The Space Congress® Proceedings

1986 (23rd) Developing Space For Tomorrow's Society

Apr 1st, 8:00 AM

The Use of UNIX in a Real-Time Environment

Robert D. Luken
Chief, Digitial Applications Branch Engineering Development Directorate NASA, John F. Kennedy Space Center, Florida

Peter C. Simons
Project Lead, LPS-II Scientific Systems Services Melbourne, Florida

THE USE OF UNIX IN A REAL-TIME ENVIRONMENT

Robert D. Luken
Chief, Digitial Applications Branch
Engineering Development Directorate
NASA, John F. Kennedy Space Center, Florida

Peter C. Simons
Project Lead, LPS-II
Scientific Systems Services
Melbourne, Florida

ABSTRACT

This paper describes a project to evaluate the feasibility of using commercial off the shelf hardware and the UNIX (trademark of AT&T Bell Laboratories) operating system, to implement a real time control and monitor system. A functional subset of the Checkout, Control and Monitor System (CCMS) was chosen as the testbed for the project. The project consists of three separate architecture implementations. A local area bus network, a star network, and a central host. The motivation for this project stemmed from the need to find a way to implement real-time systems, without the cost burden of developing and maintaining custom hardware and unique software. This has always been accepted as the only option because of the need to optimize the implementation for performance. However, with the cost/performance of today's hardware, the inefficiencies of high-level languages and portable operating systems can be effectively overcome.

INTRODUCTION

To understand the problems posed by the use of UNIX in a real-time environment, it is necessary to have some understanding of the history of UNIX, the history of CCMS, and the hardware configurations being implemented. The project entails the detailed development of three prototype configurations for evaluating the feasibility of "OFF-THE-SHELF" hardware and software for use in a second generation Launch Processing System (LPS). The first stage of this prototyping effort was to implement the Ground Operations Aerospace Language (GOAL) and its related interfaces to ground support equipment. GOAL is a unique language designed to run in a supporting real-time environment custom designed for the Checkout Control and Monitor System (CCMS) used for Shuttle launch operations. The groundrules for accomplishing this task were as follows:

Use a highly portable high level language for all coding functions. The "C" language was selected because of its demonstrated capability for implementing system level software.

Use an unmodified off the shelf operating system. The AT&T UNIX System V release 2 was selected for portability (available for most vendors hardware) and compatibility with the "C" language.

Use an unmodified commercially available computer with UNIX System V capability. The AT&T 3Bx family of computers was chosen, because of its full compatibility with System V and vendor support for UNIX. The models used for prototyping ranged from the 3B2/400 to the 3B15, both 32 bit machines.

Use an industry standard network interface for communications between remote CPU's. These include Host 3B Computers, Data Acquisition Processors and Display Processors. Ethernet was chosen as the test network, due to its wide availability.

Use an off the shelf PC type computer to develop a Data Acquisition Processor, using "C" to do data polling and exception monitoring on analog and discrete measurement data. The AT&T 6300 PC was chosen for this task because of its performance (8 mhz clock) and availability.

Use an off the shelf PC type computer to develop a Display Processor, using "C" and the Graphics Kernel System (GKS) graphics standard to provide the operator interface. The IBM PC AT with Professional Graphics Adapter and Monitor was chosen for this task because of its graphics capability and availability.

HISTORY OF UNIX

During the early 1960's, computers were expensive and had small memories. For example, one middle-priced work horse of that day, the IBM - 1620, had only 24K words of memory, and was capable of storing about 40,000 numbers. The primary design criteria at that time, was for all software (languages, programs, and operating systems) to use memory efficiently and to make programs execute as fast as possible. This was usually at the cost of being unwieldy for the programmer and other users.

UNIX grew out of the need for a more productive alternative to this early time - consuming software. UNIX was created in 1969 at Bell Laboratories, the research arm of the American Telephone and Telegraph Company. It began when Ken Thompson, a programmer at Bell Labs, decided to try to create a more cost effective and usable programming environment.

