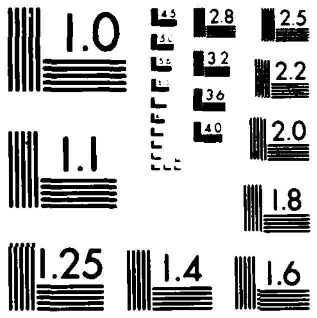
AD-A127 117 DEBUGGING TECHNIQUES FOR COMMUNICATING LOOSELY-COUPLED PROCESSES(U) ROCHESTER UNIV NY DEPT OF COMPUTER SCIENCE E T SMITH DEC 81 TR-100 N00014-78-C-0164 1/2

UNCLASSIFIED F/G 9/2 NL


MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

(12)

AD A127117

DTIC FILE COPY

rochester

Department of Computer Science
University of Rochester
Rochester, New York 14627

Debugging Techniques for Communicating, Loosely-Coupled Processes

by

Edward Tucker Smith

TR 100
December, 1981

This report reproduces a dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Computer Science at the University of Rochester.

The preparation of this report was supported in part by the Defense Advanced Research Projects Agency, monitored by the ONR, under contract number N00014-78-C-0164.

Abstract

This thesis describes work done on debugging techniques and tools for communicating, loosely-coupled processes. Our work is intended to reduce the apparent complexity of large systems of communicating programs by regarding only the interprocess activities of such programs. The use of multiple, communicating processes as a model of computation allows for a very clean "cut" of what information is interesting for debugging and what is not. Our approach to debugging is to provide the user with information about how sets of these processes behave rather than what each program associated with each process does.

Our tools provide various primitives for manipulating the interprocess activities of processes. We provide nothing to access the source code of any program. Our tools include a debugger program, a mechanism to fire and execute interprocess debugging demons and the ability to obtain transcripts of interprocess activities. The debugger provides commands for the user at a terminal for creating and manipulating individual interprocess events. Demons are an event-driven mechanism used to automatically monitor and modify interprocess events. Transcripts provide a record of interprocess events that can be replayed later.

Our debugging techniques make use of these tools to provide individual process control, communication monitoring and process testing. Process control includes the ability to create, suspend and destroy processes as well as the ability to obtain various process-related information. The communication monitoring facility monitors message traffic and can dynamically alter the contents of these messages. Process testing allows a user to isolate a process (or simulate a process) by creating and intercepting all message traffic in and out of a process.

This thesis also describes a debugging system called SPIDER that was built to demonstrate the above debugging tools and techniques. A multi-process kernel is included in SPIDER that supports communicating, loosely-coupled processes. SPIDER also includes an implementation of each of our proposed tools: a debugger program, a mechanism for firing demons and a transcriber. Several examples using SPIDER are given to show how our debugging techniques can be achieved with our debugging tools.

Table of Contents

1. Introduction.	1
1.1. The Problem	1
1.2. Debugging Programs.	2
1.3. Debugging Multiple Processes.	5
1.4. Our Goals for a Solution.	8
2. The Multi-process Model.	13
2.1. Introduction.	13
2.2. Interprocess Objects.	13
2.3. Interprocess Events.	17
2.3.1. System Calls.	18
2.3.2. Message Transmission.	20
3. Interprocess Debugging Tools and Techniques.	32
3.1. Introduction.	32
3.2. Debugging Tools.	32
3.2.1. The Debugger.	33
3.2.2. Interprocess Debugging Demons.	37
3.2.3. The Transcriber.	44
3.3. Debugging Techniques.	44
4. The Interprocess Kernel.	46
4.1. Introduction.	46
4.2. Basic Function of the Kernel.	46
4.3. Modifications for the Debugger.	47
4.4. Modifications for Demons.	48
4.5. Modifications for the Transcriber.	51
5. A Demonstration of Interprocess Debugging.	55
5.1. Introduction.	55
5.2. The Implementation of SPIDER.	55
5.3. An Example: Process Control.	57
5.4. An Example: Process Monitoring.	62
5.5. An Example: Process Testing.	65
6. Conclusions.	69
6.1. Contributions and Conclusions.	69
6.2. Future Work.	62
Bibliography.	74
A. User's Manual for SPIDER.	76
A.1. Introduction.	76
A.2. Debugger Commands.	76
A.2.1. Processes.	77
A.2.2. Ports.	78
A.2.3. Messages.	79
A.2.4. Programs.	80
A.2.5. Transcripts.	80
A.2.6. Demons.	81
A.2.7. Miscellaneous Commands.	81
A.3. System Calls.	82
A.3.1. Processes.	82
A.3.2. Ports.	83

