SCS&E Report 9409
April, 1994

Tracing Kernel Activity in SunOS 4.0

David Goodall and Stephen Russell

Communicated by Gernot Heiser

SCHOOL OF COMPUTER SCIENCE AND ENGINEERING
THE UNIVERSITY OF NEW SOUTH WALES

Abstract

This paper describes a software tool for tracing kernel activity within the SunOS 4.0 operating system. The tool is designed as a pseudo-controller with separate pseudo-devices to capture event streams from various parts of the operating system. A high resolution interval timer is attached to the target system's SCSI port in order to provide hardware support for timestamps. Use of the timer also allows measurement of CPU usage by the instrumentation system.

E-mail:
disy@vast.unsw.edu.au

1 Introduction

The instrumentation system presented in this paper forms part of the Mungi single address space operating system project [1,2,3,18] at the School of Computer Science and Engineering at the University of New South Wales. The purpose of this tool is to enable researchers to capture operating system kernel event traces to be used in simulation studies of memory management for Mungi. Instrumentation of the paging and filing systems of SunOS 4.0 is underway. The SunOS network protocols have also been investigated, and a simple example of measurements from the network system can be found in Section 4, below.

Our objective has been to provide a method for inserting trace points relatively easily into any part of the kernel source code, including both the top and bottom halves of device drivers. These trace points generate events which may be grouped together as a stream and sent to a pseudo-device for capture. The pseudo-device appends a timestamp to each event prior to storing it in its buffer.

Pseudo-devices are managed by a single pseudo-controller known to the system as *logdev*. Each pseudo-device has an event filter which allows selective enabling or disabling of events. As an example, all events generated by the file system can be sent to one pseudo-device and it is possible, via ioctl calls, to restrict capture to just file open events and file close events, for instance. Overall, there is considerable flexibility in the filtering of events. This reduces impact on the target system and the amount of data processing required for a given experiment.

The instrumentation system consists of an instrumented kernel running on a Sun 3/50 workstation, connected via Ethernet to a remote server, which is used to analyse the trace data. Currently, the system is kept isolated for security reasons. In the future we hope to be attached to the laboratory network so that workloads can be more realistically characterised.

A suite of simple user level programs is responsible for managing the length of measurement experiments, clearing device buffers on demand, and sending these buffers over the network to the file server for storage and later analysis. A user manual and installation details for the instrumentation system can be found in [17].

The organisation of this paper is as follows. Section 2 deals with related work, and Section 3 with the design and implementation of the instrumentation system, including its level of interference with the target system. Section 4 presents examples of usage, while Section 5 contains the conclusion and ideas for future development.

2 Related Work

Much recent work in the design of instrumentation for computer systems has been carried out on parallel systems, in the broad areas of closely coupled multiprocessor systems, dataflow machines and distributed systems [4,5].

With respect to the design of future operating systems, and the enhancement of existing ones, there is still work to be done in taking measurements from current systems such as UNIX. For example, how do these systems interact with applications, and with the underlying hardware? [6]. In addition, accurate traces are desirable for feeding simulators of new designs such as Mungi.

The design of the tool as a pseudo-controller with a number of devices is a generalisation of a tool described in [7]. That paper described a single pseudo-device for measuring file system activity. The measurement system described here also includes a high resolution timer attached to the Sun 3/50 SCSI port. A timer making use of the data encryption chip socket on Sun 3 and Sun 4 computers is described in [8]. Profiling of a network protocol stack with timestamps provided by an interval timer is reported in [9].

3 Design And Implementation

Figure 1, below, shows an overview of the system. The name of the pseudo-controller is *logdev*. It resides in kernel space on the target Sun 3/50, and manages a variable number of pseudo-devices which do the actual work of capturing events generated by trace points within the kernel. Each pseudo-device, via the controller, has access to a timestamp provided by a high resolution interval timer attached to the target system's SCSI port (see 3.2). The overall time elapsed from the start of an experiment is also made available to pseudo-devices, again via the controller, by the clock interrupt handler, which increments a global clock value on each 10ms clock interrupt.

