Copyright © 1978 American Telephone and Telegraph Company
THE BELL SYSTEM TECHNICAL JOURNAL
Vol. 57, No. 6, July-August 1978
Printed in U.S.A.

UNIX Time-Sharing System:

A Minicomputer Satellite Processor System

By H. LYCKLAMA and C. CHRISTENSEN
(Manuscript received December 5, 1977)

A software support system for a network of minicomputers and microcomputers is described. A powerful time-sharing system on a central computer controls the loading, running, debugging, and dumping of programs in the satellite processors. The fundamental concept involved in supporting these satellite processors is the extension of the central processor operating system to each satellite processor. Software interfaces permit a program in the satellite processor to behave as if it were running in the central processor. Thus, the satellite processor has access to the central processor's I/O devices and file system, yet has no resident operating system. The implementation of this system was considerably simplified by the fact that all processors, central and satellite, belong to the same family of computers (DEC PDP-11 series). We describe some examples of how the SPS is used in various projects at Bell Laboratories.

I. INTRODUCTION

The satellite processor system (SPS) and the concept of a satellite processor have evolved over the years at Bell Laboratories to provide software support for the ever-increasing number of mini- and microcomputer systems being used for dedicated applications. The satellite processor concept allows the advantages of a large computing system to be extended to many attached miniprocessors, giving each satellite processor (SP) access to the central processor's (CP)

file system, software tools, and peripherals while retaining the real-time response and flexibility of a dedicated minicomputer. Since the cost of the peripherals for a minicomputer often far exceeds the cost of its CPU and memory, the CP provides a pool of peripherals for the support of many SP's. Although each SP requires a hardware link to a CP, the idea of a satellite processor is basically a software concept. It allows a user program, which might normally run in the CP using its operating system, to run in an SP with no resident operating system.

This paper describes the hardware and software required for SPS, the concepts involved in SPS, and how these concepts can be extended to provide even more powerful tools for the SP. Several examples of the use of the SPS in Bell Laboratories projects are described.

II. HARDWARE CONFIGURATION

The particular SPS hardware configuration described here consists of a DEC PDP-11/45 central computer[1] with a number of satellite processors attached using a serial I/O loop[2] as one of the communication links between the SP's and the CP (see Fig. 1). Other satellite processors are attached using DR11C, DL11, and DH11 devices (see below). Each SP is a member of the DEC PDP-11 family of computers, with its own set of special I/O peripherals and at least 4K 16-bit words of memory. A local control terminal is optional. The central computer has 112K 16-bit words of main memory and 96 megabytes of on-line storage. Eight dial-up lines and various other terminals are available for interaction with the UNIX* time-sharing system,[3] supported by the MERT operating system.[4] Magnetic tape is available as one peripheral device for off-line storage of files. Access to line printers, punched card equipment, and hard-copy graphics devices is available through the connection to the central computing facility for Bell Laboratories.

III. COMMUNICATION LINKS

A number of satellite processor systems have been installed in various hardware configurations using both the UNIX and the MERT operating systems. The devices supported as communication links include the serial I/O loop mentioned above, the DL11 asynchronous

* UNIX is a trademark of Bell Laboratories.