A.3.3. Messages. 84
A.3.4. Miscellaneous Procedures. 85
A.4. Demons. 85
A.4.1. Demon Trigger Procedures. 86
A.4.2. Demon Commands. 87
B. Event Types. 88
B.1. Introduction. 88
B.2. Debugger Commands. 88
B.3. System Calls. 88
B.4. Message Transmission. 90

List of Figures

1.1. Two communicating processes. 9
1.2. A (modified) sequence of events. 10
1.3. Simple process simulation. 11
2.1. A process. 16
2.2. An event. 18
2.3. Message transmission: sending a message. 21
2.4. Message transmission: receiving a message. 22
2.5. "Send" event. 23
2.6. "CrossingOutGoingProgramBorder" event. 24
2.7. "CrossingOutGoingProcessBorder" event. 25
2.8. "CrossingInComingProcessBorder" event. 26
2.9. "CrossingInComingPortBorder" event. 27
2.10. "Receive" event. 28
2.11. "ReceiveWait" event. 29
2.12. "CrossingOutGoingPortBorder" event. 30
2.13. "CrossingInComingProgramBorder" event. 31
3.1. The program for a process. 34
3.2. A user session, with single-stepping. 35
3.3. Sending a message, with single-stepping. 36
3.4. A demon. 38
3.5. FireOnPort.demon. 41
3.6. FireOnMessage.demon. 41
3.7. ReplaceSend.demon. 42
3.8. How the demons fire. 43
4.1. The basic kernel. 47
4.2. Kernel modified to support the debugger. 48
4.3. Kernel modified to support demons. 49
4.4. Kernel modified to support the transcriber. 53
5.1. Communication paths among our Unix processes. 57
5.2. Creating a process with the debugger. 58
5.3. A process interaction. 59
5.4. Another Receiver. 60
5.5. Single-stepping Receiver. 61
5.6. MonitorBoth.demon. 62
5.7. Monitoring messages with the debugger. 63
5.8. Another monitor session. 64
5.9. WatchForReceiveWait.demon. 65
5.10. Bring up Receiver2. 66
5.11.a. A final debugger session. 67
5.11.b. A final debugger session (cont.). 68

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER TR 100	2. GOVT ACCESSION NO. AD A127117	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Debugging Techniques for Communicating, Loosely-Coupled Processes		5. TYPE OF REPORT & PERIOD COVERED Technical Report
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Edward Tucker Smith		8. CONTRACT OR GRANT NUMBER(s) N00014-78-C-0164.
9. PERFORMING ORGANIZATION NAME AND ADDRESS Computer Science Department University of Rochester Rochester, New York 14627		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Advanced Research Projects Agency 1400 Wilson Blvd. Arlington, Virginia 22209		12. REPORT DATE December, 1981
		13. NUMBER OF PAGES 90
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) Office of Naval Research Information Systems Arlington, Virginia 22217		15. SECURITY CLASS. (of this report) unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Distribution of this document is unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

none

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Communicating processes	process control
loosely-coupled processes	process testing
debugging techniques	communication monitoring
interprocess communication	

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This thesis describes work done on debugging techniques and tools for communicating, loosely-coupled processes. Our work is intended to reduce the apparent complexity of large systems of communicating programs by regarding only the interprocess activities of such programs. The use of multiple, communicating processes as a model of computation allows for a very clean "cut" of what information is interesting for debugging and what is not. Our approach to debugging is to provide the user with information about how sets of these processes behave rather than what each program associated with each process does.

