An Evaluation of the Real-Time Performances of SVR4.0 and SVR4.2

Sherali Zeadally
Department of Electrical Engineering
University of Southern California
University Park, DRB 116
Los Angeles, California 90089
zeadally@marco.usc.edu

Abstract

UNIX is one of the most widely used operating systems on current workstations. However, UNIX was originally designed as a multitasking and time-sharing system with little concern for supporting real-time applications. Recent versions of UNIX have incorporated real-time features and the designers of these systems claim to provide better response times than the standard UNIX kernel. In order to assess the benefits of these new features and verify these claims, this paper compares the real-time performances of two popular versions of UNIX namely System V Release 4.0 and System V Release 4.2 for the Intel platform.

1 Introduction

Various features have been included in the basic UNIX [8] kernel by various manufacturers in their attempt to develop their own real-time version of UNIX [2][3]. Most of the features that have been added required changes to the UNIX kernel including all or a subset of the following: fixed priority process scheduling, fast process synchronisation, fast file system, real-time timers, asynchronous I/O, resident program support, resource preallocation and others. All these new features give maximum preference to a real-time process which is ready to execute. However, there is another major factor that real-time versions of UNIX have attempted to overcome: decreasing the amount of time it takes by the operating system kernel to start executing a process that is supposed to respond to some external event. That is, the aim is to provide deterministic response times to respond to events fast enough in order satisfy some real-time requirement.

The main reason for choosing two UNIX System V versions for comparing real-time performances of UNIX-like operating systems is because they have both evolved from the same standard UNIX kernel. This allows better comparisons to be made in terms of the benefits of the improvements made as opposed to other real-time versions of UNIX which are often hybrids of several other operating systems (e.g. FlexOS [6] includes functions from UNIX, DOS, and VMS operating systems).

UNIX System V Release 4.0 (SVR4.0) was the first version that came out as a result of the rework made to the basic kernel to add real-time enhancements. The two major modifications were: the addition of a preemptible static priority scheduler to the existing round-robin scheduler and the insertion of *preemption points* [5]. As the basic kernel is not preemptive, it can only be split into processing steps that must run to completion without interruption. In between the processing steps, "safe"[1] places known as *preemption points* have been identified where the kernel can safely

[1] A safe place is a region of code where all kernel data structures are either updated and consistent or locked via some semaphore.

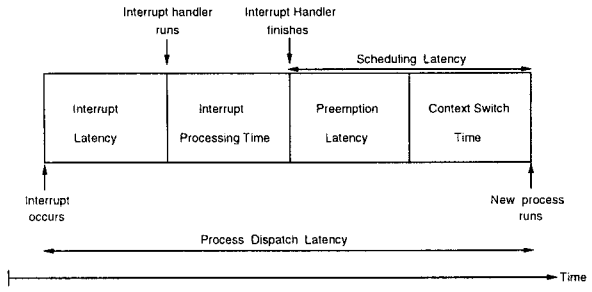

Figure 1: Process Dispatch Latency

interrupt its processing and schedule a new process.

SVR4.2, the successor to SVR4.0, was released in June 1992. The designers of SVR4.2 claim that it provides better real-time performance than SVR4.0 because the 4.2 kernel has been made fully preemptible and re-entrant [6]. In addition, SVR4.2 also supports multiprocessing.

2 Description of Terminologies

Important parameters commonly used in benchmarks when assessing the real-time performance of operating system kernels include: interrupt latency, context switch time, kernel preemption latency, and process dispatch latency. A pictorial description of these parameters is given in Figure 1 as used in the context of this paper. The various parameters are described in detail below.

- Interrupt latency: this is the time between the generation of an interrupt signal and the execution of the first instruction of an appropriate interrupt handler.

- Interrupt processing time: this is the time for which the interrupt handler runs. The kernel blocks all interrupts at this and all lower *interrupt priority levels* during this period. The timely response of the system depends on this interval being as short as possible; that is, interrupt handlers should be efficient and perform minimum processing necessary in order to allow the kernel to re-enable interrupts as quickly as possible.