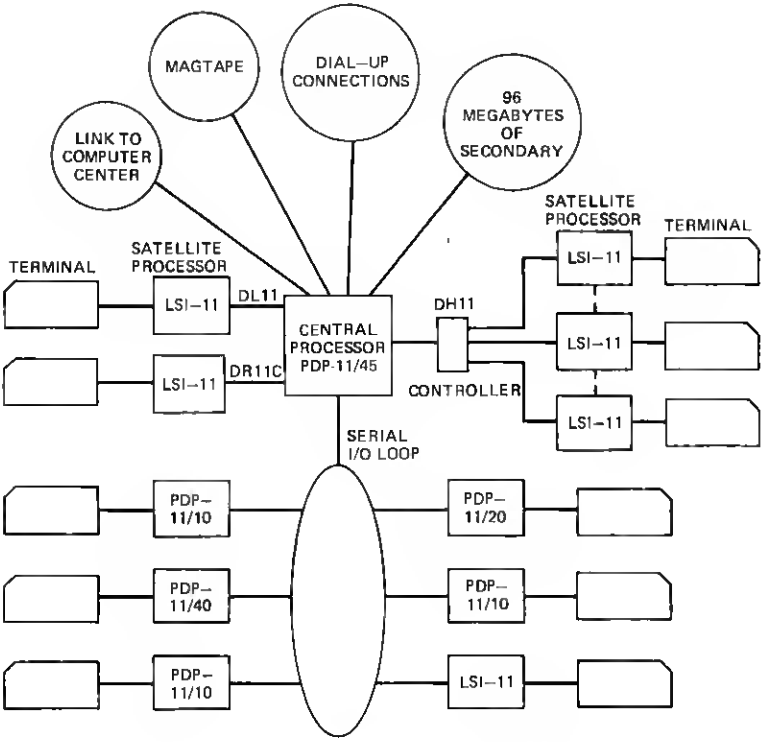

Fig. 1—Satellite processor hardware configuration.

line interface unit and the DH11 multiplexed asynchronous line interface unit. These are all essentially character-at-a-time transfer devices. The asynchronous line units may be run up to a baud rate of 9600. The most efficient communication link is the UNIBUS link device, which is a direct memory access device permitting a transfer rate of 100,000 words per second. However, the device limits the inter-processor distance to 150 feet. Another efficient link is the DR11C device, which permits word-at-a-time transfers. Its actual transfer rate is limited by software to about 10,000 words per second.

The choice of communication link is based on the distance between the SP and the CP, data transfer rate requirements, and the cost of the link. The I/O loop allows an SP to be placed at least 1000 feet from the CP and supports a data transfer rate of 3000 words per second. Thus, an SP with 16K words of memory can be loaded in 5 seconds.

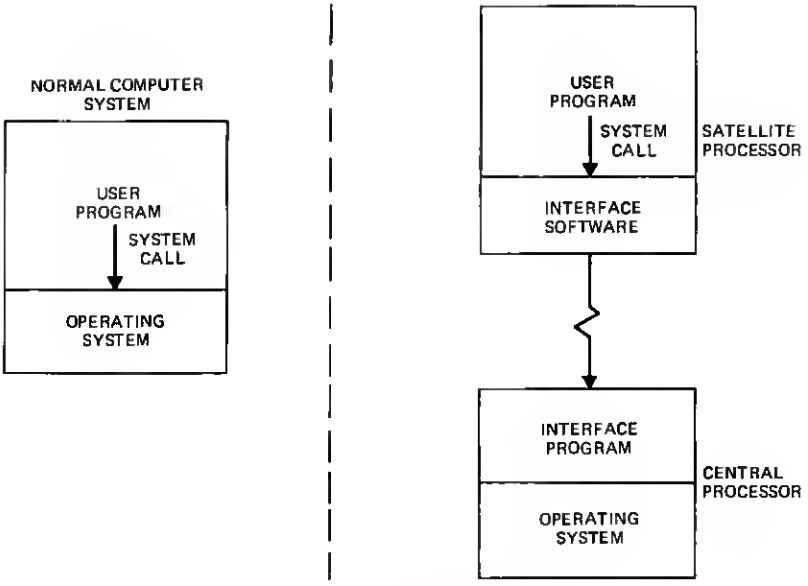

Fig. 2—Satellite processor concept.

IV. SP SOFTWARE

The satellite processor concept extends an operating system on a CP to multiple SPs. In an operating system such as the UNIX system, the interface or communication between a user program and the system is by means of the *system call*. These UNIX system calls manipulate the CP file system and other resources managed by the operating system. In the SP concept, the interface between a user program running in the SP and the operating system which is being emulated by the central processor is also the system call (see Fig. 2), except that here the extension is achieved by *trapping* the system call in the SP and passing the system call and its arguments to the CP. A process running in the CP on behalf of the SP then executes the system call and passes the results back to the SP. Control is then returned to the SP user program. Each SP executes a program locally, has access to the CP's file system and peripherals by means of the system call, and yet does not contain an operating system. This technique of partitioning a program at the UNIX system call level provides a clean, well-defined communication interface between the processors.