Chapter 1
Introduction

1.1. The Problem.

Almost without exception, anyone who has learned how to write and execute computer programs has learned something about debugging them. We believe the presence of errors in computer programs has an alarming acceptance among programmers and users alike. To this day, software released by a certain large software house is delivered with a disclaimer that if they always waited to find and fix "the last bug" they would never release any software.

Much attention is now being paid to debugging systems and to their impact on the development of software in various programming environments. In particular, work is being done on development systems that enable a human user to create and maintain large, complex systems of programs. Not surprisingly, as user's programs are becoming increasingly larger and more complex, the size of these development systems is also growing to the extent that the development systems have themselves become large and complex systems of programs.

This thesis describes work done on debugging techniques and tools for communicating, loosely-coupled processes. Communicating, loosely-coupled processes are separate computer programs that communicate with each other by sending and receiving messages, but otherwise do not share memory (such as variables). These communicating processes serve as a mechanism for organizing large, complex systems of programs. With our techniques, we intend to reduce the apparent complexity of such systems by using debugging tools that regard only the interprocess activities of such programs.

The use of multiple, communicating processes as a model of computation allows for a very clean "cut" of what information is interesting for debugging and what is not. For the purposes of interprocess debugging, we will treat processes as separate, communicating "black boxes". We claim that debugging the individual programs associated with processes does not allow the user to cross process boundaries and handle interprocess activities. Giving the user a debugger for each source program of each process will fail to suppress irrelevant details and will swamp him with information about the internal states of each process. Instead, our approach is to provide the user with information about how these sets of processes behave rather than what each program associated with each process does.

1.2. Debugging Programs.

The following discussion reviews some systems for debugging single programs with respect to the debugging tools provided and the language used to communicate with the user. In particular, we are looking at systems that attempt to maintain a close relationship between the debugging environment and the user's conceptual view of the solutions to his problems. This relationship is the primary motivation for the design of our debugging tools for the interprocess domain. The following is not meant to be a complete history of all debuggers and is of course limited by the author's personal knowledge. [Satterthwaite75] contains an excellent history of debugging systems for the interested reader.

Recent debugging research is concerned with on-line or interactive debuggers and with experimentation on more powerful debugging interfaces. Such work describes debugging languages that are very similar to the conceptual language used to solve problems by the user. [Myers80] describes a system called Incense that provides an interactive, graphical display of a program's data structures. Incense obtains information about the state of these structures from the running program. The user indicates which structures he wants to see and Incense responds with an appropriate display. The design of the displays used in Incense is intended to parallel the user's view of what he believes his structures look like. This display includes the use of visual analogues to represent structures (such as boxes and arrows for linked list structures) and the use of language constructs (such as variable and enumeration type names) where appropriate. Although Incense allows interactive control over how structures get displayed, it is just a display program and does not allow interactive alterations of the structures.

Another graphical approach is the Pygmalion system in [Smith75]. A language is provided for both viewing and manipulating hierarchical picture elements called icons. Icons appear to be nested and linked boxes on the user's terminal screen. Graphical manipulation by the user on icons causes Pygmalion to create Lisp programs that perform the same function on Lisp structures. Running an icon program causes the associated Lisp program to run and the resulting changes to data structures in the Lisp code are mapped onto the screen as changes to icons. This system makes a significant step towards creating, executing and debugging a program in the same conceptual language used to solve problems by the programmer. In particular, the "boxes and arrows" representation of Pygmalion for nested and linked icons and the iconic operations corresponds to the user's imaginary view of such structures.