Ken Thompson was working on a program called SPACE TRAVEL that simulated the motion of the planets in the solar system. The program was being run on a large computer made by General Electric, the GE-645, which ran an operating system called Multics. Multics was developed at MIT and was one of the first operating systems designed to handle multi-tasking. However, its use on the GE computer was expensive and awkward. Thompson obtained a small computer made by Digital Equipment Corporation called the PDP-7. He began the burden of transferring his SPACE TRAVEL program to run on the smaller computer. In order to use the PDP-7 conveniently, Thompson created a new operating system that he christened UNIX, a play on the word Multics, since it incorporated a simpler implementation of the multi-tasking feature of that system. Thompson was successful enough in this effort to attract the attention of Dennis Ritchie and others at Bell Laboratories, where they continued the process of creating a streamlined multi-user multi-tasking environment.

Under Thompson, Ritchie and others, UNIX became operational in the Bell Laboratories patent organization in 1971 on a PDP-11/20. UNIX had evolved from the best ideas of its predecessors. During the early 1970's UNIX ran primarily on computers that were manufactured by Digital Equipment Corporation; first on the PDP-7 and, then, on the PDP-11 family, where it achieved widespread acceptance throughout Bell Labs. During the same time, universities and colleges, many of which were using the PDP-11/70 computers, were given license to run UNIX at minimal cost, since AT&T was not allowed to sell computers and software. This move by AT&T eventually led to UNIX being run at over 80% of all university computer science departments in the United States. Each year, thousands of computer science students graduate with some experience in running and in modifying UNIX.

UNIX, like most operating systems, was originally written in what is called "assembly language." This is the primitive set of instructions that controls the computer's operations. Since each computer has its own unique set of internal instructions, moving UNIX or any other operating system to another computer involves a significant programming effort. The solution to this problem, and perhaps the key to UNIX's popularity today was the decision to rewrite the operating system in a higher level language; - one more portable than assembly language.

The language was called B. In 1973 it was modified extensively by Dennis Ritchie and renamed C. A powerful control language featuring modern data structures, C is much easier to use and understand than assembly language. Although there is some overhead in program execution, it is outweighed by its productivity as compared to assembly code.

The use of C makes UNIX easily portable to other computer systems, since only a small portion of the UNIX kernel is written in assembly code. This has allowed the system to be easily ported to a number of other computers, including IBM 370, Honeywell, Amdahl 470 and Cray II.

In January of 1983, AT&T announced that it was licensing a standard version of UNIX for the commercial marketplace. This version was called UNIX System V, and was based on the version in use internally at Bell Labs. It contained many of the features found in Berkely UNIX 4.2 BSD, a derivative of UNIX System III.

At the same time that AT&T announced UNIX System V, it also announced that it was licensing the top three semiconductor manufacturers of 16/32 bit microprocessors (Intel, Motorola and National) to develop "ports" for their products. Each port consisted of a microprocessor, a ROM, and System V software.

AT&T required manufacturers to reproduce 98% of all known bugs in UNIX System V. This means that except for execution speed, the operating system will behave the same regardless of which hardware it is running on. This standardization means that software developers are assured that their programs will work on the largest number of machines. This also means that hardware can be improved without affecting software compatibility.

HISTORY OF CCMS

The Checkout Control and Monitor System (CCMS) was developed at the Kennedy Space Center during the early 1970's and became operational in 1976. CCMS is a system of hardware and software used to monitor and control the operations associated with the checkout and launch of the Space Shuttle. This includes facilities, ground support equipment (GSE), and flight interfaces. Also included are systems at other locations, such as Vandenburg AFB and the Shuttle Avionics Integration Laboratory at Johnson Space Center.

The CCMS is a distributed set of minicomputers (up to 64 in a cluster), performing front end processing (FEP) and application program processing (CONSOLE), tied together through a Common Data Buffer (CDBFR). The CDBFR is a high speed multi-ported memory subsystem that can be partitioned into read/write areas for the computers connected to it. The CDBFR acts as a global data storage area and as a communications switch. Through an interrupt structure within the CDBFR, one computer can send a packet of data or commands to another computer.

Each FEP is a Modcomp minicomputer with special interfaces and two auxiliary processors, that is used for data acquisition and control of a GSE or flight interface. Data rates require FEP's to process a data byte in real-time, as often as one every 100 microseconds. The processing consists of two sets of high and low limit checks, significant change checks, control logic responses, exception handling, CDBFR interface handling, and responses to commands from consoles to change sampling rates, inhibit exception checking, etc. The operating system contained within the FEP is a special purpose, highly optimized design that takes advantage of the pipeline capability of the three processors. It is implemented in assembly language, with the auxiliary processor functions being implemented in microcode (a level below assembly).