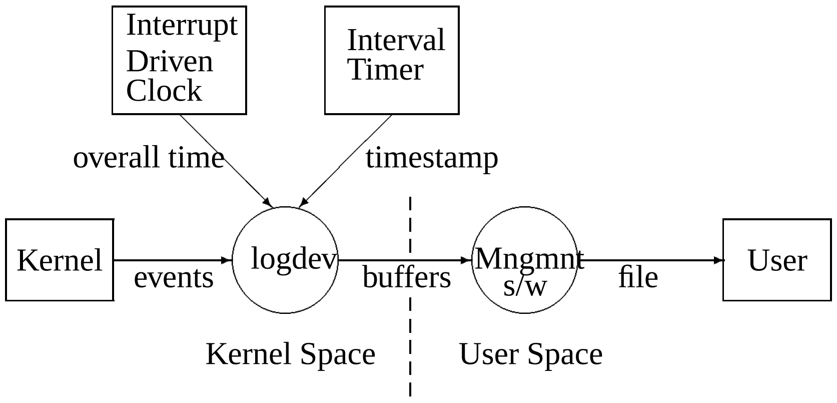

Figure 1, System Overview.

The management software resides in user space, distributed between the target system and the file server, and provides *logdev* with parameters such as which devices and events to enable for a given experiment. As mentioned in the introduction, it is also responsible for controlling the duration of an experiment, and for clearing buffers of events from the target machine to the file server for storage and analysis.

Each part of the system will now be discussed in turn, followed by comments on the level of interference caused to the target system by the instrumentation system.

3.1 Kernel

Probes of the following form are inserted at relevant points in the kernel source code :

logdevwrite(dev, event, length)

where *dev* is the minor device number of the device to which the event is being written, *event* is the event data to be stored and *length* is the size of the event data in units of four bytes. The first byte of the event structure is expected to hold the event identification. A timestamp is appended to each event before it is written to the specified device's buffer.

Currently there are minor devices dedicated to event streams from the file system, paging system and network protocols. Outside of the kernel devices are accessed via system calls referencing the appropriate device special file. These device files share the *logdev* major device number, and the minor device number is used to indicate the subsystem of interest. For example,

fd = open("/dev/filesystem",O_RDONLY,0);

returns a file descriptor which can be used in all user level operations involving the device collecting file system events. Access can thus be limited to an authorised group.

Provision for new devices is made statically at kernel creation time by increasing the size of the device array managed by the *logdev* controller. A device special file is then created for each new device using the UNIX *mknod* system call. Dynamic run-time allocation of buffer space within the kernel has proved difficult and unreliable, so a static approach has been adopted.

3.2 Clocks

As already indicated, the measurement system contains two clocks, the high resolution interval timer, and the regular 10ms interrupt available from the operating system clock interrupt handler.

3.2.1 Interval Timer

This timer is attached to the SCSI port of the workstation and consists simply of an eight MHz crystal module, a 32 bit counter, a simplified SCSI interface, and some logic. The data structures initialised for the SCSI host adapter on the Sun 3/50 are borrowed in order to read and write synchronously to the SCSI bus. The prototype gives a range of resolutions from 125 nanoseconds to 32 microseconds.

When an enabled event occurs, the minor device to which it is written stops the interval timer, reads it, appends the time value to the event and writes the timestamped event to its buffer. The device then restarts the timer to measure the time which elapses until the next event. The time taken to write the event to the buffer is not incorporated in the timestamp.

At the end of an experiment the time line for the events can be reconstructed by incrementing the intervals for each event, less a fixed amount of overhead, K, for each event (see 3.5). This gives a time line which is relatively free of measurement overhead.

This way of using the timer has some limitations :

- Removal of timer overhead from the time line is not suitable for multiple timers operating in a distributed instrumentation system. Such environments require a global concept of time.
- The timer is not suitable for attachment to a file server unless it has a separate, free, SCSI port, so that contention for the SCSI bus can be avoided.
- Only one minor device, i.e. stream of events, can be enabled for the duration of any given experiment, as event streams are stored in separate buffers and it is not possible to reconstruct the original timeline from multiple buffers.

