LA-UR-02-5579

Approved for public release;
distribution is unlimited.

Title: A Dynamic kernel modifier for Linux

Author(s): Ronald G Minnich

Submitted to: LACSI Symosium 2002
Sante Fe, NM

A Dynamic Kernel Modifier for Linux

Ronald G. Minnich

Los Alamos National Laboratory*
Advanced Computing Laboratory
MS-B287
Los Alamos, NM 87545 USA
rminnich@lanl.gov

September 3, 2002

Abstract

Dynamic Kernel Modifier, or DKM, is a kernel module for Linux that allows user-mode programs to modify the execution of functions in the kernel without recompiling or modifying the kernel source in any way. Functions may be traced, either function entry only or function entry and exit; nullified; or replaced with some other function.

For the tracing case, function execution results in the activation of a *watchpoint*. When the watchpoint is activated, the address of the function is logged in a FIFO buffer that is readable by external applications. The watchpoints are time-stamped with the resolution of the processor high resolution timers, which on most modern processors are accurate to a single processor tick.

DKM is very similar to earlier systems such as the SunOS trace device[1] or Linux TT[5]. Unlike these two systems, and other similar systems, DKM requires no kernel modifications.

DKM allows users to do initial probing of the kernel to look for performance problems, or even to resolve potential problems by turning functions off or replacing them. DKM watchpoints are not without cost: it takes about 200 nanoseconds to make a log entry on an 800 Mhz Pentium-III. The overhead numbers are actually competitive with other hardware-based trace systems, although it has less accuracy than an In-Circuit Emulator such as the American Arium. Once the user has zeroed in on a problem, other mechanisms with a higher degree of accuracy can be used.

*Los Alamos National Laboratory is operated by the University of California for the National Nuclear Security Administration of the United States Department of Energy under contract W-7405-ENG-36.

1 Introduction

One of the most frequently asked questions about kernels is "where is all the time going?" We have long wondered, for example, where the time is spent when a program sends a packet. We have found after asking people who should know that no one really knows. Many people have conjectures based on intuition, but in all too many cases the intuition has proven to be wrong. Very simple probing can be done using sufficiently clever external programs, but for the most part, the kernel is a black box which is difficult to measure.

A number of systems have been developed over the years to help resolve these questions. One such system is the SunOS *trace device*[1]. The trace device was present in SunOS at least as early as SunOS 4.1, ca. 1991. It is possible it was present even earlier, but the exact time of creation is hard to fix for reasons discussed below. Another device is the Linux Trace Tool (or *TT*) device. This device provides sophisticated software tracepoint support, albeit at moderate cost. We give an overview of these devices below.

1.1 SunOS/Solaris trace device

The SunOS trace device consisted of several components: a macro that logged data into a FIFO; a device that could give user programs access to the FIFO; and user programs that read the data from the device and turned it into useful information. None of these components were available to us in 1991 when we first encountered this device. As far as we know, the trace device at the time was for Sun internal use only. The only hint of the trace device's existence were the mysterious TRACE macros scattered throughout the kernel source.

We found that Sun programmers had used the TRACE macro in many places in the kernel. The macro was normally disabled, i.e. compiled to no code at all, but could be enabled if desired when the kernel was built. There was no problem in using the macro, since in kernels as shipped it was compiled out. The macro accepted a fixed number of parameters as well as a constant describing the class of trace entry it was creating. In the source as shipped this macro was never enabled or even available; the include and source files for the trace device were not included in the SunOS source tree. We did not even know the values of the constants for trace classes.

Since the SunOS trace device was not available to us, we really have no idea what ultimate functionaly the macro and device provided. The presence of the macros in SunOS source did provide us with a very convenient set of examples and hooks for tracing. We decided to implement our own macro and trace device;we will now describe the trace device we implemented at the Supercomputing Research Center in 1991.

Our trace device had several capabilities. If the trace device was not opened, then no data was recorded in the FIFO. Once the user program had opened the trace device, it could control the size of the FIFO. The user program could also set a mask, so that only events of a certain class were recorded. We defined the trace class constants as bits so facilitate trace enables. The device supported memory-mapped I/O for maximum efficiency: memory-mapped I/O was worth at least a factor of two improvement in performance on SunOS at that time.