The local SP software required to support SPS consists of two small

functional modules, a communication package and a trap handler. The communication package transfers data between the SP and the CP on behalf of the program running in the SP. The trap handler catches processor traps (including system call traps) within the SP on behalf of the SP user program and determines whether to handle them locally or transmit the trap to the CP via the communication package.

4.1 SP communication package

The satellite processor communication package resides in the SP at the top of available memory and occupies less than 300 words. Actual size depends on the communication link used. The communication package normally resides in read-only memory. The functional requirements of the communication package include CP-SP link communication protocol, interpreting and executing CP commands, and sending trap conditions to the CP. The basic element of communication over a CP-SP link is an 8-bit byte, and messages from the CP to the SP are variable length strings of bytes containing commands and data. The SP communication package is able to distinguish commands from data by scanning for a special prefix byte. This prefix byte is followed by one of five command code bytes. Following is a list of the five commands and their arguments, which can be sent from the CP to the SP.

read memory	address	nbytes
write memory	address	nbytes
transfer	address	
return		
terminal i/o		

Each argument is two bytes (16 bits) and is sent twice, the second byte pair being the twos complement of the first to ensure error free transmission. Also, the data following the read memory and write memory commands have a checksum associated with them to guarantee proper transmission. If within the byte stream of data, a data byte corresponds to the command prefix, it is followed by an escape character to avoid treatment as a command.

This communication package is sufficient to enable the user at an SP terminal to communicate with the CP as a standard login terminal. When the SP communication package is started, it comes up in terminal I/O mode, passing all characters from the local SP terminal to

the CP over the communication link. In the reverse direction, all CP output is printed on the local SP terminal. The five communication commands listed above are only invoked when a program is downloaded and executed in the SP. The **read memory** and **write memory** commands are used to read and write the memory of the SP, respectively, starting at the specified address, **address** and continuing for **nbytes** bytes. The **transfer** command is used to force the SP to transfer to a specified address in the SP program, normally the beginning of the program. The **return** command is used to return control back to the SP at the address saved on the SP stack. When the CP wishes to write on or read from the local SP terminal, the SP is given the **terminal i/o** command.

4.2 SP trap handler

The second functional module which must be loaded into the SP is the trap handler. It is prepended to each program to be executed in the SP. This is the front-end package which must be link-edited with the object code produced by a UNIX compiler. The trap handler catches all SP traps and passes those that it cannot handle to the CP via the communication package. The trap handler determines the trap type (and, in the case of **system call** or SYS traps, the type of SYS trap). If the trap is an illegal instruction trap, the handler will determine if it has the capability to emulate this instruction, or whether it must be passed to the CP. If the trap is to be passed to the CP, a five-word communication area in the SP is filled with the state of the SP at the time of the trap. The communication package causes an interrupt to occur in the CP, thereby alerting the CP process running on behalf of the SP. The SP trap state is then read from the communication area and, upon processing this trap in the CP, the CP process passes argument(s) back in the communication area of the SP. Control is then returned to the SP.

The trap handler also monitors the SP program counter and local SP terminal 60 times a second using the 60-Hz clock in the satellite processor. This permits profiling a program running in the SP and controlling it from the local SP terminal. Upon detecting either a rubout character (**delete**) or a control backslash character (**quit**) from the local SP terminal, a signal is passed back to the CP, causing the SP program to abort if these signals are not handled by the SP process. At the same time a check is made to see if there have been any **delete** or **quit** signals from the CP process. If the SP has no local terminal, setting a -1 in the switch register will turn control

Table I

Instruction	PDP-11/20	PDP-11/45
mul (multiply)	830 μs	3.8 μs
div (divide)	1200	7.5
ash (shift)	660	1.5
ashc (double shift)	720	1.5
xor (exclusive or)	440	0.85
sob (sub. and branch)	400	0.85
sxt (sign extend)	400	0.85

over to the CP process. If an undebugged program in the SP halts, restarting it at location 2 will force an iot trap to the system trap handler, which in turn causes the memory of the SP to be dumped into a core file on the CP.

