UNIX[1] AND THE REAL-TIME TELEMETRY SYSTEM

William N. Waggener
Technical Director
Fairchild Weston Data Systems
P.O. Box 3041
Sarasota, FL 34230-3041

ABSTRACT

Since the first computer-based telemetry systems were designed in the mid-1960's, the operating system has been the nemesis of the system software designer. The requirement to acquire telemetry data at high rates, in real time, without loss, is in direct conflict with the direction computer operating systems have taken over the last two decades. The "lean and mean", single user operating systems of the 1960's have been replaced by multi-tasking, multi-user systems which emphasize multiple applications at the expense of real-time performance. Recently, there has been enormous interest in hosting real-time telemetry systems under the UNIX operating system. From an applications standpoint UNIX has much to offer the user but it certainly complicates the life of the real-time system software designer. In this paper, a critical look is taken at the role of the operating system in a real-time telemetry system with particular emphasis on the use of UNIX POSIX and real-time extensions.

INTRODUCTION

Despite the title and abstract, this paper deals more with system concepts than with software. The marriage of the real-time telemetry system with the general purpose, digital computer dates back to the mid-1960's. During the intervening two decades, the technology of both the telemetry "front end" and the host computers have changed enormously although the system functionality remains relatively unchanged. The telemetry subsystem acquires, formats and stores the data from one, or more, data streams. The system is real-time in a very concrete sense; if the

[1] UNIX is a trademark of AT&T

data is not processed, it is irretrievably lost. Common sense dictates the use of backup storage, however, there is a large burden on the telemetry system and it's host processors to maintain real-time processing.

Fueled by the rapidly increasing technology and decreasing costs of computer technology, users are demanding more sophisticated processing in the telemetry subsystem and the telemetry system is becoming more software intensive each year. Users are also becoming much more sensitive to the costs associated with their growing software investments. The advantages of software "portability" from one system to another, and from one generation to another are quite evident. This has, naturally, lead to interest in operating systems which run on a variety of hardware and on the use of portable, high level languages. In this paper, attention is focused on the impact of the operating system, specifically UNIX, on the real-time telemetry system.

DATA ACQUISITION AND THE OPERATING SYSTEM

The tasks associated with telemetry data acquisition software have not significantly changed since the first general purpose computer-based telemetry systems were designed in the mid-1960's. A top-level data flow diagram of the data acquisition subsystem is shown in Figure 1. The telemetry front end (TFE) hardware requires setup and control information and, in turn, supplies one, or more, data streams to the data acquisition process. Since the very inception of the telemetry/computer system, the primary task of the acquisition software is to format the input data on disk or magnetic tape. Secondary tasks include processing of "quick look" data, providing data to application programs and other peripherals and responding to operator commands.

As computer and TFE technology has increased, the basic tasks of the data acquisition software have evolved, both in response to technology, and to changes in the operating system environment. Figure 2 shows a simplified data flow of the acquisition portion of a typical telemetry system operating under a multi-tasking, multi-user operating system such as DEC's VMS. Data streams from the TFE interface with the computer system using a custom buffered I/O channel which provides both data buffering and direct memory access (DMA) control. All of the input data is directed toward ring buffers in the host memory from which a data formatting module accesses the raw data and formats it to high speed

disk. Selected data is directed by the buffered I/O to a local database, commonly referred to as the current value table (CVT), where it is accessed for “quick look” display and derived parameter processing. Applications programs access both the formatted data and the CVT data for further processing.