- Kernel preemption latency: traditional UNIX systems can preempt a process running in user mode immediately. If, however, the current process is executing in kernel mode then it cannot be preempted (except by external interrupts) but can only voluntarily and explicitly give up the processor. Specifically, a process executing within the kernel relinquishes the processor only when it calls the *sleep()* routine to suspend itself until a needed resource becomes available, or after the completion of a system call when it is about to return to user mode thereby allowing preemption to occur. Thus, the kernel can execute for a significant amount of time before giving up the processor to another process. This period of time is called the *preemption latency*. A user process executing a system call in kernel mode can be preempted by an external interrupt. In this case, interrupt handler processes the interrupt and then gives control back to the processor at the end of interrupt processing. In the case of a standard UNIX kernel, execution of the system call continues until it finishes. This implies

that even if a higher priority process has been made available as a result of the interrupt, it cannot run and has to wait until the system call completes. However, new versions of UNIX including System V Release 4 attempt to reduce preemption latency by inserting preemption points in the kernel. With these preemption points, it is possible to switch to another higher priority process immediately (i.e. at the next preemption point) without having to wait until the end of system call execution. Thus, the preemption latency in systems with preemption points is bounded by the *maximum* interval between any two preemption points.

- Context switch time: this includes the time spent in saving the *context*[2] of a currently running process process, in locating another process ready to run, and in restoring the context of that process. When the kernel preempts a process its context is saved. When the kernel schedules it to run, its context is restored and it continues to execute again.
- Scheduling latency: this is the preemption latency plus the context switch time.
- Process dispatch latency: this is the total time from the occurrence of an external interrupt to the beginning of execution of the process assigned to respond to that interrupt.

3 Overview of Measurements

The objective is to evaluate the real-time performances of SVR4.2 and SVR4.0 principally on how quickly each can preempt the kernel and dispatch a new process in order to respond to external events. Thus, performance evaluations will focus on the various components of process dispatch latency and include: interrupt latency, context switch time, and preemption latency. Measurements of interrupt processing time have not been taken since they are likely to vary with the implementation of interrupt handlers. Particular attention is also paid to *worst-case* figures which are much more important in real-time applications than *typical-case* figures. DELL UNIX (SVR4.0) and Consensys UNIX (SVR4.2) were chosen because of their reputation as well performing ports of the System V Release 4 product to the Intel 80x86 architecture.

All timing measurements have been made using a microsecond transputer (T801) clock. A network adapter with one megabyte of on-board static Random Access Memory (RAM) has been used. The network adapter connects to the workstation EISA bus and its on-board shared memory is accessible by both the host and the transputer. The clock is made available to both the transputer and the UNIX host by using a common shared memory location on the network adapter. This location is continually updated by the transputer one microsecond clock. A transputer program is used to generate packets at certain user-defined intervals. All the experiments described in this paper use an inter-packet transmission interval of 10 milliseconds and each test run was over a period of 10 seconds. A simple device driver ("tatm") has been implemented to take timing measurements on the UNIX host system.

[2]The use of the word *context* here implies a snapshot of the process runtime environment. It consists of processor registers in use (e.g. program counter, stack pointer), associated kernel data structures such as *user* and *proc* in the case of a UNIX process, and other operating system variables that are used to manipulate the process at any given time.

In all tests performed, a user-level process (henceforth referred as the *test* process) is put to sleep in the `tatm` driver by making a blocking read system call. The interrupt handler of the driver simply does a wakeup of the sleeping process and returns. Two sets of experiments have been undertaken on both SVR4.0 and SVR4.2. One set of experiments runs the test process in the "time-sharing" class and the other runs the test process in the "real-time" class. Three scheduling classes (time-sharing, system, and real-time) are supported in UNIX System V Release 4 [5]. Both system class processes and real-time class processes use a fixed priority scheduling policy with real-time processes being assigned higher priorities than system processes. The "system" scheduling class is intended for standard system daemon processes such as the page stealer. Time-sharing processes have the lowest priority levels. In addition, only real-time processes can make use of preemption points to preempt the kernel.

The following procedures have been used to measure the following parameters:

- Interrupt latency: the transputer program copies the value of the clock into an array holding timestamps in main (transputer) memory at a granularity of approximately four microseconds at the time when each interrupt is generated. An interrupt is generated by means of the transputer program writing to a special memory location. The interrupt handler of the UNIX host records the times (using the transputer clock via shared memory) at which it begins execution for each interrupt in an array in local memory. At the end of a test run, interrupt latency times are calculated by subtracting corresponding timestamps. The values obtained give the interrupt latencies for the interrupts generated.

