2

UCRL- 86132
PREPRINT
CONF-811040--11

DEBUGGING IN A MULTI-PROCESSOR ENVIRONMENT

James M. Spann

MASTER

This paper was prepared for submittal to
9th Symposium on Engineering Problems
of Fusion Research
Palmer House
Chicago, Illinois

October 26-29, 1981

September 29, 1981

Lawrence Livermore Laboratory

Unclassified

DEBUGGING IN A MULTI-PROCESSOR ENVIRONMENT*

James M. Spann
Lawrence Livermore National Laboratory
P. O. Box 5511, L-535
Livermore, CA 94550

Summary

The Supervisory and Control and Diagnostic System (SCDS) for the Mirror Fusion Test Facility (MFTF) consists of nine 32-bit minicomputers arranged in a tightly coupled distributed computer system utilizing a share memory as the data exchange medium. Debugging of more than one program in the multi-processor environment is a difficult process. This paper describes what new tools were developed and how the testing of software is performed in the SCDS for the MFTF project.

The Inter-Processor Communications System (IPCS) of SCDS coordinates tasks running on various computers. A task that runs under the IPCS is called an execute module (EM). Normally all computers are running simultaneously various EMs; EMs running on one computer may have to communicate with another computer. The EMs are the application control programs that require testing. Conventional debugging tools do not allow coordinated debugging of a set of EMs communicating across computer boundaries. The testing of EMs required modifications to the IPCS and development of a new debugging utility. Modifications to the IPCS allow the user to communicate with EMs directly via his/her terminal; this together with other existing software utilities were incorporated into a single debugging tool. The SCDS group has been using this debugging technique in a multi-processor environment for over a year.

*"Work performed by LLNL for USDOE under contract number W-7405-ENG-48"

Introduction

Software testing and debugging in a multiprocessor environment requires a new set of tools for the software engineer. The supervisory and control and diagnostics system (SCDS) for the Mirror Fusion Test Facility (MFTF) consists of nine 32-bit minicomputers arranged in a tightly coupled distributed computer system using a shared memory as the data exchange medium (Figs. 1 and 2). We developed a new debugging tool to aid in code testing in this environment. This debugging tool (called EmDeBug) allows a user from one terminal connected to one processor to debug and test multiple coordinated Pascal programs running on several of the processors in SCDS as though the user was dealing with only one computer and one program.

Each computer (Perkin-Elmer 7/32 or 8/32) in the network runs the vendor-supplied operating system OS-32/MT. A code running under OS-32/MT is called a task. The inter-processor communications system (IPCS) of SCDS coordinates tasks running on the various processors. A task running under IPCS is called an execute module (EM) and has attributes that are characteristic to EMs exclusively. One such attribute is that each EM has a mail box in which to receive mail from other EMs. EmDeBug (execute module debugger), described in this article, allows a user to control and to symbolically debug several EMs simultaneously from a single terminal regardless of on which physical processor a particular EM is executing.

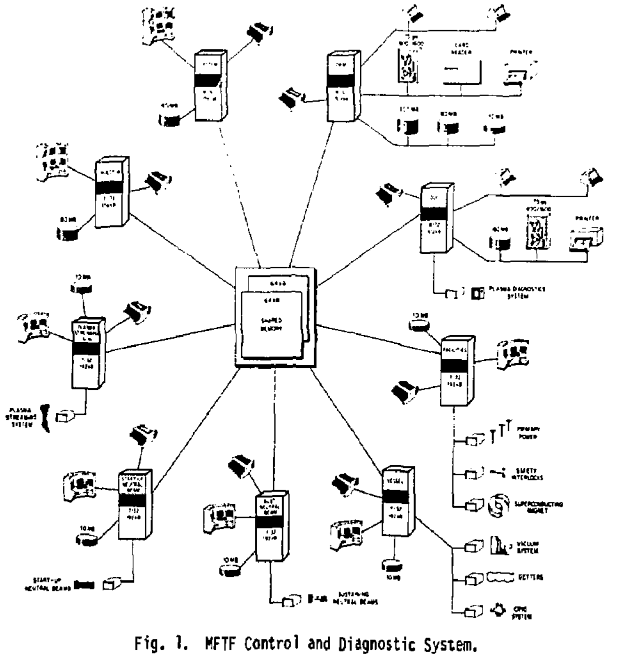

Fig. 1. MFTF Control and Diagnostic System.

[illegible]

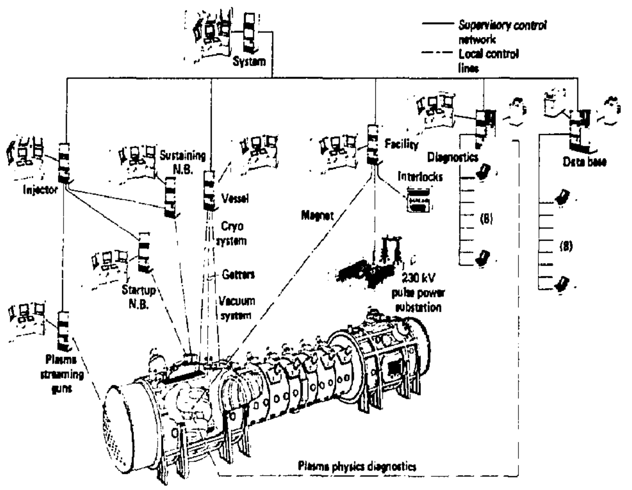

Fig. 2. Lines of Communication Between EmDeBug and EMs Under Test. The user is able to test EMs on many processors simultaneously.