Variations from the simplified data acquisition flow diagram result from differing system configurations, performance requirements, composite data rate and so on. The use of a preprocessor has a major impact on the acquisition process as does the operating system environment. Early systems using single user, foreground/background tasking operating systems performed little more than disk, or tape, formatting tasks. With these operating systems, interrupt latency times were measured in microseconds and data streams could be processed on a frame-by-frame basis. The acquisition process could be given the highest priority level and the acquisition process could gain total control of the processor. This was the heyday of the real-time telemetry system. With a mid-1960’s dual bus mini-computer, data transfer rates to memory of over 500 kilowords (16-bit words) per second were achievable. As operating systems became more complex and incorporated multi-tasking, frame rate processing became impossible as interrupt latencies went from microseconds to hundreds of milliseconds. The change in the operating system environment has had a major impact on the design of the data acquisition software design. The appearance of the preprocessor and the CVT are direct reflections of the operating system changes. Moving telemetry systems to UNIX based systems is expected to make further changes in the software architecture of telemetry data acquisition systems.

UNIX AND REAL-TIME PROCESSING

The history of the UNIX operating system is well-known[1] and the system has many champions and detractors[2]. The advocates of UNIX point to portability of UNIX based systems over a wide variety of computer platforms as a major argument for adopting UNIX. The opponents cite weaknesses in the technical details as well as an image of a programmers system with it’s cryptic shell commands. Both sides will agree, however, that standard UNIX (whatever, that is) is totally unsuited for real-time applications without some significant changes.

UNIX was designed to support a time-sharing environment with strong software development ties. There are three distinct components to UNIX:

Scheduler
File System
Shell and Utilities

The scheduler and the file system are part of what is commonly called the kernel. The shell and the vast number of UNIX utilities are one reason for it's popularity; however, it is the kernel which determines it's suitability for real-time applications.

The hierarchy of the typical operating system is illustrated in Figure 3. The typical UNIX implementation includes the basic kernel, a set of standard system calls, one, or more, shells and a set of utilities. An application program calls the operating system kernel through the standard system calls to perform I/O, file handling and other system related functions. A real-time system must also communicate directly with the system hardware which, normally, requires circumventing the standard operating system calls. When the standard system calls are circumvented or extensions to the operating system are used, the code is non-portable.

Some of the major functions of the UNIX kernel are briefly considered in the following paragraphs to identify weaknesses in UNIX for real-time applications with the modifications required to address the real-time system. This discussion is at a high level and the reader interested in more in-depth treatment of the various functions is directed toward the reference material[3456].

Scheduler

The standard UNIX scheduler is a time-sliced scheduler which runs processes in the run queue in a priority established by the kernel. Unfortunately, the scheduler is extremely democratic and adjusts process priorities in accordance with the CPU usage history of the process giving higher priority to those processes with low usage. This is exactly opposite the strategy required for the real-time process. Furthermore, standard UNIX does not permit preempting lower priority processes by higher priority processes until the process is blocked, run to completion or reached the end of it's time slice. Although an application can alter the

process priority using the nice(2) function, this does not avoid the preemption problem. System kernel calls by low priority processes can not be preempted and a high priority process must wait until the call is complete.

Real-time extensions to UNIX require a redesign of the scheduler such that any process on the system can have exclusive use of all of the system resources and high priority processes can preempt lower priority processes both in the user and the kernel modes.

Interrupt and Event Handling

Interrupt and event handling is a crucial issue in many real-time systems and vendors of real-time UNIX extensions stress performance measures such as context switch latencies. It is important that the system respond to a request for service not only rapidly, but also with a guaranteed maximum time. Because of a lack of preemption and other features, standard UNIX is both slow and cannot guarantee a bounded service time. It can be confusing to compare performance figures for the service latency between vendors because of differences in how the latency times are specified. When a process requires service, the latency depends on a sequence of operations and on the load on the system. The service latency is illustrated in Figure 4 and depends on the interrupt service time, the kernel preemption latency, the context switching time and, finally, the process dispatch code latency.

Total service latencies ranging from about one millisecond to about ten milliseconds have been reported for real-time UNIX extensions running on 68K and 386 processors. These are impressive times for UNIX based systems but are a far cry from the single-user dedicated operating system. Such is the price of progress.