The trap handler consists of up to four separate submodules:

(*i*) Trap vectors, communication area, trap routines (400 words)
(*ii*) PDP-11/45 instruction emulation package (500 words)
(*iii*) Floating point instruction emulation package (1000 words)
(*iv*) Start-up routine.

Of these, the first is always required. The illegal instruction emulation packages are loaded from a library only if required. The start-up routine depends on the options specified by the user of the program to be loaded.

Estimates have been made of the execution time of the various emulation routines. The times are approximate and assume a PDP-11/20 SP, a PDP-11/45 CP, and an I/O loop connecting them.

The running times for the PDP-11/45 instructions emulated in the SP are shown in Table I. If execution time is important in a SP program, these instructions should be avoided. In C programs, these instructions are generated not only when explicit multiplies, divides, and multiple shifts are written, but also when referencing a structure in an array of structures. Using a PDP-11/35 or PDP-11/40 with a fixed point arithmetic unit as an SP would reduce the execution time for these instructions.

The average times to emulate floating point instructions in the SP are shown in Table II. For applications which require large

Table II

Instruction	PDP-11/20	PDP-11/45
add	2100 μs	4 μs
sub	2300	4
mul	3500	6
div	5600	8

quantities of CPU time running Fortran programs, it is possible to use a PDP-11/45 CPU with a floating point unit as an SP.

V. CP EMULATION OF TRAPS

During the time that the SP is executing a program, the associated CP process is roadblocked waiting for a trap signal from the SP. Upon receiving one, the CP process reads the SP trap state from the communication area, decodes the trap, and emulates it, returning results and/or errors. A check is also made to see if a signal (quit, delete, etc.) has been received.

Of the more than 40 UNIX system calls[5] emulated, about 30 are handled by simply passing the appropriate arguments from the SP to the CP process and invoking the corresponding system call in the CP. The other 10 system calls require more elaborate treatment. Their emulation is discussed in more detail here.

To emulate the signal system call, a table of signal registers is set aside in the CP process, one for each possible signal handled by the UNIX system. No system call is made by the CP process to handle this trap code. When a signal is received from the SP, this table is consulted to determine the appropriate action to take for the CP process. The SP program may itself catch the signals. If a signal is to cause a core dump, the entire SP memory is dumped into a CP core file with a header block suitable for the UNIX debugger.

The stty and gtty system calls are really not applicable to the SP process, but if one is executed, it will be applied to the CP process's control channel. The prof system call is emulated by transferring the four arguments to the profile buffer in the SP memory. Upon detecting nonzero entries here during each clock tick (60 times per second), the SP will collect statistics on the SP program's program counter. Upon completion of the SP program, data will be written out on the mon.out file. The sbrk system call causes the CP process to write out zeros in the SP memory to expand the bss area available to the program. An exit system call changes the communication mode between the SP and the CP back to the original terminal operation mode. It then causes the CP process to exit, giving the reason for the termination of the SP program.

The three most time-consuming system calls to emulate are read, write, and exec. The exec system call involves loading the executable file into the SP memory, zeroing out the data area in the SP memory, and setting up the arguments on the stack in the SP. A system read call involves reading from the appropriate file and then

transfering this data into the SP buffer. The system write call is just the reverse procedure.

The fork, wait and pipe system call emulations have not been written at this time and are trapped if executed in a SP. One possible means of emulating the fork call would be to copy an image of the parent process in one SP into another SP, permitting the piping of data between two SPS.

VI. TYPICAL SESSION

Supporting a mini-PDP-11 as an SP on a CP running the UNIX system combines all the advantages of the UNIX system programming support with the real-time response and economic advantage of a stand-alone PDP-11. In a typical SP programming session, a programmer sitting at the local SP terminal logs into the CP and uses the UNIX editor to update an SP program source file. It could be assembly language or one of the higher-level languages available on the UNIX system (C, Lil,* Fortran). Assume a C source file prog.c. When the edit is complete, the following commands are issued:

```
% cc −c prog.c
% ldm −me prog.o
% 11l a.out
```

