists of the initialisation part that interprets the command line arguments and arranges the system call intercept, the actual system call interceptor, and a set of functions for interpreting the system call arguments and results.

3 Implementation

Trace is implemented in C with embedded in-line assembly language, using the Microsoft C/C++ compiler version 7.00. The total length of the code is 2422 lines of which 56 lines are in-line Intel iAPX86 assembly language.

Trace works by trapping the MS-DOS function request software interrupt. Whenever the program being traced generates such an interrupt *trace* intercepts it and performs the appropriate logging. In order to intercept the MS-DOS interrupt a custom user-defined interrupt handler is installed. The interrupt handler is written in C. The special (and nonstandard) `interrupt` keyword is used in the C function definition of the interrupt handler to make the compiler generate special code for it. The special handling involves saving and restoring all register values, setting up the data segment register to point to the program's data, and returning from the function using a *return from interrupt* instruction.

The functioning of the interrupt handler routine is relatively simple. First a series of tests are performed to check if the default handler should be invoked directly instead of the user defined function. This should happen in the following cases:

- before the program traced has started executing,
- if MS-DOS is executing a critical section or a critical error handler routine,
- if the process that generated the interrupt is not the process traced and
- if the interrupt handler is being recursively invoked.

Once these tests are passed the handler interprets and logs the input parameters, calls the real interrupt handler, interprets and logs the results, and returns to the calling process. MS-DOS does not support automatic process context switching. At various points the interrupt handler calls the MS-DOS function *setpsp*

```
open("CON", 0) = 10
close(6) = ok
close(7) = ok
close(8) = ok
close(9) = ok
close(10) = ok
close(65535) = Error (Invalid handle)
close(65535) = Error (Invalid handle)
close(65535) = Error (Invalid handle)
close(65535) = Error (Invalid handle)
close(65535) = Error (Invalid handle)
open("R:\WINDOWS\WINSTART.BAT", 0) = Error (File not found)
open("R:\WINDOWS\system\WINSTART.BAT", 0) = Error (File not found)
get_current_disk() = C:
getcwd(3, 3200:0000) = ok    "TMP"
```

Figure 4: Sample output from tracing a widely used windowing package.

to change MS-DOS's idea of the executing process between *trace* and the process being traced. All output to the log file is done under *trace*'s PSP while the invocation of the real MS-DOS function interrupt is performed under the traced program's PSP.

Upon entry in the interrupt handler all the hardware register values can be accessed as part of the C stack frame (i.e. as parameter variables), because the function prologue has saved all registers on the stack. The part written in assembly language is used to transfer all C stack-based variables into registers, issue the real MS-DOS interrupt and transfer back the registers to the C variables. In this way all interpretation of the arguments and the results can be done using the C language constructs.

The code of *trace* that is executing in the interrupt context does not have the same stack frame as the normal C code, since it uses the stack of the traced process. For this reason many of the C library routines that rely on the standard C pointer conventions can not be used (pointers to variables on the stack can not be dereferenced.) The single most important function that could not be used was the *stdio* `printf` formatted printing function. The problem was solved by compiling a modified version of the `printf` function taken from the C library of the 4.3 Net/2 BSD Unix release distributed by the University of California, Berkeley.

4 Experience

During the last three years *trace* has been used as a standard tool for a number of purposes. We have found *trace* to be a useful debugging tool. *Trace* can be applied on an arbitrary executable file making it suitable for debugging programs that have been compiled without symbolic information. Furthermore by redirecting the output of *trace* to the console or printer, the workings of the program until its crash can be easily monitored. The output file that *trace* generates can be used by tools such as *awk* [AKW79] to find calls that return an error, match a particular constraint, or to collect statistics. *Trace* can also be used to debug third party programs pinpointing to an absolute path name that the program is attempting to use, or the missing configuration file that is not reported. Sometimes undocumented features can be found in this way. Figure 4 contains part of a trace log from the initialisation part of widely used windowing package. The reader can see a series of calls to the *close* system function with a wrong argument (65535) and the attempt to open a parameterisation file (`winstart.bat`.)