File Handling

File handling is a major task of the operating system and a major deficiency of standard UNIX. UNIX has a bad habit of scattering files all over the disk and requiring that all file transfers take place through operating system buffers. As a result, file transfer rates of less than 200 kilobytes per second have been reported on machines as powerful as the VAX 11/780.

A real-time UNIX must be able to provide contiguous files on disk and to provide paths for direct writing to files without kernel buffering.

Memory Locking

UNIX allocates memory to running processes and when more memory is required it swaps processes out to disk. This is also unacceptable for real-time processes and a common real-time extension provides some mechanism for locking processes in memory.

Inter-process Communication

The mechanism provided by UNIX for inter-process communication relies on signals. Unfortunately the signals are quite primitive and do not even identify the process sending the signal. Proposed extensions provide a much more capable inter-process communication mechanism.

Asynchronous I/O, DMA and Timers

In standard UNIX a process can be suspended while waiting for an I/O request to complete. Real-time extensions are required in this area to permit DMA transfers. The standard UNIX timers have a coarse resolution and improved timer resolution is also required.

POSIX AND PORTABILITY

In response to the desire to provide an operating system interface and environment which permits applications to be “ported” from one computer to another, a committee sponsored by the IEEE Computer Society created the POSIX IEEE Std 1003.1-1988[7]. The POSIX Standard defines the standard system calls shown previously in Figure 3. A simplified summary of the system calls defined by the POSIX Standard is shown in Figure 5. The standard system calls defined by POSIX can be grouped into six general categories:

Process Primitives
Process Environment
Files and Directory Calls
Input/Output Primitives
Device and Terminal Functions
System Database Access and Miscellaneous

There are several important points which must be recognized. First, POSIX does not define the kernel implementation. Secondly, the current POSIX Standard is “plain vanilla” and does not support any real-time extensions. Finally the interface is defined only in terms of C language bindings.

The implications of the first two points are that POSIX conforming systems will not currently support real-time applications for the reasons cited for standard UNIX. The IEEE Computer Society Working Group 1003.4 is currently working on real-time extensions to address most of the issues discussed previously. Two important issues which are not addressed by the 1003.4 group are kernel preemption and support for multiprocessors. Both of these issues are extremely important for real-time applications. The POSIX real-time extensions are not expected to be issued until 1991 and further extensions are expected to be required to provide the real-time performance required of telemetry systems.

The current POSIX Standard is defined in terms of C language bindings. Working group 1003.5 has been chartered to provide Ada language bindings. At the moment, the author is not aware of any FORTRAN bindings for POSIX. Despite it’s age, FORTRAN continues to be an important language for telemetry applications and either POSIX bindings are required or mixed C/FORTRAN or Ada/FORTRAN software will have to be developed. Although it sounds almost trivial, the development of FORTRAN bindings can lead to some interesting problems because of the fundamental differences between C functions and FORTRAN functions. Because C functions can either return a value or be called like a procedure, deciding whether to implement a FORTRAN call as a function or as a subroutine can be a dilemma for the designer.

SUMMARY AND CONCLUSIONS

Standard UNIX is not ideally suited for real-time telemetry applications primarily because of the design of the scheduler, the file system and general I/O handling. The current POSIX Standard does not address these issues, however, the 1003.4 real-time extension will address many of the important real-time deficiencies. The real-time POSIX extensions will not address all of the issues even when it is released and additional, proprietary extensions will probably be required to provide adequate real-time performance for many telemetry applications.

In the interim, real-time performance can be achieved for many telemetry applications using UNIX based operating systems with proprietary extensions.

It is very important for the user to understand that the data acquisition portion of the software system is probably not portable and requires rewriting for a new platform. The user will, however, gain the important advantage of the ability to port a large portion of applications programs which do not call system specific real-time extensions. When the POSIX real-time extensions are defined, a POSIX-conforming system will be able to port an even larger portion of the system software to a new platform.