Another system that provides graphical input of programs is ThingLab, described in [Borning79]. ThingLab is an object-oriented system for building constraint-based simulations. The language of ThingLab allows the user to program a simulation using a graphical representation of objects and their various constraints. Values of objects and their constraints can be changed dynamically using the same graphical capabilities used to build the simulation. Though less general than Pygmalion in its programming abilities, ThingLab provides a variety of graphical displays (such as bar graphs, arithmetic operations and constant and variable constraint values) oriented toward a large class of simulation problems. With these the user is able to create and watch the simulations of certain problems in nearly the identical way the actual problems appear.

The Cornell Program Synthesizer, described in [Teitelbaum79], is a noteworthy system combining a syntax-driven editor, program interpreter and debugger in one programming environment. This system provides all control over the editing, execution and debugging of a program in one source language. The system has a full-screen, derivation-tree program editor that does not allow the user to write a syntactically illegal program. Compilation, in fact, is performed automatically at certain times during program editing. Debugging is combined with program execution using various visual aids. These aids include a full-screen display of the executing source text (with an indication of the currently executing statement), a separate screen area devoted to display of current variable values and another screen area for program output. All of these features allow the user to stay completely in the same source language, whether editing, executing or debugging his programs. Of course, the user is limited to solving problems in the programming language provided rather than a language that represents his problems more closely.

No description of such program development systems is complete without including the various Lisp systems. In terms of debugging, editing and otherwise manipulating all objects of a program in the same language, Lisp excels. [Sandewall78] is an excellent description of the powerful, interactive nature of the Lisp environment. However, even with this type of control, it is still easy to create systems in Lisp too large and complex to comprehend easily. Again, the user must think of his problems, the problem's representation and solution's representation completely in the programming language of the system.

[Masinter80] describes Scope, one of the Lisp support systems of Interlisp. Scope maintains a current data base of information about the user's program. This data base is updated appropriately when the program is edited and as the program executes. Scope answers questions posed by the user about the state of the program by querying this data base. Scope can answer questions dealing with the static form of the program (like the location of procedure definitions or procedure calls), questions dealing with the dynamic execution environment of the program (like the names of all currently executing procedures) or questions related to both (like the location of the call of a particular executing procedure). This tool provides a type of information filtering for the user while debugging. All potentially relevant information is gathered by the system and delivered to the user when he asks for it.

[Model79] describes higher-level and more automatic techniques for monitoring complex systems. This work introduces the concept of "meta-monitoring", which is the selective reporting of high-level information about program execution. The system uses knowledge about the significance of various types of program information to identify the occurrence of important events and suppress irrelevant details. Display of information about such events to the user is made with visual analogies to the computational states of the program. With all this, the user is provided with an interpretation of the activities of a complex system that is more relevant, informative and comprehensible than provided by the raw information alone.

[Winograd75] describes how very large systems (Lisp systems in particular) grow until they reach a point he calls the "complexity barrier". Once this point is reached, the system developers can no longer easily make any significant changes since the system has become too complex for anyone to comprehend how a particular change will affect other parts of the system. Winograd claims that complex systems are not hard because of any "intrinsic difficulty of the tasks the system must carry out". Instead, the problem is the difficulty of managing all the various parts of a complex system. Current programming environments, no matter how knowledgeable they are about the programs they help to write, still rely on the user to perform many tedious, bookkeeping tasks. Winograd suggests ways that the apparent complexity of such systems can be reduced based on techniques found in artificial intelligence systems.

These systems share several common characteristics in their solutions to the debugging problems of single programs. The systems all maintain consistency with a language, either the programming language at hand or with the conceptual language used to model the problem. The systems all provide for the compression of information, either through filtering based on what the user wants to see or through a visual analogy to a computational state. The systems all monitor programs dynamically, either keeping track of changing information as the programs run or providing for runtime analysis so all potentially relevant information is recorded. The systems all provide for the testing of parts of programs without involving the entire program. We intend to achieve similar characteristics with our debugging techniques for the interprocess domain. In particular, we intend to provide

- a language for describing objects and operations in the interprocess domain,

- mechanisms for filtering information about processes and their behavior with each other,

- dynamic information and control over processes as they execute and

- mechanisms for testing of individual processes before integrating into a system of processes.