The user program had access to a read-only trace buffer as well as read-only control variables. The kernel would always write into the FIFO, and the user program used the counters to determine when new data was in the FIFO, and also how much data it had missed in the event of overrun. We required that the size of the buffer be a power-of-two so that we could use a classic hardware trick: a 32-bit FIFO counter counted the number of FIFO entries, and the low-order N bits of the counter mapped to the slot number of the next entry.

One of our user programs was a TCL program which made reading and dumping the data easy. Once data was saved, we used other programs to analyze the data. Using this information we were able to measure and optimize the performance of Mether[4], MNFS[3], and the NFS file system.

About the same time that we developed our trace device, Brent Welch at XEROX PARC developed his own trace device driver and set of tools. Our systems were very similar. Neither system was ever generally released to the Open Source community. The vibrant Open Source community we take for granted today had barely begun to exist in 1991.

1.2 Linux Trace Tools

Once the research community made the transition to open source operating systems such as FreeBSD and Linux, the capabilities of the SunOS trace device were no longer available. SunOS as shipped contained a substantial number of tracepoints. We did not need to add many of their own. The Open BSD-based and Linux operating systems, in contrast, had no support for tracepoints, and hence no installed tracepoints. It is true that adding tracepoints, a tracepoint device, and user programs is not a difficult task, but trying to keep one's changes in sync with kernels that may be changing on a weekly basis is a burden.

Nevertheless the Linux Trace Tools (LTT) effort has made a first try at implementing a trace device capability. It is an extremely capable device.

The LTT consists of a set of kernel patches, device driver, and user programs. The LTT provides a set of GUIs for viewing events.

The problem with the LTT, from our point of view, is that the patches are very specific to a kernel version. The latest stable LTT, for example, only supports kernel versions 2.2.17 and 2.4.0-test10. The patches are large (4000 lines for 2.4.10); invasive, requiring modifications to 47 source files in 2.4.10; and difficult to turn on and off, since turning them off requires a kernel recompile, as

does adding new tracepoints. For this reason tracepoints are not added lightly.

Finally, the LTT has a bigger time footprint for recording a trace than we are comfortable with. The device itself is complex, and does a great deal of work per trace.

1.3 DKM

We implemented DKM because we need to monitor the Linux kernel at a time when versions are changing almost weekly. The LTT, as mentioned, requires patching the kernel, and has major difficulties with several recent kernel versions. LTT has had a hard time keeping up with the pace of change of Linux. Because we are tracking the latest kernel, it is essential that we be able to instrument the latest kernel.

We also decided that we were no longer willing to recompile the kernel every time we wanted to add a tracepoint. In the SunOS days we often found that we had built a kernel and needed yet another tracepoint. Every trace package with which we are familiar imposes the same requirements: adding traces requires a kernel rebuild and reboot step. We wanted the ability to add and remove tracepoints at will, to any running kernel, without changing the kernel source at all. We also wanted the monitoring system to be easily removed with one command.

Once we had implemented DKM we realized that with just a bit more work we could add two additional valuable capabilities: we could *nullify* a function, and we can also *replace* a function in a running kernel. This type of control has great utility in the protocol stack. While debugging one problem, for example, it would have been very useful to be able to disable delayed acks in TCP. There is a a kernel API for doing this, but it does not work. DKM would have allowed us to experiment with turning the delayed-ack function off or replacing it with one that worked better.

DKM consists of a pseudo-device module and a user-level library which allows the user to insert tracepoints into a live kernel. No kernel patches are needed. Many Linux tools that deal with kernel symbols (e.g. all the module tools) can only use exported symbols as found in `/proc/ksyms`. If a symbol is not exported, the kernel has to be rebuilt. For DKM, the exportable symbols from a kernel need not be changed, since the kernel symbol table (`System.map`) can be used to locate functions. The basic user-level program takes a list of addresses to watch in the kernel and sets tracepoints. We use Perl scripts to convert symbol names in the kernel to addresses.