Even with a fully compliant POSIX system with the real-time extensions, real-time performance will vary from platform to platform in accordance with the vendor specific kernel implementation. When throughput performance is paramount, the system design will most likely require a dedicated data acquisition processor running under a real-time kernel and controlled by a real-time UNIX with multiprocessor extensions. **A trade between performance and portability is inevitable.**

UNIX based telemetry systems are here to stay with the help of real-time extensions. A portion of the software will be uniquely tied to proprietary extensions to UNIX but a greater portion of applications software will be portable.

REFERENCES

1. Emerson, Sandy,"The UNIX Universe", Datamation, August, 1984, page 76

2. Small, C. H., "UNIX Operating Systems", EDN, May 17, 1984, pp. 103-123

3. Riche, D.M., Thompson, K., "The UNIX Time Sharing System", Bell System Technical Journal, Vol. 57, No. 6, Part 2, July-August 1978, pp. 1905-1929

4. Thompson, K., "UNIX Implementation", Bell System Technical Journal, Vol. 57, No. 6, Part 2, July-August 1978, pp. 1931-1946

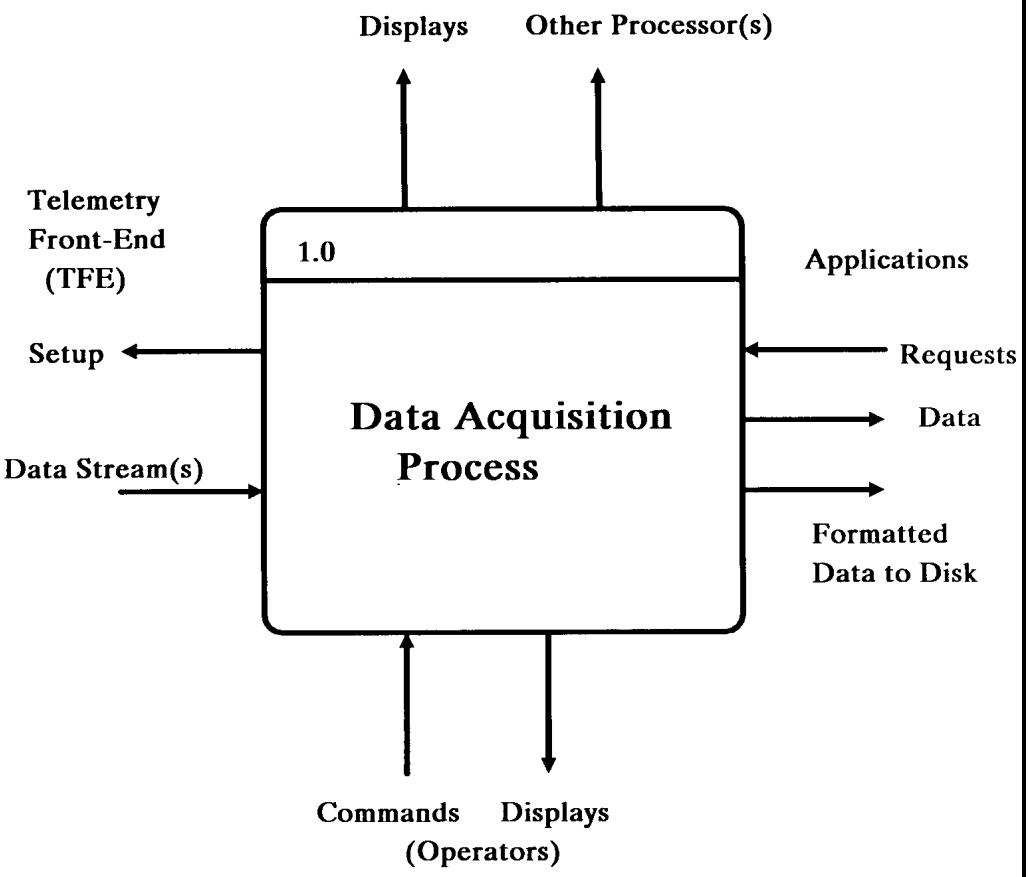
5. Bach, M.J., The Design of the UNIX Operating System, Prentice-Hall, 1986