3.2.2 Interrupt Driven Clock

The regular 10ms system clock interrupt is used to increment a monotonically increasing clock value. This value is used to measure the overall duration of an experiment and allow calculation of the CPU usage by the measurement system.

The operating system alarm clock utility is used to set a rough time limit for an experiment, and the increasing clock value is used to give a more exact overall measure, to a resolution of 10ms. In order to do this, a 'time event' is forced into the event stream with the value of this clock, when the system alarm goes off.

3.3 Pseudo-Controller

As stated above, the *logdev* pseudo-controller supports one device for each separate stream of events. The controller supports the following system calls in order to manage these devices : open, read, ioctl, and close. Devices may only be written to from within the kernel.

Each device consists of an event filter, to allow selective enabling of events, an enable flag, and pointers to the storage for events. The storage is singly buffered at this time. This means that a number of events are missed if the buffer becomes full while execution is within non-preemptable kernel code. This is most critical in areas such as networking, where loops are employed to break up the processing of large packets.

One solution to this problem is to double buffer the device, and allow continuous monitoring, even while buffers are cleared via the network. The effects of clearing the buffer must then be removed from the trace in later processing. This can lead to escalating overhead in order to differentiate between the causes of events.

The preferred solution is to write each event out of the target system in synchronous fashion via a convenient parallel port, as in [10]. This method has the desired effect of lowering measurement overhead. The events can be timestamped and collected by a separate dedicated system. This forms the basis of a possible future distributed instrumentation system.

To return to the current situation, device buffers are allocated statically, just after the system tables, at system start up. Buffers are cleared by a user level process executing on the target machine, as discussed in the next section.

3.4 Management Software

The management software consists of three simple processes. A timer process and a reader process execute on the target machine, and a process to accept and store event buffers executes on the file server machine. The following example illustrates the case of using a single pseudo-device. The sockets mentioned below are stream sockets, employed for reasons of reliability while development of the system continues.

Firstly the process on the file server is started. It creates a receive socket, prints the socket number to the screen, and waits for packets of events to be sent from the target machine.

On the target machine, a timer process is started with all experiment parameters, including the number of the receive socket on the file server machine. The timer process forks another process to read from the specified pseudo-device. The timer process then sleeps, waiting for the alarm to go off at the end of the experiment's duration. At this time it will disable the device and clear anything left in the device's buffer.

The reading process enables a specific device, and events for that device, using ioctl calls. It then makes a read call to the device. The semantics of the read call are such that the reading process sleeps until the device signals that its buffer is full. Tracing is disabled at this point. The reading process wakes, completes the read, sends what it has read over the network to the storing process on the file server, re-enables the device, and then reads again. This cycle is repeated until the reader is killed by the timer process at the end of the experiment.

3.5 Interference With The Target System

In common with other designers of such instrumentation tools, we have a goal of minimum interference with the target system. Since the various event streams, and events within these streams, may be enabled or disabled by the experimenter, the amount of interference will vary with each experiment.

The amount of interference acceptable to researchers has been steadily falling with technological advances, particularly with respect to parallel systems. A figure of up to five percent of CPU time has been acceptable for instrumentation tools aimed at non-parallel systems in the past. One percent, and preferably below, is more acceptable for parallel and distributed instrumentation systems today [10,11].

Inclusion of a high resolution interval timer in the target system allows us to measure the amount of CPU time spent in measurement. As already stated, the timer measures time elapsed between events to a resolution of 125 nanoseconds. The time spent in measurement T_m, will be the overall time of the experiment, T, less the sum of all intervals, T_i, plus the number of intervals, n, multiplied by a small figure for unavoidable overhead, K.

$$T_m = T - T_i + (n * K)$$

The value of K has been measured for the instrumentation system and found to be 29.75 microseconds on our Sun 3/50 implementation. The value includes calling a write procedure for each event, and testing whether the specified device and event are enabled.

This form of elapsed time measurement is not applicable to a distributed instrumentation system, since events in such a system may be compounded of events occurring on separate nodes, in a global time sense [12]. In addition, the measure of CPU usage is only a guide to possible interference since we cannot know what the system may have done had it not been engaged in the measurement system code at some particular point in time, i.e. we cannot quantify the cost of ‘lost opportunity’.