The basic operation of DKM is to replace the first few instructions of a traced application with a call to a dynamically generated code stub. The instructions replaced by the stub are moved to, and executed in, the stub. We described the actual stub in more detail below. Note that it is essential that the stack frame of the stub be identical to the stack frame of the function.

DKM does not support arbitrary code placement. We target placing tracepoints at function entry. The set of instructions used by GCC at function entry is very limited, and we have exploited this fact: we can not parse the entire x86 instruction set, and hence can not substitute for an arbitrary instruction sequence anywhere in the kernel, only a limited set of instructions (25 at last count) that are typical of those used at the start of a function: load/store from stack; move registers; grow/shrink the stack; and so on.

2 Architecture of DKM

As mentioned above, DKM consists of a pseudo-device module for Linux; a small library for manipulating tracepoints; a set of assembly-code stubs for code modification support; and a set of instruction patterns that the library uses to determine how to build the dynamic code stub for the code modification.

Save for the pseudo-device, all the code works in user-mode for programs, and was in fact developed and debugged in user mode prior to being tested in the kernel. The library interface in both user and kernel mode is the same, although we provide additional interface funtions for user-mode access to the device. DKM can be used for much more than just kernel modifications.

Tracepoint insertion is a specialized instance of the more general case of modifying the function entry. A set of tracepoint support functions and an API are provided as part of the standard DKM source, although users can easily write their own and use them instead.

Because tracepoints are both the most common application of DKM and exercise all the functions of DKM, we will discuss them in detail below. The mechanics of the tracepoint implementation are the most interesting so we

will cover that part first.

3 Inserting tracepoints

Tracepoints are code segments that record execution of a specific function. A given tracepoint only records an individual function. Because they augment the function, and hence increase the number of instructions the function executes, tracepoints must exist in memory allocated outside the function. Tracepoint code is responsible for calling a trace function, executing the first few instructions of the function *in that function's stackframe*,and resuming the function at the correct place. On Pentiums, tracepoint insertion is complicated by the variable-size instruction set of the x86 architecture.

A tracepoint consists of a dynamically allocated code stub that contains a *prologue*, used to call the logging function; the *relocated code* that used to be at the start of the function; and a *jumpback* which jumps back to the rest of the function. The prologue is fixed in size. The relocated code will vary in size, since on a CISC machine like the x86 instructions are variable length. The size of the jump is similarly fixed in size. Overall, however, the code stub will vary in size due to the variation in size of the relocated code.

Inserting a tracepoint requires that we:

- figure out the relocated code size, i.e. how many bytes of instructions we need to relocate from the function to the tracepoint stub (note: on CISC machines this is more complex than, e.g., on Alpha)
- Compute the stub buffer size, given the relocated code size, and the prologue and jumpback code size
- allocate memory for the stub
- copy the prologue code to the stub
- save the relocated code from the start of the function to the stub after the prologue code
- install the jumpback code into the stub after the relocated code
- replace the start of the function with a call to the stub; fix up the relative address in this call so it calls the stub buffer correctly

Figure Omitted: The figure has been omitted since pdflatex can not seem to figure out how to import postscript at present. We will fix this for the final paper if accepted.

Figure 1: How the code stub modifies the function.

- fix up addresses in the stub to point to the correct locations, including call stack return addresses, calls, relative jumps, and the jumpback address

Figure 1 shows how the code looks before and after the tracepoints are inserted. This same technique is used both in kernel and user modes.

Most of the code to handle tracepoint insertion is written in C, in an architecture-independent manner. Some code is necessarily written in assembly; the prologue code, which is (on x86 machines) 2 instructions; the jumpback code, which is 1 instruction; and the call code, which is 1 instruction. C code needs to rewrite pieces of the assembly code with addresses; to support the C code the assembly code exports sizes of assembly code blocks and offsets into the blocks for addresses. The C code has no knowledge of the details of assembly code. The only issue in C is the need to fix up relative addresses for jumps and calls. To support the address fixup, the assembly code also generates named variables that the C code can use to locate fixup addresses.