6. Ramamurthy, G., "An Analytical Model For UNIX Systems", AT&T Technical Journal, September/October, 1988

7. IEEE Standard Portable Operating System Interface for Computer Environments, IEEE Std 1003.1-1988, IEEE 1988

Figure 1

Data Acquisition Data Flow Diagram



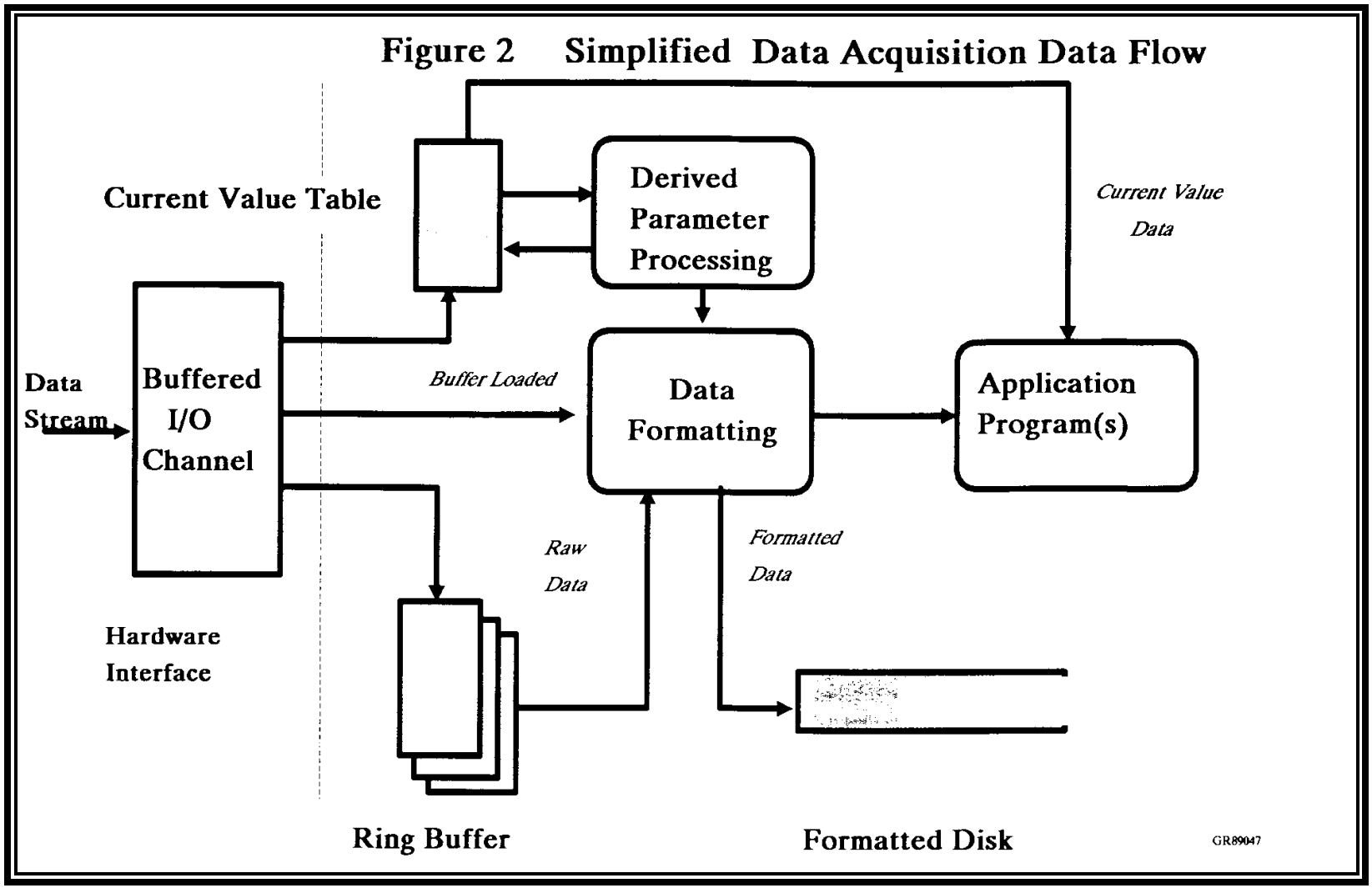


Figure 2 Simplified Data Acquisition Data Flow
Current Value Table
Derived Parameter Processing
Current Value Data
Data Stream
Buffered I/O Channel
Buffer Loaded
Data Formatting
Application Program(s)
Raw Data
Formatted Data
Hardware Interface
Ring Buffer
Formatted Disk
GR89047