- Context switch time: in addition to the test process sleeping in the kernel, another user-level "soak" program comprising a tight loop is run in user mode in the time-sharing class. The "soak" program ensures that the processor is never idle and spends no time in the kernel. The point of having a process always runnable is to be able to measure a *full* context switch. Thus, when an external interrupt occurs, a hardware trap transfers control from the test process to the interrupt handler of the `tatm` driver to service the interrupt. The interrupt processing performed simply wakes up the sleeping test process. The test process is bound to be the next one to run before any other user-level processes since in UNIX the priority level of a sleeping process is always raised to be above that of compute-bound processes; this is to improve the responsiveness of the system to external events like key-presses and mouse-clicks. As a result, a full context switch to the test process *immediately* takes place. The last instruction in the interrupt handler reads the value of the transputer clock and it is read again as the first instruction when the awoken process begins execution. The time difference gives the context switch time. For each test run, timestamps are recorded in two arrays by the *tatm* driver in local memory. Context switch times are calculated by subtracting corresponding timestamps.

  A possible weakness in the context switch measurement test is that it is difficult to measure *strictly* context switch time for time-sharing processes. This is because the experiment assumes that the awoken process is definitely going to be the one to run before any other processes. While this is certainly true for a real-time process, it may not necessarily be the case for a time-sharing process because a system process with a higher priority can execute before it. For instance, it might be prevented from executing by higher priority processes such as system processes responsible for OS activities (e.g. as locking pages in memory, swapping processes out, manipulating timer queues). Thus, in this particular case, the measurement

results give scheduling latencies rather than *pure* context switch times. Ideally, the best way to measure context switch time for all processes is to take timing measurements at different points in the kernel code. However, it was not possible to adopt this method because UNIX source code was not available.

- Scheduling latency: scheduling latency was measured in the same way (i.e. at the same places in the `tatm` driver) as context switch time except that an additional "find" program was run. The "find" program was made to do a recursive search of a Networked File System (NFS) over the Ethernet network. During the time the find program runs, it spends a considerable proportion (over 95%) of its execution time in the kernel performing system calls. However, the 'find" program spends approximately 80% of the time sleeping waiting for I/O. The rationale for choosing such a program was that at least part of the sample measurements taken will reveal the effect of system calls on scheduling latencies. Other operations such as process creation (fork) or process overlay (exec) have been experimented with but could not be used because they execute for such a short period of time that prevents measurements to be taken over a long interval. Furthermore, the "find" program causes interrupts to be generated by the Ethernet adapter. The interrupt priority level of the Ethernet driver was the same as that of the `tatm` driver.

4 Results

The experimental results for context switch time and interrupt latency on SVR4.2 and SVR4.0 are summarised in Table 1 when the test process runs in real-time class.

There are several observations that can be made based on Table 1:

1. The average interrupt latency on the UNIX versions used is in the range 26-28 microseconds.
2. The interrupt latency of SVR4.2 has *not* improved over SVR4.0. In fact, interrupt latency has become slightly worse. This confirms some of the views that, owing to neglect, performance of this parameter is not getting any better [7].
3. The average context switch time is in the range 77-86 microseconds. There has been a slight improvement of about 10.5% of SVR4.2 over SVR4.0.
4. *Worst* case context switch time is 3 times better (i.e. less) for SVR4.2 than SVR4.0 which should provide better response time.

   A possible explanation for the improvement in context switch time for SVR4.2 is because the compiler used with SVR4.2 gives better optimisations than the one used with SVR4.0.

For completeness, SVR4.0 and SVR4.2 have also been compared when running the test process in the time-sharing class. Results are presented in Table 2. Two notable points worth making are:

1. The results for the average values of interrupt latency are nearly the same as corresponding ones in Table 1. However, maximum interrupt latency when the test process is run in the time-sharing class is almost two times worse than when it runs in the real-time class on either operating system. This result implies that the worst case time for which interrupts are locked out has improved in both SVR4.2 and SVR4.0 for a real-time process.

	Minimum	Median	Maximum	Mean	Standard Deviation
Interrupt latency (SVR4.0) (microseconds)	23	27	100	26	7.65
Interrupt latency (SVR4.2) (microseconds)	25	29	95	28	17.47
Context switch time (SVR4.0) (microseconds)	80	84	8500	86	158.95
Context switch time (SVR4.2) (microseconds)	72	76	2800	77	51.2

Table 1: Test Process in Real-Time Class

	Minimum	Median	Maximum	Mean	Standard Deviation
Interrupt latency (SVR4.0) (microseconds)	24	27	275	27	7.46
Interrupt latency (SVR4.2) (microseconds)	25	29	245	29	8.48
Scheduling latency (SVR4.0) (microseconds)	80	80	9600	89	191.23
Scheduling latency (SVR4.2) (microseconds)	68	72	9600	92	326.93

Table 2: Test Process in Time-Sharing Class

2. The scheduling latency values for both operating systems are similar. This might indicate that the code path in SVR4.0 is already optimised and there is not much scope for further improvement in SVR4.2.