The C code needs to replace the instructions at the beginning of the traced function with a call to the tracepoint stub. To do this, the code must replace an integral number of instructions with a call instruction. The size of the instructions to be replaced may be larger than the call instruction.

We call this process *sizing*. The issue of sizing is a bit complicated, but we have simplified it greatly by restricting ourselves to code patterns most commonly found at function entry. Once again we use generated assembly code to initialize data structures used by C. The structures describe the code patterns and the size of the code, as well as whether the instruction has a fixed part and an argument (e.g. immediate data).

Tracepoint handlers are called from the tracepoint stub. The call to the tracepoint stub pushes the PC of the traced function on the stack, so this PC is available to the trace-

Figure Omitted: The figure has been omitted since pdflatex can not seem to figure out how to import postscript at present. We will fix this for the final paper if accepted.

Figure 2: Code structure to track both calls and returns

point handler. The handler can examine all the parameters passed to the function, and determine the PC of the function.

Every traced function can have a different tracepoint handler, although to date we have only used one particular tracepoint handler. The tracepoint handler code is written in C and can be arbitrarily complex, with the proviso that if a time-critical function is being traced then the tracepoint handler should do as little as possible. We could in theory have the tracepoint handlers do far more than simply log the function execution: in the limit they could even modify system call behavior. So far, we have limited ourselves (in kernel) to filling the FIFO with trace information.

We had not been using the system long when users demanded the ability to track returns. Code for returns is necessarily more complex because the caller of the function is dynamic. The prologue code must be augmented to save the caller address for later use and to arrange for function return to be routed through code to trace returns.

We show the additional structure in Figure 2.

The first function we implemented in DKM was the tracing function entry, followed by the code to support function exit. Once we had these we realized that nullify functions and replacing a function with a different function would be easy. Since we needed those capabilities we implemented them. The existing library was sufficiently powerful that adding these two new functions only took an hour or so.

4 Nullifying Functions

Nullifying a function is simple. DKM saves the function entry code as before, and replaces it with code to set a return value for the function, followed by an immediate return. The function entry code is not executed.

The API allows the user to specify the return value for the nullified function. We have experimented with replacing the system time function with one that always returns 0. The effects on the kernel (and X11) are revealing.

This capability would have been very useful to us last year when dealing with a bug in delayed ACK handling. We suspected that a particular function in the kernel was at fault, and testing would have involved repeated kernel compiles – too painful to deal with at the time. Had we been able to simply nullify the function in a second or so, we would have been more willing to verify the cause of the problem.

5 Replacing Functions

Replacing a function with another function is the hardest thing to do in DKM. Because the replacement function must become part of the kernel address space, DKM requires that the user compile the function in to the DKM kernel module. This operation is doable but hardly for the faint of heart. Nevertheless it provides an extremely useful debugging capability.

For example, if one is trying to track certain behaviour in a particular kernel system call, one can easily replace that kill with code that provides the extra information. This operation normally requires rebuilding the kernel. What is nice about DKM is that the replacement is trivial to undo, as opposed to direct changes to the kernel that can only be undone by another kernel rebuild. Our experience has shown that DKM can shorten the edit, compile, test iteration for kernel changes from the current 10 minutes to a few seconds.

Once a given change has been tested and shown satisfactory it can be compiled into the kernel for good.

We had a recent experience where this capability would have been very useful. It so happens that the Linux code to allocate a new Process IDs (PIDs) had a simple bug that failed on SMP kernels under high load. Testing of a replacement function took some time, and each iteration required a kernel build and reboot. Had DKM been available we could have experimented with fixing the function with much less time lost waiting for kernel compiles and reboots.

Another useful test would be to see the effect on Linux of PIDs of more than 15 bits, i.e. in a range of 1-32767 (0 is an invalid PID). Since the container for 15-bit pids is

a 16-bit `short`, we can make a simple test by replacing the new PID function with one that returns pids in the range 32768-65536. This is a trivial test but could quickly point out any problems. Since backing out replacement functions is a single `ioctl` call, any problems caused by our alternate new PID function can be quickly eliminated.