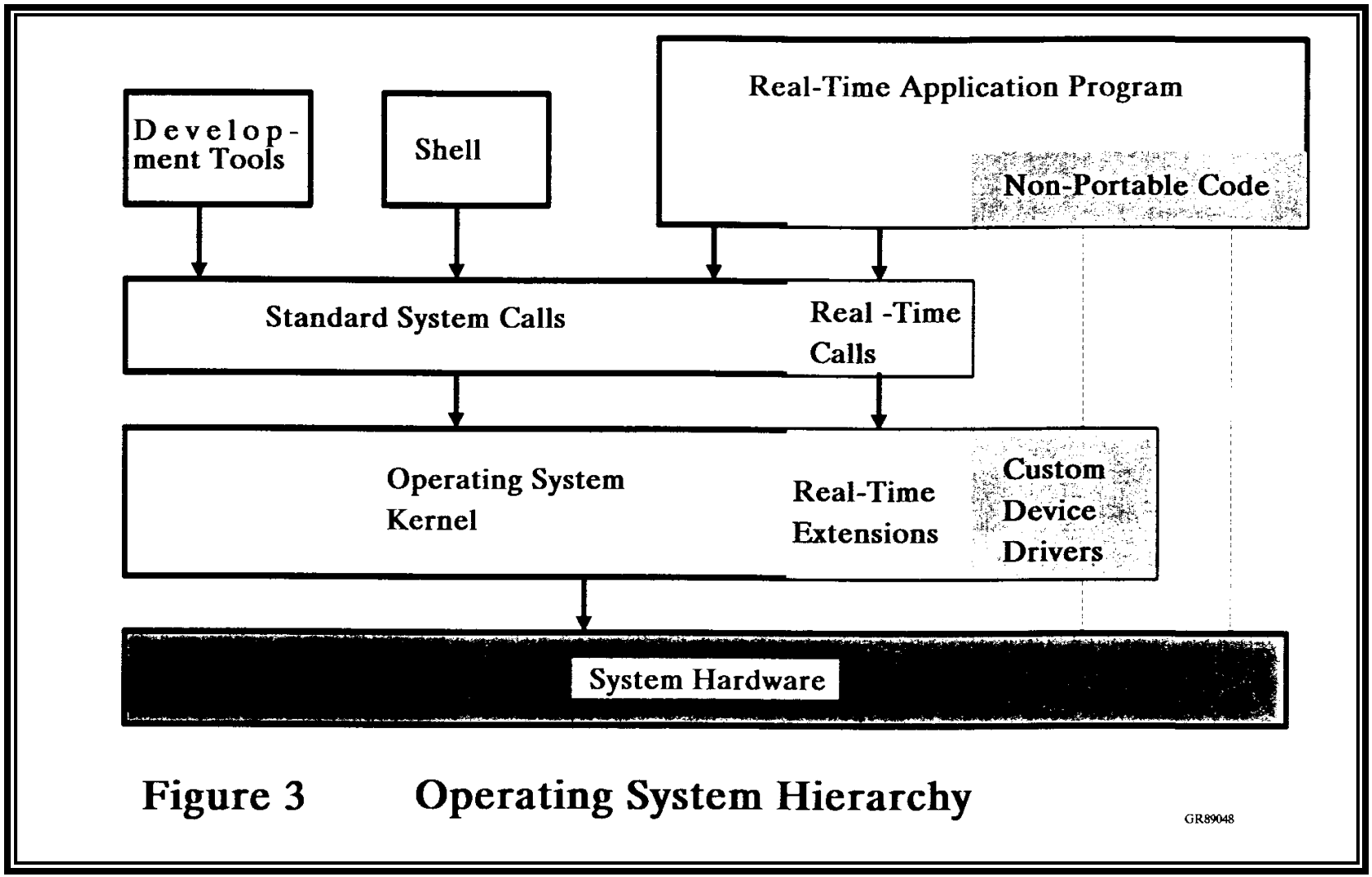


Figure 3 Operating System Hierarchy

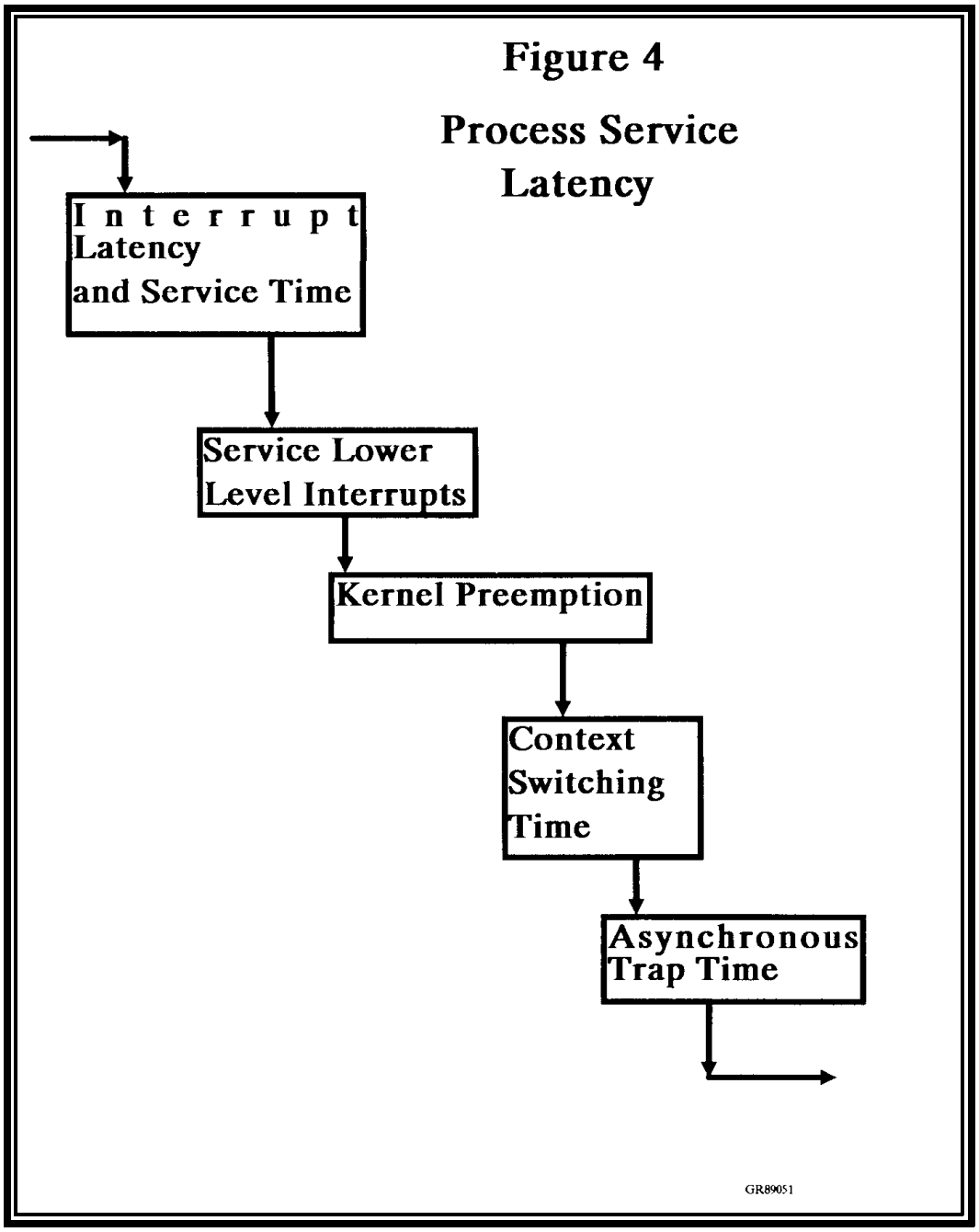