Scheduling latency results for both SVR4.0 and SVR4.2 when using the test process in the real-time class are given in Table 3. The results show that average scheduling latency for SVR4.2 is better than SVR4.0 by about 18.8%. The other interesting result from Table 3 is the significant decrease of about 41% in the worst case scheduling latency of SVR4.2 over SVR4.0. Both these improvements mean that there is an amelioration in kernel preemption latency and determinism for SVR4.2.

A graphical representation is also presented in Figure 2. The presentation format used has been devised by Faller [4]. The ordinate shows the times for scheduling latency. The abscissa shows the percentage cummulative frequency of these times. The value on the ordinate corresponding to P%

	Minimum	Median	Maximum	Mean	Standard Deviation
Scheduling latency (SVR4.2) (microseconds)	72	76	3272	121	184.45
Scheduling latency (SVR4.0) (microseconds)	80	80	5544	149	257.91

Table 3: Scheduling Latency of SVR4.0 (real-time) versus SVR4.2 (real-time)

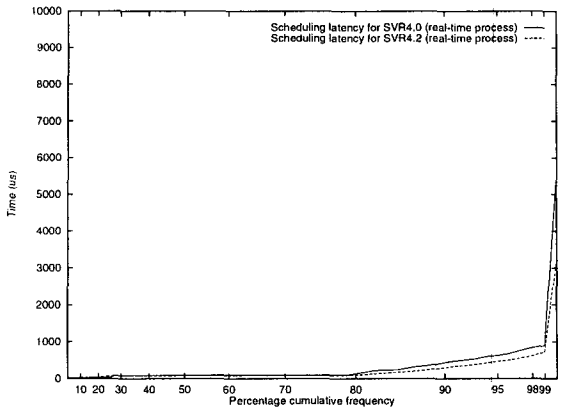

Figure 2: SVR4.0 (real-time) versus SVR4.2 (real-time)

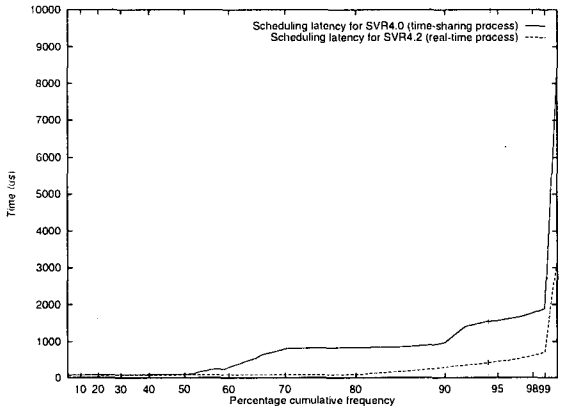

Figure 3: Scheduling Latency of SVR4.0 (time-sharing) versus SVR4.2 (real-time)

on the abscissa is called the Pth percentile of the distribution. For example, an 90th percentile of 1 millisecond means that 90% of the scheduling lattency times were at or below 1 millisecond. An exponential scale has been used on the abscissa in order to highlight the relatively small number of readings of particular interest. Results have been plotted at 1% intervals on the abscissa. The graphical results show that for the 20% of the sample measurements taken when find is executing and therefore making system calls in the kernel, better scheduling latencies are obtained with SVR4.2 than SVR4.0.

5 Performance Benefits of Preemption Points

In order to better understand the real performance impact that preemption points have on scheduling latency, two experiments were conducted using SVR4.2. The first executed the test process in the time-sharing class so that preemption points are not used (as if they were turned off). The second used the test process in a real-time class which therefore allows preemption to take

	Minimum	Median	Maximum	Mean	Standard Deviation
Scheduling latency (SVR4.2) (microseconds) (with preemption points on)	72	76	3272	121	184.45
Scheduling latency (SVR4.0) (microseconds) (with preemption points off)	72	72	7424	123	298.72

Table 4: Preemption (real-time) versus Non-Preemption (time-sharing) in SVR4.2

place in the kernel. The results obtained are presented in Table 4. The main deduction that can be made is that the use of preemption points improves worst case scheduling latency by a factor of two.