Our work is derived from two different research directions: debugging systems for single programs and debugging systems for multiple process systems. The following section describes the multi-process direction and is followed by our goals for solving the problem.

1.3. Debugging Multiple Processes.

The following reviews some multiple process work with respect to the debugging facilities they provide. As with the single program systems described above, we are concerned here with the relationship between the debugging environment and the user's conceptual view of his programs' activities. In particular, we are looking at these systems for the tools and techniques provided to debug the multi-process environment. This is not intended to be a complete description of all multi-process systems. [Lantz80] contains an exhaustive history of multi-process, message-based systems for the interested reader.

The following describes how communication monitoring can be achieved in a multi-process environment. [Bobrow72] describes a facility for monitoring process activities in the Tenex operating system. This system provides an "invisible" debugger that is protected from program faults by residing in another process.

[Balzer73] makes a reference to a similar communication monitor implemented by reconnecting the communication ports of processes to a debugger process without the knowledge of the other processes. Balzer also notes that "nonexisting parts of a system can be simulated at a terminal by a user who dynamically supplies the needed data." [Baskett77] describes the communicating, multi-process features of Demos. Demos provides processes, called "tasks", with one-way message paths, called "links". Some links are provided with each process initially, including a link to a "Switchboard" task that allows a process to obtain certain public links. Processes can also obtain links from other processes in messages. One debugging feature described is a communication's monitor for Demos processes. This monitor is inserted between two communicating processes by manipulating their links. Note that these suggest debugging techniques for working strictly outside the domain of the programs running the processes themselves. This interprocess debugging provides information about how the processes behave by analyzing their communication.

[Rashid80] describes the Interprocess Communication Facility (IPC), a model for a message-based, multi-processing system. The IPC model provides communication ports for sending and receiving messages and a message format that is machine-independent. An IPC-specific debugger called Black Flag is described in [Philips81]. A modification to the operating system is made that allows Black Flag to interpose itself between a sending process and its destination port. A user of Black Flag can monitor and even manipulate the message traffic along this route by changing the contents of messages. However, Black Flag provides neither control over the processes nor any automatic filtering of messages.

[Feldman79] describes the distributed computing programming language Plits. Processes in Plits, called "modules", are distributed on distant machines and share no information directly. Modules in Plits communicate using messages. A Plits message is composed of a set of slots, each slot containing a name/value pair. Slots can be individually added and removed from messages. For a module to access a particular slot in a message, the slot name must be declared in the module and must exist in the message. Modules can thus manipulate messages that contain slots with name/value pairs unknown to it. In particular, this allows a monitor or a debugger to intervene and modify information in messages in transit without the knowledge of the communicating processes.

[Gertner80] presents a system for evaluating the performance of communicating processes. This work describes a language for modeling process behavior with composite finite state machines to describe multiple process state and message traffic. Various mechanisms are introduced to reduce the number of possible interprocess state changes for the performance evaluator. This system essentially provides a relatively sophisticated message traffic monitor. However, since it is not intended for use as a true process debugger, no features are provided to control or manipulate processes. In fact, special care is taken to ensure that the monitor does not interfere in the operations of the system being monitored.

The following systems provide features for debugging individual processes in a multi-process environment. [Reiser75] describes Bail, a debugger capable of accessing the multiple process environment of a Sail program. Bail allows the user to insert breakpoints and access variables in procedure instances within process instances of a running program. [Ball75] describes the RIG message-based operating system. RIG is a large, message-based, multi-process operating system with no shared variables between processes. Debugging in the RIG system is accomplished using a multi-process debugging program capable of accessing the (compiled) code of all processes (from [Ball78]). The debugger provides the usual assortment of debugging features on a per process basis, such as the reading and writing of code and data and the setting of breakpoints in individual processes. However, in neither Bail nor the RIG debugger are any interprocess features available: the debuggers cannot monitor message traffic, cannot start or stop processes and cannot send or receive messages.