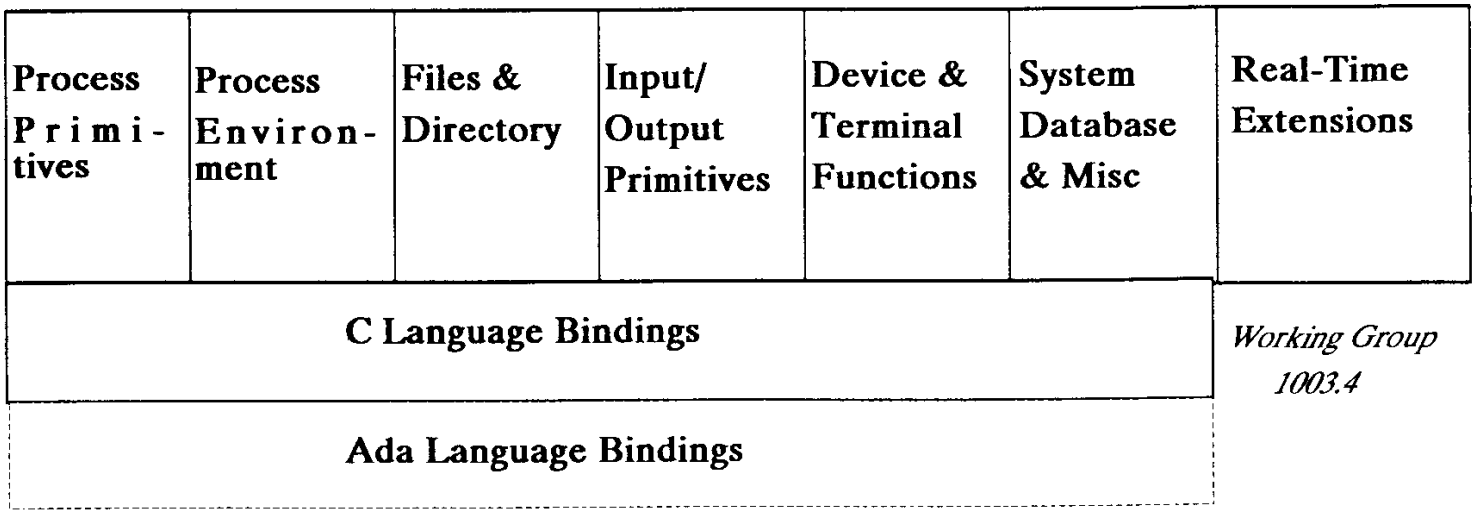


Figure 4
Process Service Latency
Interrupt Latency and Service Time
Service Lower Level Interrupts
Kernel Preemption
Context Switching Time
Asynchronous Trap Time
GR89051

Figure 5

POSIX
IEEE Std 1003.1 - 1988



GR89049