Coordinated Debugging Difficult

Coordinated debugging of more than one program in the multiprocessor environment is an extremely difficult process. Before EmDeBug was developed for multiprocessor debugging, the programmer was required to have a terminal assigned to each processor that was running the debugger. This enabled program testing across processor boundaries. But the process was cumbersome and expensive, and made poor use of valuable resources (terminals) for which other programmers were competing. Even with this method, it was difficult to test two EMs at a time and next to impossible to test modules across three processors. The problem became unmanageable as more of the processors were brought on line and more programmers started testing. Thus, it became imperative for us to develop a high-level debugger that would enable programmers to efficiently test multimodule systems running on multiple processors.

Sum Of The Parts Equals The Solution

The facilities of IPCS--together with certain software utility programs--allowed us to construct a new software debugging tool. A symbolic debugger already existed for local debugging on a single processor. And various other software utilities programs existed for examining the messages (known as mail) sent between EM's. These and several other programs were put under the control of EmDeBug and comprise the EmDeBug system for multiprocessor debugging.

Symbolic Debugger

The Pascal/32 symbolic debugger facility allows an interactive user to examine the state of a Pascal program; to address and display active variables by their symbolic names; and to establish breakpoints for tracing and further state examination. This debugger was provided with the Pascal compiler as a debugging tool for a user debugging one program. An option in the Pascal compiler allows construction of a symbol table file that the debugger uses to address variables and procedures by name. When an EM is to be tested, the debugger is included as a procedure and linked into the EM code for local debugging. The debugger becomes active when a breakpoint, explained below, is executed.

The following is a partial list of the symbolic debugger commands:

- LIST provides a list of all the symbolic names which may be used in conjunction with a specified name.
- LOCATE returns the relative memory address of the specified symbolic name.
- DISPLAY causes the contents of the memory location specified by the symbolic name (variable, routine, or components) to be displayed.
- BREAKPOINT sets a dynamic breakpoint at a specified program line number or relative memory address. A breakpoint causes the program to interrupt execution and transfer control to the debugger when the code gets to the specified place. The program is then suspended until the user restarts it, at which time it continues from the breakpoint.
- TRACEBACK provides a listing of all of the active procedures which have been executed to date starting with the current procedure and backwards in time to the initial statement of the program.
- EXAMINE displays a memory location, specified either symbollically or as a relative memory address, in hexadecimal notation.
- REGISTERS displays the computer register and program counter contents for the process that caused the debugger to be entered.

To reiterate, all of the above commands are available after the program being debugged has executed a breakpoint that activates the debugger program.

What is the IPCS?

The inter-processor communications system (IPCS) is the distributed operating system that manages the communications between all the processors and the execute modules. EM's can only be started by other EM's. (Note that there is a way to get the first EM into the system.) When an EM sends mail to another EM, the IPCS queues the mail (puts it in the mailbox) and then loads and starts the recipient EM task if it is not already running. When the EM completes its job, it returns its mail through the IPCS to the sender, called the "parent" of the mail, and removes itself from memory thus completing the cycle.

Each EM task performs a well-defined job; it is loaded into memory, runs to completion and then is removed from memory. This is the normal operation of IPCS-handling system control type mail. Any errors generated during execution of an EM causes messages to be sent to the system operator. EmDeBug uses this feature to communicate with EMs across processor boundaries.

EmDeBug has the special ability to attach to an EM (on any processor) for testing. EmDeBug, itself is an EM, runs on the current processor and is activated by a user. When EmDeBug attaches to an EM, the IPCS re-routes the mail (messages) from the EM task to EmDeBug (Fig. 3). *The EM program on finding mail in its mail*-box determines if it is message from EmDeBug (known as EM commands) and takes appropriate action.

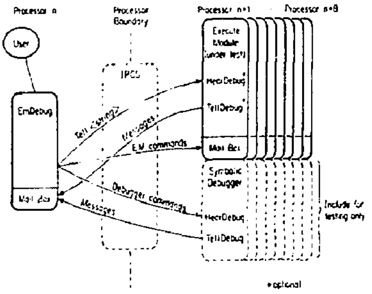

Fig. 3. EmDebug Communication Paths.

EmDeBug Commands

There are three major groups of EmDeBug commands: symbolic debugger commands, EM commands, and utilities commands.

Execute Module Commands

The EM commands and their interface with the IPCS make EmDeBug a debugger tool for a multiprocessor program. With EM commands, the user may start EM tasks, cancel running EMs, connect to and disconnect from EMs, and force an EM to break as if it had come to a breakpoint (which then activates the symbol debugger).