DKM function replacement allows testing of alternative low-level kernel functions without modifying the kernel in any way, and in a way that can be backed at the cost of a kernel system call. In the worst case, a user can remove the DKM module and ensure that any and all DKM modifications are totally removed.

6 Library Interface

DKM provides two library interfaces. The first allows a program (or the kernel) to modify its own code space, and manages the set of modifications with reference counts. This code is used by either the kernel or an application to modify itself. The second provides an interface to the kernel device. This latter code is used by programs to modify the kernel.

6.1 Common DKM functions

We show the common library functions in 6.1.

When these functions are run in user mode the tracepoint functions must write-enable the code page containing the function. In kernel mode write-enabling code segments is not an issue.

6.2 DKM Kernel Interface Library

The kernel library interface provided for the DKM device allows user program to set and clear tracepoints; query device status; invoke test functions; and read the tracepoint buffer.

An application which uses this library will open the device, intialize it, set tracepoints on some functions, and periodically dump the FIFO to a file. For timing purposes, the application can set its own "marks" in the trace log kernel via the `dkmtestentry` function, which will place an entry in the FIFO.

name	*Function*
void libtpinit(void);	Initialize the functions
TRACEPOINT *addnulify(void *func, unsigned long retval);	Nullify function `func`. Set the return value to retval.
TRACEPOINT *addreplace(void *func, void *callfunc);	Replace `func` with `callfunc`.
TRACEPOINT *tracepoint(void *func, void *bfunc);	Add a tracepoint to a function `func`, with `bfunc` as the tracepoint handler.
TRACEPOINT *rtracepoint(void *func, void *bfunc, void *rfunc);	Add a tracepoint to a function `func`, with `bfunc` as the tracepoint handler for function entry, and `rfunc` as the tracepoint handler for function exit.
int untracepoint(void *func);	Remove a tracepoint for function `func`. Tracepoints actually have reference counts, so a tracepoint is not removed until all users have removed it.
void remove_all_trace(void);	Unconditionally remove all breakpoints, ignoring reference counts. The main use of this function is when unloading the device, if ill-behaved programs have left breakpoints hanging around.

Table 1: common library functions

name	*Function*
int dkminit(int exponent);	Initialize the device. *exponent* defines the size of the buffer, as a power of two, with an offset of 12 (i.e. starting at a minimum of one page)
int dkmget-info(struct dkm_info *info);	Get information from the device about the size of the buffer, current tracepoint count, and the last tracepoint read from the FIFO
int dkmtes-tentry(struct trace_entry *entry);	For testing, insert a tracepoint record into the tracepoint FIFO
int dkmtrace-point(struct dkmtrace *tracep);	Set a tracepoint. The dkmtrace structure includes the address of the tracepoint and any options.
int dkmrtra-cepoint(struct dkmtrace *tracep);	Set a tracepoint. The dkmtrace structure includes the address of the tracepoint and any options. This variation will trace both calls to the functions and returns from the function.
int dkmsetre-place(struct dkmreplace *tracep);	set a replace point. The parameters are set in the `dkmreplace` struct.

Table 2: tracepoint library functions

name	*Function*
int dkmset-nullify(struct dkmnullify *tracep);	Set a nullify. The parameters are set by the dkmnullify struct.
int dkmuntra-cepoint(struct dkmtrace *tracep);	Clear the tracepoint. The tracep struct has a virtual address which is used to locate the tracepoint. This function is in fact used to remove all types of tracepoints, nullifies, and replaces; in reality this parameter should be a union.
int dkmread-trace(struct trace_entry *entries, struct dkm_info *info);	Read the DKM trace buffer
int dkm-dumptrace(FILE *f, struct trace_entry *entries, size_t size);	dump the trace_entry area *trace* for *size* bytes to FILE *f*
int dkmtestset-trace(void);	Set a tracepoint to a test function
int dkmtest-trace(void);	invoke the test function. If the test tracepoint has been set then the invocation of the function will result in a trace entry being written to the FIFO
int dkmtestclr-trace(void);	Clear the test tracepoint
int dkmread-dump(FILE *f, struct trace_entry *entries, struct dkm_info *info);	Read all tracepoint entries from the device and dump them to the FILE *f*.
int dkm-stamp(int val)	Put an entry in the trace FIFO with a PC value of `val`. This function is used to test the DKM kernel to user interface.