Some of the techniques used in multi-process debugging are similar to those provided by single program debuggers since the debugging problems are similar. For example, a user with multiple processes needs to monitor messages and their contents just as a user with a single program needs to monitor procedure calls, the arguments of the calls and their results. Other techniques are unique to the multi-processing domain: for example, a user can take advantage of the multiple loci of control in the multi-process domain to specify the order in which suspended processes are to be resumed. In general, we claim that debuggers of multi-process systems should be able to deal with the objects and activities of the multi-process domain as program debuggers deal with the programming language constructs of code and data. We also claim multi-process debuggers should provide other features appropriate to communicating multi-process domain. The above systems provide for some of these features but do not take as general an approach to the solution as we will here. The next section gives our goals for this work and introduces our approach to solving these problems.

1.4. Our Goals for a Solution.

Our primary goal with this work is to provide tools and techniques for debugging a message-based, multi-processing environment. In particular, we claim such a debugging system should provide:

- individual process control,

- communication monitoring facilities,

- process testing features.

Process control should include the ability to create, suspend and destroy processes as well as the ability to obtain various process-related information available in the system (such as length of message queues, current process state, etc.). The communication monitoring facility should be able to monitor message traffic, (automatically or under the control of the user), and dynamically alter the contents of these messages. Process testing features should allow a user to isolate a process (by controlling the flow of messages in and out of the process) and simulate a process by creating message traffic dynamically.

Our approach to providing such features involves the use of mechanisms based on the idea of an <u>interprocess event</u>. Each such event represents a request to perform some interprocess activity, (in the same sense that a procedure call in a program is a "request" to execute that procedure). Our intention is to provide tools that can monitor and manipulate such events to achieve these abilities.

We give here a simplified example of what we mean by an interprocess event and how we plan to debug processes by controlling events. Figure 1.1 below shows two communicating processes. This figure is labelled to show three events, the sending of a message by Process1, the arrival of the message at Process2 and the receipt of the message by Process2.

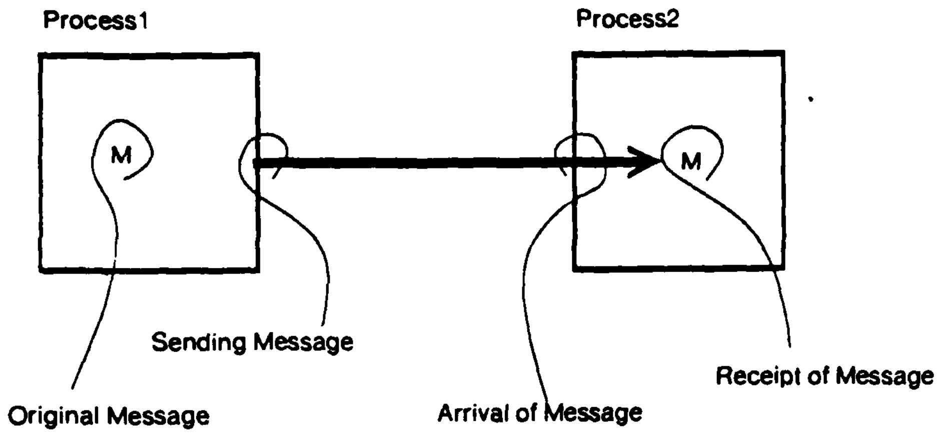


Figure 1.1. Two communicating processes.

Figure 1.2 shows these events and indicates where other events could be inserted into this sequence to provide a simple monitoring of this message traffic.