Each CONSOLE is a Modcomp minicomputer similar to the one used in the FEP. It has the same two auxiliary processors, but has additional peripherals. The peripherals consist of moving head disk, auxiliary storage, and three-channel display generator. The processing in the CONSOLE consists of executing up to six concurrent GOAL (Ground Operations Aerospace Language) application programs (with three levels each), exception monitoring, display monitoring, and the associated data interface handling. The operating system contained in the CONSOLE is a superset of the one used in the FEP. It too is implemented in assembly code with microcoded functions in the auxiliary processors.

The GOAL executor is at the heart of the CCMS and is the portion of the system which attempts to utilize the operating system in a real-time environment. GOAL is an interpretive language, that is, it is compiled to an intermediate form which is, in turn, interpreted by an operating system resident executor. GOAL has the ability to react to external interrupts from various sources, these sources include, measurement exceptions, programmable function keys and panels, count down time and GMT interrupts and interrupts from other keyboard functions. At the center of the GOAL system is the Interpretive Code Processor, its responsibility is to parse and execute the GOAL intermediate code to carry out program execution. This is also the module which redirects flow of control when informed of an interrupt it is expecting.

When the CCMS was designed and implemented, the power and performance of minicomputers was very limited. At that time only about four minicomputers could contain and address more than 64k bytes of main memory (core). They were also very slow, about one microsecond for a memory access, compared with today's 100 to 200 nanosecond access. This resulted in a real need to squeeze the maximum amount of performance out of the hardware and to make it all fit in less than 64k words of memory. Therefore software could not be kept hardware independent and portable, nor implemented in a high level language. The use of processors with microcoded functions, further locked the hardware and software into an inseparable combination, with no easy replacement options; but was necessary to get the needed performance from the hardware available.

PROTOTYPE DESCRIPTION

The prototype for the evaluation project consists of a functional representation of a single string of the CCMS composed of off the shelf hardware and software. The intent of the prototype is to evaluate the performance of existing CCMS GOAL language applications running under the UNIX operating system hosted on three different hardware configuration phases. The hardware implementation for the first phase of the project has a Data Acquisition Processor that is interfaced with standard CCMS GSE, two Display Processors, and an Application Processor, connected together with Ethernet. The purpose of this phase is to represent a typical local area network.

The hardware implementation for the second phase of the project consists of the same hardware modules as the first but connected together with a CDBFR, as used in CCMS, instead of Ethernet. The purpose of this phase is to represent a CCMS hardware compatible configuration and evaluate the feasibility of replacing existing FEP's and CONSOLES.

The hardware implementation for the third phase of the project consists of the Data Acquisition Processors and Display Processors connected to a large Host computer that handles all the application and data processing functions. The software implementation for the project consists of a rewritten (in the "C" language) version of the GOAL executor (CGOAL), an Interrupt Processor, an External Events Driver, and the Broadcast Receiver. The Interrupt Processor handles the routing of interrupt data between the Data Acquisition and Display Processors. The External Events Driver provides the interface between the GOAL executor and the outside world, and allows existing application programs to execute as if they were in the CCMS environment. The Broadcast Receiver handles exception messages sent by the Data Acquisition Processor.

SOFTWARE DESIGN

The CGOAL executor is at the heart of the prototype and is the portion of the system which attempts to utilize the UNIX system in a real-time environment. GOAL has the ability to react to external interrupts from various sources, these sources include: measurement exceptions, programmable function keys and panels, count down time and GMT interrupts and interrupts from other keyboard functions. Interrupts posed the biggest problem in the development of the CGOAL real-time system. There is no access to the actual interrupt structure of the 32100 microprocessor because this would threaten the portability of the UNIX system. This factor made it obvious that immediate attention to interrupts was not possible in an unmodified System V UNIX. The Interpretive Code Processor is responsible for parsing and executing the GOAL intermediate code. This is also the module which would redirect flow of control when informed of an interrupt it is expecting. It became obvious that the I/C Processor was busy enough processing interpretive code and did not need the added responsibility of managing the network, handling requests to external CPU's and processing possible interrupt data. For this reason it was decided to take advantage of UNIX's powerful multiprocessing capabilities.