Using *trace*, one can also discover the way some programs perform interesting functions. Multitasking environments, terminate and stay resident utilities, command processors and DOS extenders were the first programs we tried *trace* on. We found *trace* to be very valuable when developing programs for new environ-

```
get_country()
write(1, 9672:89EA, 8) = 8   "13/10/92"
write(1, 9672:629D, 2) = 2   "  "
get_country()
write(1, 9672:89EA, 6) = 6   "  9:10"
search_next()
write(1, 9672:6270, 2) = 2   "\r\n"
write(1, 9672:89EA, 12) = 12 "TRACE    EXE"
write(1, 9672:89EA, 10) = 10 "     26989"
write(1, 9672:629A, 1) = 1   " "
get_country()
write(1, 9672:89EA, 8) = 8   "03/02/93"
write(1, 9672:629D, 2) = 2   "  "
get_country()
```

Figure 5: Sample output from tracing the MS-DOS directory listing command.

ments and compilers whose debugging capabilities were not as advanced as the the ones we were used to. Often a single program-run under *trace* and an examination of the log file were enough to pinpoint the error source.

Programs can also be made more efficient by studying the output of *trace*. Calls to the operating system are relatively expensive; *trace* provides a way to find the places where they can be grouped together by using techniques such as buffering or caching. Figure 5 contains *trace* log from the MS-DOS directory listing command. Is is obvious that the command's speed could be improved by storing the country information (which is presumably used to determine the data and time output formats) at the start of the program instead of querying it for every date and time value printed, and by using a buffer for output instead of calling the *write* function for small character sequences. A special flag makes *trace* produce a count report of the system calls made by the program. This provides a program profile of overall resource usage.

5 Conclusions

We have described the design and implementation of an MS-DOS system call transaction logging tool. System call transactions can be logged by acting as an intermediary between the traced process and the operating system. Under the MS-DOS system this can be implemented by trapping the MS-DOS function interrupt vector. This approach can be used for debugging, profiling, program verification, environment modelling, and interface documentation.

References

[AKW79] Alfred V. Aho, Brian W. Kernighan, and Peter J. Weinberger. Awk — a pattern scanning and processing language. *Software — Practice and Experience*, 9(4):267–280, 1979.

[Bac85] Maurice J. Bach. *The Design of the UNIX Operating System*, page 376. Prentice Hall, 1985.

[Bea83] Bert Beander. VAX DEBUG: An interactive, symbolic, multilingual debugger. In M.S. Johnson, editor, *Proceedings of the Software Engineering Symposium on High-Level Debugging*, pages 173–179. ACM SIGSOFT/SIGPLAN, March 1983.

[GKM83] Susan L. Graham, Peter B. Kessler, and Marshall K. McKusick. An execution profiler for modular programs. *Software — Practice and Experience*, 13:671–685, 1983.

[Ign92] Gary R. Ignatin. Let the hackers hack: Allowing the reverse engineering of copyrighted computer programs to achieve compatibility. *University of Pennsylvania Law Review*, 140:1999–2050, 1992.

[Kil84] T. S. Killian. Processes as files. In *Proceedings of the USENIX Summer 84 Conference*, pages 203–207. USENIX Association, 1984.

[LMKQ88] Samuel J. Leffler, Marshall Kirk McKusick, Michael J. Karels, and John S. Quarterman. *The Design and Implementation of the 4.3BSD Unix Operating System*, page 104. Addison-Wesley, 1988.

[Rod86] R. Rodriguez. A system call tracer for UNIX. In *USENIX Conference Proceedings*, pages 72–80, Atlanta, GA, USA, Summer 1986. USENIX.

[Spi89] Diomidis Spinellis. v08i002: A C execution profiler for MS-DOS. Posted in the Usenet newsgroup comp.sources.misc, August 1989. Message-ID: <64297@uunet.UU.NET>.

[Sta89] Richard M. Stallman. The GNU source-level debugger. Distributed by the Free Software Foundation, 675 Mass Ave, Cambridge, MA 02139, January 1989.

[SUN90] Sun Microsystems Inc., Mountain View, California. *SunOS Reference Manual*, 1990. Release 4.1.

[Tic82] Walter F. Tichy. Design, implementation, and evaluation of a revision control system,. In *Proceedings of the 6th International Conference on Software Engineering*. IEEE, September 1982.