Table 3: tracepoint library functions (cont.)

7 Performance

The performance measure of DKM falls into two main areas: impact of DKM on the kernel, and the precision and accuracy with which DKM measurements can be made. In the next section we discuss the overhead of DKM.

7.1 Performance

The first test is to see how much time it takes to insert time stamps into the kernel. We measure this time by a sequence of back-to-back `dkmstamp` operations, and measuring the time between each. We iterate until the value converges to within a tick or so.

Measured performance on an 865 Mhz Pentium-III is 625 TSC ticks. A tick on this sytem is about 1.15 nanoseconds; thus, cost to add a trace entry from user mode is about 720 nanoseconds on this CPU.

The next number we measure is pure tracepoint cost. To measure this value we enabled the test checkpoint via `dkmsettesttrace` and call `dkmtesttrace` many times. The measured performance on average is 170 TSC ticks, which comes out to 200 nanoseconds.

The 200 nanosecond number is remarkably consistent across a number of measurement cycles. If we set a watchpoint on `open_msr`, and run a program which opens `/dev/cpu/0/msr` one million or ten million times, the additional cost averages to 200 nanoseconds per call.

The cost of tracing returns as well as function entry is much higher – it triples the cost to 600 ns. Users need to carefully consider whether they need call/return tracing or just call tracing, rather than just blindly enabling call/return tracing.

A comparison with alternative hardware-based techniques is in order. Users wishing to trace program execution can modify functions to output trace information to a parallel port, as in [2]; use an In-Circuit Emulator (ICE) to trace functions; or output trace information to a PCI bus and use a PCI bus analyzer to store tracepoint information.

Programs that output trace data would need to output at minimum 9 bytes of data to the parallel port: 8 bytes of TSC, and one byte of function identifier. Output byte instructions take about one microsecond on most systems, so this option would require 8 microseconds to execute: 40 times slower than our current performance. The parallel port approach was fast in 1989, but is too slow now.

We have used In-Circuit Emulators for function timing. They can provide very precise timing, but can be difficult to use. The system must be partly disassembled and moved to a workbench to use an ICE. There are systems on which it is impossible to use them at all (laptops, embedded boards, or non-Pentium systems). Finally, the ICE can greatly impact the performance of the system, typically requiring that the system run at half-speed.

A final option is to place a PCI trace card in the system. Tracepoint support functions can output a trace value to a well-known address. Measured performance of PCI bus write cycles shows that this option would be no faster than the software-based technique outlined in this paper.

In summary, the cost of DKM tracepoints is about 200 nanoseconds for tracing function entry, and 600 nanoseconds for tracing function entry and exit. DKM tracepoints are at least as fast as other hardware-based options.

8 Example usage

In this section we present a sample program that uses the kernel device. The program is called `watchmany` and can be used to watch arbitrary functions in the kernel. The start of the source to the program is shown in 3. The important call is to `dkminit`. The value of '1' indicates a 4096-byte buffer.

The middle section of `watchmany` is shown in Figure 4. The first step is to get information out of the kernel about the device. This information is used in later calls to the library.

The `dkmdumpstatus` call provides an informative message about the state of the device. The user must next allocate an array of trace entries large enough to accomodate the entire kernel trace entry buffer. Having a large enough array is an absolute requirement imposed by the kernel device. If the passed-in array is not large enough, the kernel module returns EINVAL to the user program.

The next step is to scan all the arguments and set watchpoints for them.

The program waits for the user to hit return, removes all tracepoints, and then dumps all the trace entries accumulated by the kernel.

```
#include "dkm.h"

int
main(int argc, char *argv[])
{
  struct dkm_info info;
  struct dkmtrace *tracep;
  struct trace_entry *entries;
  char *type = argv[1];
  int amount, i;
  int dumptrace = 0;

  if (dkminit(1) < 0) {
    perror("kbpinit()");
    exit(1);
  }
```

Figure 3: setup code for `watchmany`

The output of the program is shown in Figure 6. The format of the entries is `calling PC/Arg High-32-bit/Low-32-bits`. The exit entries can be distinguished by the fact that the value is not a valid kernel Program Counter value.