The TASK command causes the specified EM to be brought into memory and started. All error messages sent by the EM are then displayed on the user's terminal. If the EM specified by the TASK command to already in memory, EmDeBug will simply attach to it and not load it.

The TASK command will also let the user specify the file from which the execute module is to be loaded, and hence the user may load a surrogate execute module for testing and debugging rather than the one normally run. For example, an execute module surrogate that only prints out the contents of any mail received could be loaded and used for testing communication with other EM's. The use of surrogates is a fairly powerful testing and debugging technique. Further, generalized utility program stubs can now be written and widely used by programmers as surrogates for testing interfaces with other EMs.

The TELL command sends a text string message to an execute module via the IPCS. Communication between the EM under test and the user is provided by two procedures which are part of the EM. If an execute module under test wishes to send a message to the user running EmDeBug, the EM may do so by the procedure TellDbug (pronounced Tell Debug).

TellDbug sends the text string in the message to EmDeBug, *which then displays it for the user. Execute* Modules that wish to receive messages from the user running EmDeBug can do so via the procedure HearDbug (pronounced Hear Debug).

HearDbug receives the message initiated by the EmDeBug command TELL. The message (text string) so transmitted *could be used to select (user determined) branches* within the EM to aid in testing internal routines. These two procedures allow a user bidirectional communication with an EM and also with the debugger attached to the EM. (See Fig. 3.) TellDbug and HearDbug would be normally removed when testing is completed.

The DISCONNECT command detaches EmDeBug from an execute module. It leaves the EM in memory, but the EM will no longer receive messages (via HearDbug) from it; EmDeBug will also not receive any messages sent via TellDbug from the EM.

The CANCEL command causes the currently attached EM to cancel itself; i.e. terminate and leave memory.

The DEBUG command will force the currently attached EM to break as if it had executed a breakpoint and then enter the symbolic debugger.

Utility Commands

The utility commands are among the most useful commands available to the EmDeBug user. These commands provide general information concerning the activities and en- vironment of the processor(s) during a test session. The following is a list of the more interesting utility commands.

- ONRECORD logs all subsequent commands and messages to a disk log file. This provides a report of a debugging session that can be used as a reference later for a group discussion and documentation.

- COMMENT allows the user to enter comments into the log file.

- MAP displays a map of memory showing the current memory allocation of programs running on the specified processor.

- MTREE allows a user to examine the mail tree. Mail in the shared memory is linked into a tree for debugging purposes.

- TABLE displays important IPCS tables (processor status, logical-to-physical map, etc.).

- EXECUTE MODULE RESTART allows the user to specify that the attached EM should automatically reload and restart an if it goes end-of-job.
- TASK TABLE displays a list of the EMs that EmDeBug is currently attached to and can receive messages from. The table contains the EM name, load file name, mail id number and if execute-module-restart is active.

- BATCH processes a list of EmDeBug commands from a disk file. This makes it easier for the user to repeatedly set up a group of EM's for testing.

- EDIT activates a full feature editor. This lets the user inspect source files during a test session and make modifications.

Physical Requirements of EmDeBug

EmDeBug executes on the 32-bit minicomputers manufactured by Perkin-Elmer Corporation. EmDeBug was developed in the Pascal language provided by the University of Kansas, which also provided the symbolic debugger. The code size of EmDeBug is about 60K bytes in length. However all but 5K bytes of this program can be shared by more than one user; i.e. the first user requires 60K bytes to run EmDeBug, but each additional user only requires a 5K bytes of additional memory space. Several of the utility programs are execute modules; these are brought into memory and run and then removed from memory when completed to keep the memory requirements as small as possible.

Conclusions

Symbolic debugging software tools have been around since the late 1950s. They have proved so useful that they are now routinely available at most computer facilities. With the advent of multiprocessor systems, where one task may invoke other tasks running on other processors, the traditional debugging tools lose effectiveness.

What we have presented in this article is an enhancement of an already existing tool to allow debugging in the traditional way in a multiprocessor environment. The tool makes heavy use of features of the Inter-Processor Communications System that is installed on the system.

The work presented in this paper is probably not very transportable because of the heavy use of features of the IPCS, which is also a home-grown system. However, it is our hope that the outline presented here will be of use to other groups planning multiprocessor systems.

We believe this work is original and hope that others contemplating building multiprocessor systems will recognize the importance of designing the communication system to support multiprocessor debugging functions.

Bibliography

1. R. L. Glass, "Real-Time: The 'Lost World of Software Debugging and Testing'," Communications of the ACM, May 1980.

2. P. McGoldrick, "IPCS User's Manual," SCDS System Software Manual Section 6.1.2, Internal LLNL Document.

3. J. Spann and P. McGoldrick, "Execute Module Debugger," SCDS System Software Manual, Section 6.1.3.5.6, Internal LLNL Document

4. R. Young, Pascal/32 Symbolic Debugger, an informal Kansas State University report, 1978.

5. R. Young and V. Wallentine, Pascal/32 Language Definition, an informal Kansas State University report, 1978.