CPU usage by the instrumentation system is dependent on the number of events enabled, their size, their frequency of occurrence, and the capture buffer size, as well as factors such as bus contention. As a general guide, an experiment which generates 5000 events of length 8 bytes, over a sixty second period, will require around 3 percent of available CPU time on the target system, if 4096 byte event buffers are cleared over the network when full.

4 Instrumentation Examples

These examples are representative of initial work carried out in order to validate the instrumentation system. Instrumentation of the network protocols has been valuable in demonstrating the limitations of the system. Instrumentation of the file system has allowed capture of traces which produce results comparable with those of earlier studies [16].

The following table shows throughput measurements of the Transport Control Protocol (TCP) for varying packet sizes under almost ideal load conditions. I.e. the number of active processes executing on the systems under test was limited to as few as possible. As mentioned in the introduction, the instrumentation system, i.e. target and server, is isolated, so that workloads are not currently realistic. Events for this example were captured at the Internet Protocol (IP) level.

TCP Packet Size (Bytes)	*Throughput (Bytes/Sec)*
64	320
128	640
256	1281
512	2558
1024	5123
2048	174856
4096	260451
8192	262587
16384	261425
32768	258615

Table 1, TCP throughput in bytes per second.

The testbed was the Sun 3/50, with instrumented kernel, attached via Ethernet to an NFS file server. The network was loaded via one stream socket connection. A process on the instrumented workstation initiated a stream connection with a process on the file server and sent it a continuous stream of bytes. Limited variation in packet size was achieved by changing the size of the send and receive socket buffers. The above figures are calculated from a steady state portion of transmission, leaving out the start up phase.

The capture device's buffer was set at 64Kb in order to allow a reasonable measurement period, but to avoid clearing the buffer over the network. Throughput figures are volatile and difficult to reproduce exactly for the higher rates, presumably due to background operating system activity and hardware contention. The best throughput figure is for a TCP packet size of 8192 bytes, which matches the operating system page size.

The throughput performance is not spectacular. The implementation of the TCP/IP protocols in SunOS 4.0 predates more recent published research into protocol processing overhead [9,13]. In general, the performance of this implementation is affected by operating system memory management and by a layered design, which requires the operating system to process an interrupt for each layer between the hardware level and the user level, upon receipt of packets. Note that the number of interrupts is not the same at each level. For example a 4096 byte TCP packet is received as four Ethernet packets. Thus, for large TCP packets, the number of interrupts declines as the user level is approached [14].

The creation of the trace of events for the 8192 byte packet size required usage of 0.59 percent of CPU time on the target system by the measurement system. Again, note that the event buffers were not cleared over the network in order to avoid interference.

Other measurement experiments have been conducted with the instrumentation system, [15], similar to the file system study reported by Ousterhout et al. in [16]. The figures obtained compare reasonably well with the figures reported in [16], although our workload was somewhat artificial. It was produced by user programs directed at processing and analysing large files of trace events, and by the general editing of small files, all running under SunView on the target Sun 3/50 machine.

5 Conclusions and Future Directions

This paper has presented a software tool for capturing event traces of kernel activity in SunOS 4.0. Within its limitations of overhead and buffering, the tool is usable and shows promise for future development.

Future work for the tool includes studies, with realistic workload, of file system and paging system behaviour pertinent to the design of a single address space operating system. In its current form, the tool is suitable for limited measurement of sub-systems such as the network protocols. In a double-buffered form, it should provide useful continuous traces of the paging system and file system.

Future development of the tool should use knowledge gained from designing the interval timer in order to construct a system that writes events directly out of the SCSI port to an independent system for timestamping, collection and processing. This would form the nucleus of an instrumentation system for a small distributed computing system.

6 Acknowledgements

Thanks are due to Dimitri Spyridopoulos and Eric Tong who assisted with the development of the instrumentation system by using it as part of their undergraduate honours projects. Thanks also to Ric Forster of the CaST laboratory for technical assistance.