9 Conclusions

DKM is a system for modifying functions used in the kernel. While the original use of DKM was inserting tracepoints into a kernel, it has expanded to provide function nullification and replacement. In our work on kernels we have frequently had need of both these capabilities.

DKM is a loadable module and requires no patches or changes to the kernel; users do not need to recompile the kernel to use DKM capabilities. DKM is almost unique in this regard. DKM is also simple to use, and does not require even a limited understanding by users of the instruction set of the machine it is used on, in contrast to, e.g., the Dyninst tools. Any programmer familiar with the C language can use DKM with ease.

Performance of DKM is comparable to the best performance of hardware-based monitors. DKM tracepoints require 160 cycles, and as a result DKM is suitable for functions that take 1600 or more instruction cycles. An example is an open system call, which we have measured

```
  dkmgetinfo(&info);
  dkmdumpstatus(stdout);

  entries = malloc(info.total_size);

  argc -= 2;
  argv += 2;

  tracep = malloc(argc * sizeof(*tracep));

  if (! tracep) {
        perror("tracep");
        exit(1);
  }

  for(i = 0; i < argc; i++) {
    sscanf(argv[i], "0x%x",
        (unsigned long * )&tracep[i].v);
    tracep[i].options = 0;
    if (*type == 'r') {
      dkmrtracepoint(&tracep[i]);
     else
      dkmtracepoint(&tracep[i]);

  }

  // wait for user to hit LF ...
  getchar();

}
```

Figure 4: Setting up the tracepoints

```
  printf("Removing tracepoint\n");
  for(i = 0; i < argc; i++)
    if (dkmuntracepoint(&tracep[i]) < 0)
      perror("untrace one");

  dkmreaddump(stdout, entries, &info);

  return 0;
}
```

Figure 5: Finishing up: dumping the buffer and removing the tracepoints

```
# ./watchmany r 0xc01c7070 0xc01c5fb0
next_empty 1 last_read 0 \
  total_entry 4096 total_size 65536


Removing tracepoint
next_empty 283 last_read 0 \
  total_entry 4096 total_size 65536
282 entries
0: 0xc01c5fb0/0x35d 0x1310:0xd53bd3c3
1: 0x4c/0x1035d 0x1310:0xd53bdaf7
2: 0xc01c5fb0/0x35d 0x1310:0xd53bfb39
3: 0x14/0x1035d 0x1310:0xd53c000e
4: 0xc01c5fb0/0x333 0x1310:0xd541fa18
5: 0x20/0x10333 0x1310:0xd542049d

.
.
.
.
```

Figure 6: Output of the program for `ping localhost`

at several thousand ticks depending on which device or file system is opened. `Gettimeofday`, in contrast, is a much shorter call and tracing it perturbs it significantly. We have experimented with nullification of `gettimeofday` with interesting results.

References

[1] Sun Microsystems Corporation. *SunOS Reference Manual, 1990.* Sun Microsystems, 1990.

[2] Not Yet Known. Paper on parallel port monitor. In *Late 1980s Usenix Conference*, 1989.

[3] Ronald G. Minnich. Mether-NFS: A modified NFS which supports virtual shared memory. In *Proc. of the Usenix Symp. on Experiences with Distributed and Multiprocessor Systems*, pages 89–108, San Diego, CA (USA), 1993.

[4] Ronald G. Minnich and David J. Farber. The Mether system: Distributed shared memory for SunOS 4.0. In *Proceedings of the 1989 Summer Usenix Technical Conference*, pages 51–60, Summer 1989.

[5] Karen Yaghmour and Michael R. Dagenais. Measuring and characterizing system behavior using kernel-level event logging. In *Proceedings of the 2000 Annual Usenix Technical Conference*, 2000.