Figure 3 illustrates the overall improvement in scheduling latency from SVR4.0 (running a the test process in the time-sharing class) to SVR4.2 which runs the test process in the real-time class. The conclusion from this result is that there has been a significant shift from non-deterministic response a process experiences in the time-sharing class in SVR4.0 to a more deterministic behaviour for a process in the real-time class as illustrated by the scheduling latency percentile distribution for SVR4.2.

6 Improving Kernel Preemption

The major challenge for providing guaranteed deterministic response time is to make a kernel 100% preemptible. However, because most system calls used in monolithic kernels like UNIX tend to modify critical shared data structures such as process tables, semaphores and scheduling queues, it is difficult to achieve 100% preemptibility.

One approach that has been investigated in this paper is the use of preemption points at non-critical places in the kernel where preemption is possible.

Another method to achieve partial preemption is to assume that the kernel preemption is *always* enabled and to disable preemption when executing a critical region by simply setting a flag and clearing the flag to re-enable preemption after executing the critical code. The benefit of this method relative to using preemption points is reduced preemption latency. The disadvantage however is the overhead incurred in the setting and clearing of flags which can be quite large if the number of critical regions in the kernel is high.

An alternative approach to improving preemption in the kernel is the use of *multiple* semaphores rather than a single one to protect global data structures. Having each semaphore controlling access to an independently used data structure leaves other data structures free for access by other processes. Furthermore, no other process will be able to access the data structure which the preempted process is using as no other process has the required semaphore lock. Therefore, the kernel can be preempted at any point in its execution. This approach does give fast preemption times. However, its main disadvantage is that the entire kernel must be modified in order to assign various semaphores to the data structures used by the kernel. This can be a rather tedious process and is

likely to involve a large amount of work. The approach has frequently been used for multi-processor systems [1].

Another approach is to provide for kernel services using a shared library of services accessed via *subroutine calls*. Its main advantage over the traditional monolithic user-kernel interface is that it is much faster to access kernel services using the shared library approach because all that needs to be done is to link the operating services with the processes that use them. All that is needed is a subroutine call and the setting of a flag to indicate that the called subroutine is being executed in kernel mode. This compares to the trap interface where a trap instruction has to be executed each time a system call is made and there has to be a context switch from user-level to kernel mode. However, with the shared library approach, since the operating system and the application processes share the same stack, it is necessary to enforce some protection and security on the library code to avoid possible corruption of kernel code.

7 Conclusion

This paper has shown that interrupt latency has *not* improved in SVR4.2. Worst case interrupt latencies are in the order of hundreds of microseconds. More attention is required to minimize interrupt latency in future UNIX releases. Otherwise, the real-time responsiveness of the kernel to external events will be poor. Context switching has improved in SVR4.2 but is still quite high with the worst case in the order of milliseconds. The results have shown that SVR4.2 is still only partially preemptible. Preemption points give better scheduling latencies. However, they have a serious limitation as they cannot preempt interrupt processing. In order to achieve predictable, deterministic response time, the process dispatch latency must be minimized. This requires overheads like interrupt latency, context switching, scheduling latency and interrupt processing to be kept as low as possible.

8 Acknowledgements

The author would like to thank Steve Rago of Prologic for his valuable discussions and his encouragement to write this paper. The author also thanks Brendan Murphy of Cambridge University Computer Laboratory, UK for his help and support on many aspects of this work.

References

[1] M. J. Bach and S. J. Buroff. *Multiprocessor UNIX Operating Systems.* AT&T Bell Laboratory Technical Journal, 63(8):1733-1749, October 1984.

[2] P. G. Bond. *Priority and Deadline Scheduling on Real-Time UNIX.* In Proceedings of EUUG Conference, pages 201-208, October 1988.

[3] S. M. Doughty, S. F. Kary, S. R. Kusmer, and D. V. Larson. *UNIX for Real-Time.* In Proceedings of UniForum Conference, pages 222-230, 1987.

[4] N. Faller. *Measuring the Latency Time of Real-Time UNIX-like Operating Systems.* Department of Information Computer Sciences Institute, University of California at Berkeley, 1992. Technical Report No. 37.

[5] B. Goodheart and J. Cox. *The Magic Garden Explained: The Internals of UNIX System V Release 4.* Prentice Hall, 1994.

[6] K. Marrin. *Multithreaded Real-Time Operating Systems.* Computer Design, pages 77-88, March 1993.

[7] S. Rago. February 1994. Private communication.

[8] K. Thompson and D. M. Ritchie. *The UNIX Time-Sharing System.* CACM, 17(7):365-375, July 1974.