7 References

[1] S. Russell, A. Skea, K. Elphinstone, G. Heiser, K. Burston, I. Gorton, and G. Hellestrand. Distribution + Persistence = Global Virtual Memory. In IEEE International Workshop on Object Orientation in Operating Systems, Vol. 2, pages 96-99, Dourdan, France, 1992.

[2] G. Heiser, K. Elphinstone, S. Russell, and G.R. Hellestrand. A Distributed Single Address Space System Supporting Persistence. School of Computer Science and Engineering Report 9302, University of NSW, Kensington, NSW, Australia, 2033, March 1993.

[3] J. Vochteloo, S. Russell, and G. Heiser. Capability-Based Protection in the Mungi Operating System. In IEEE International Workshop on Object Orientation in Operating Systems, Asheville NC, 1993.

[4] M. Simmons, R. Koskela, I. Bucher. Editors. Instrumentation for Future Parallel Computing Systems. Addison Wesley 1989.

[5] M. Simmons, R. Koskela. Editors. Performance Instrumentation and Visualization. Addison Wesley 1990.

[6] T.E. Anderson, H.M. Levy, B.N. Bershad, and E.D. Ladzowska. The interaction of architecture and operating system design. In Symposium on Architectural Support for Programming Languages and Operating Systems, Vol. 4, pages 108-21, 1991.

[7] I. Hu. Measuring File Access Patterns in UNIX. In Performance Evaluation Review, Vol. 14, No. 2, pages 15-20, 1986.

[8] P.B. Danzig and S. Melvin. High Resolution Timing with Low Resolution Clocks and A Microsecond Resolution Timer for Sun Workstations. In Operating Systems Review, Vol. 24, No. 1, pages 23-26, Jan. 90.

[9] D. Banks and M. Prudence. A High-Performance Network Architecture for a PA-RISC Workstation. In IEEE Journal on Selected Areas in Communications, Vol. 11, No. 2, February 1993.

[10] O. Endriss, M. Steinbrunn, and M. Zitterbart. Netmon-II a monitoring tool for distributed and multiprocessor systems. In Performance Evaluation, (North Holland pub.), Vol. 12, No. 3, pages 191-202, June 1991.

[11] J.E. Lumpp, Jr., T.L. Casavant, H.J. Siegel, and D.C. Marinescu. Specification and Identification of Events for Debugging and Performance Monitoring of Distributed Multiprocessor Systems. In IEEE 10th International Conference on Distributed Computing Systems, pages 476-483, 1990.

[12] M. Spezialetti and J.P. Kearns. A General Approach to Recognising Event Occurrences in Distributed Computations. In 8th International Conference on Distributed Computing Systems, pages 300-307, 1988.

[13] D.D. Clark, V. Jacobson, J. Romkey, and H.Salwen. An Analysis of TCP Processing Overhead. In IEEE Communications Magazine, Vol.27, No. 6, pages 23-29, June 1989.

[14] S.J. Leffler, M.K. McKusick, M.J. Karels and J.S. Quarterman. The Design and Implementation of the 4.3BSD UNIX Operating System. Addison Wesley, 1989.

[15] D. Spyridopoulos. Tracing the Sun Operating System to help resolve file system and GVM issues. Honours thesis, School of Computer Science and Engineering, University of NSW, Kensington, NSW, Australia, 2033, 1993.

[16] J.K Ousterhout, H. Da Costa, D. Harrison, J.A. Kunze, M. Kupfer and J.G. Thompson. A Trace-Driven Analysis of the UNIX 4.2 BSD File System. In 10th Symposium on Operating Systems Principles, pages 15-24, 1985.

[17] D.S. Goodall. Operating System Instrumentation. Masters Project Report, School of Computer Science and Engineering, University of NSW, Kensington, NSW, Australia, 2033, 1993.

[18] G. Heiser, K. Elphinstone, S. Russell, J. Vochteloo. Mungi: A Distributed Single Address-Space Operating System. In *Seventeenth Annual Computer Science Conference* volume 16, number 1, Christchurch, New Zealand, January 1994. Australian Computer Science Communications.