The CGOAL system was planned to run as several co-processes: the I/C Processor, the Interrupt the External Events Driver (EED) and the Broadcast Receiver. The I/C Processor operates as previously stated. The Interrupt Processor acts as a clearing house for interrupt data received from the Display or Data Acquisition Processors. It is the Interrupt Processor's duty to sort through incoming data and pick out information for which the I/C Processor has specified an interrupt vector. Should an interrupt be detected, the specified interrupt vector is passed to the I/C Processor who re-routes flow of control depending on that vector. Interrupts not specified are ignored by the processor. The External Events Driver is the systems link to the outside world. The EED's major task is to accept requests from the I/C Processor to manipulate or read outside data, format the data into an Ethernet packet structure, send the packet via the Ethernet LAN and wait on the response. The same sequence of events occurs for requests to the Display Processor work station. The Broadcast Receiver has a single, clear cut task: monitor the network's Broadcast Port to receive exception message Broadcasts from Data Acquisition Processors. These Broadcasts are sent to the Interrupt Processor for processing.

Since several processes run concurrently, for CGOAL, the problem of interprocess communication became apparent. The UNIX System V Interprocess Communications package (IPC) handled this problem easily. Several queues were set up and dedicated to specified communication paths between processes. The only problem expected with this was linked to the fact that 10 separate CGOAL processes can run system-wide, which could cause some confusion as to the ownership of the

messages on the queues. This was corrected by placing task numbers in the messages as protocols [Drawing attention to the fact that a message was pending on an incoming queue for a process]. The IPC package provided several sets of semaphores which were used to solve this problem. Various semaphores are set to signal arrival of messages and the number of messages pending on the queue. These semaphores are also used to keep general system data and to signal processes to perform actions for which an IPC message is not necessary. These semaphore values are checked periodically in each process as they loop through a predetermined sequence. The I/C Processor checks for interrupts pending and other messages after completion of execution for each statement. The EED checks for incoming Ethernet data or IPC requests from the I/C Processor each time through its idle loop, etc.

The major stumbling block is overcoming UNIX's context switching routine. Context switch times and orders are entirely controlled by the UNIX kernel. Changing these routines would require access to kernel-level procedures. This access has been restricted by the creators of UNIX to insure portability between target machines. Because of this task switching routine, the processes can spend too much time in idle loops waiting on semaphores to change when the processes required to change them are not even running. A means exists to cause a process to suspend until a time value expires, but this time value can at best be 1 second which is far too long for a critical process to be idle. Research is currently being conducted into the use of system signals to wake up an idle process, but the lack of available signals and the various scenarios in which a process can be activated leave little hope for this method.

UNIX overhead in process accounting also steals execution time from the real-time system. It is possible to deactivate some of the process accounting, but the general system and disk accounting tasks are firmly anchored in the system kernel.

Some research was also done into the possibility of changing the priorities on various modules in the CGOAL system to try and improve performance. This did not seem to solve anything because a higher priority task could steal CPU time from a lower priority, though equally important, task. No significant performance enhancement was observed.

PRELIMINARY RESULTS

Completion of the Phase 1 of the prototype revealed some important results:

UNIX would support a pseudo real time system for demonstration purposes. A real system implementation would require architectural refinements that would optimize the strengths and weaknesses of a UNIX based system.

Though slower than the current LPS system (only the 3b2 has been tested), faster UNIX based machines would allow the performance levels needed for actual control and checkout functions. Use of the 3b15 should give a good indication of the difference in performance due to a faster processor. This coupled with handling the critical real-time functions with front-end hardware would isolate the event driven processes from the majority of the application tasks. UNIX task switching functions and system accounting provide too much overhead for actual real-time execution. The system degrades rapidly as more real-time concurrencies are run. Some modification to task switching and an ability for a process to relinquish control to the task dispatcher is necessary. For actual control of interrupt functions some access to UNIX kernel level programming will probably be needed. New developments in real-time UNIX enhancements by private companies appear to greatly improve performance of UNIX in a real-time environment.

Ethernet is probably not fast enough to transfer large amounts of data to and from processors in an acceptable time. This is due mostly to the layers of protocol involved with Ethernet interfacing. Intelligent controllers could alleviate part of this problem, by handling higher levels of the protocol in the controller. Also token passing networks appear to offer less degradation at higher utilization rates than collision detection type networks, such as Ethernet.