cc -c compiles the C program prog.c in the CP and produces the object file prog.o. ldm -me combines the SP trap handler (-m) and instruction emulator (e) with the C object file prog.o, generating an a.out object file. 11l loads the a.out file into the SP, and starts it with the SP terminal as the standard input and output. The programmer then observes the results of running the program or forces a core dump, and uses the UNIX debugger to examine it. If any program changes are required, the preceding steps are repeated. During this typical SP support sequence, the programmer initiates the editing, compiling, loading, running, and debugging of a program on a mini-PDP-11 without leaving its control terminal. It is the speed and convenience of this procedure along with the availability of high-level languages that make the satellite processor concept a powerful mini-PDP-11 support tool.

* Lil is a little implementation language for the PDP-11.

VII. USES

Some SP's may be disconnected from the CP when their software has been developed and the final product is a "stand-alone" system. Other SPS may always have a CP connection; they supply the real-time response unavailable from the CP, combined with access to the CP's software base, file system, peripherals, and connection to the computing community.

One use of the SPS system is discussed in a paper in this issue.[6] Here LSI-11 microcomputers connected to a CP by means of a DH11 device are used in a materials research laboratory, remote from the CP, to collect data, control apparatus and machinery, and analyze the results.

One of the more interesting applications of the satellite processor system is its use to support a digital sound synthesizer system. The hardware consists of an LSI-11 processor with 24K words of memory, two floppy disks, a TV raster scan terminal, and much more special digital circuitry interfaced to the LSI-11 Q-bus to provide the control of the DSSS. The heart of the software consists of a multi-tasking system designed to handle about 100 processes.[7] The basic program directs the machine's output devices such as oscillators, filters, multipliers, and a reverberation unit. The data for the program are stored and retrieved from the floppy disk. The SPS is used to download programs from the CP and produce core dumps of the LSI-11 memory back at the CP for debugging purposes. Tbe CP is also used for program development.

VIII. SUMMARY

The advantages of the SPS system are the use of higher level languages, ease of program development and maintenance, use of debugging tools, interactive turn-around, use of a common pool of peripherals, access to files on the CP secondary storage, and connection to central computing facilities. The SP requires a minimum amount of memory since it does not contain an operating system or other supporting software. One additional advantage is that any SP may be located in a remote laboratory location.

The ability to extend an operating system to an SP may be used for purposes other than supporting software development for the SP. A new operating system environment may be defined by rewriting the CP process which acts on behalf of the SP program. In this way a new set of "system calls" emulating another operating system may

be extended to an SP. SPs other than PDP-11s may also be supported by writing an appropriate SP communication package and CP interface package. Cross compilers would be required on the CP to support software development for these non-PDP-11 processors.

Another avenue of research which has not yet been explored with the SPS concept is that of distributed computing. With a powerful SP, e.g. PDP-11/45, a compute-bound program could run on the SP rather than on the CP itself, thereby transferring the real-time load from the CP to the SP. The CP would only be called upon to load the program initially and to satisfy certain file requests. The total computing power of the system would increase greatly without duplicating the entire computer system.

REFERENCES

1. *DEC PDP-11 Processor Handbook.* 1975.
2. D. R. Weller, "A Loop Communication System for I/O to a Small Multi-User Computer," Proc. IEEE Intl. Computer Soc. Conf., Boston (September 1971).
3. D. M. Ritchie and K. Thompson, "The UNIX Time-Sharing System," B.S.T.J., this issue, pp. 1905-1929.
4. H. Lycklama and D. L. Bayer, "UNIX Time-Sharing System: The MERT Operating System," B.S.T.J., this issue, pp. 2049-2086.
5. K. Thompson and D. M. Ritchie, *UNIX Programmer's Manual,* Bell Laboratories, May 1975, sixth edition.
6. B. C. Wonsiewicz, A. R. Storm, and J. D. Sieber, "UNIX Time-Sharing System: Microcomputer Control of Apparatus, Machinery, and Experiments," B.S.T.J., this issue, pp. 2209-2232.
7. D. L. Bayer, "Real-Time Software for Digital Music Synthesizer," Proc. Second Intl. Conf. of Computer Music, San Diego (October 1977).