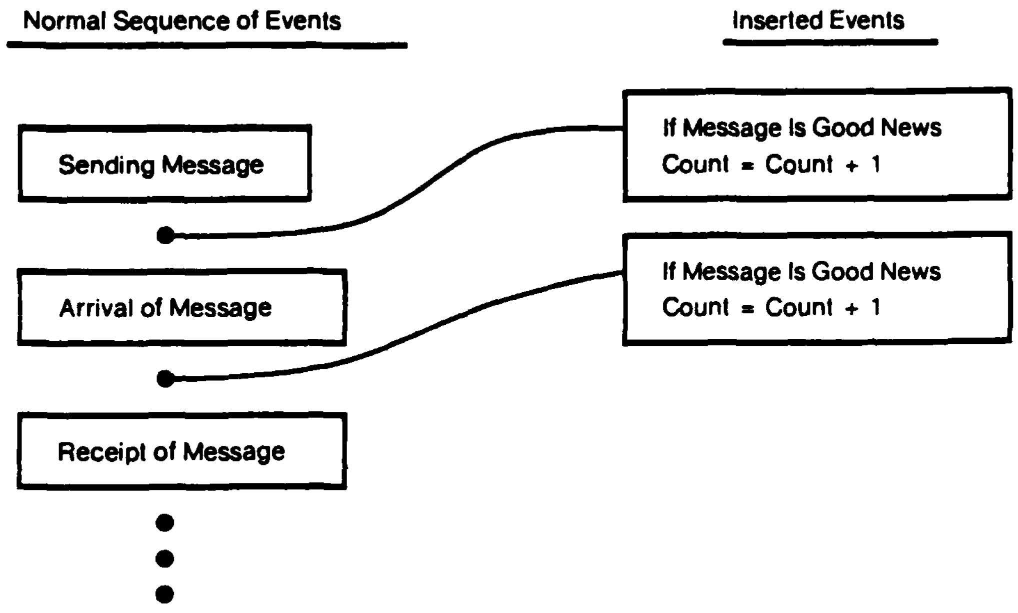


Figure 1.2. A (modified) sequence of events.

Figure 1.3 shows how Process1 could even be simulated by inserting the message-arrival event into this sequence. Note that the monitoring event would not record that any process ever sent a message, just that a process received one.

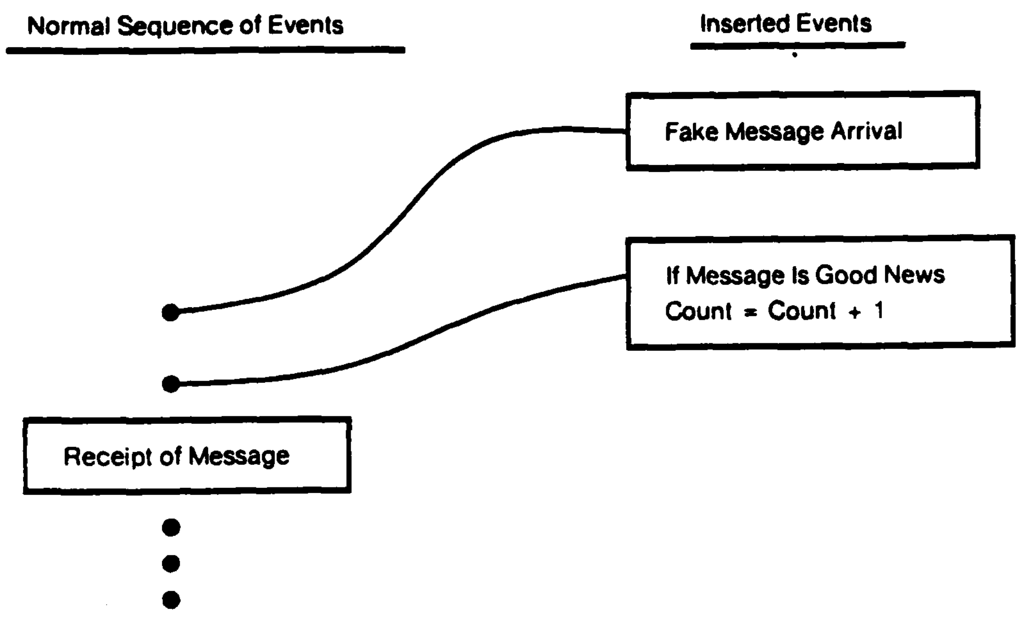


Figure 1.3. Simple process simulation.

Our debugging tools will provide us with the ability to manipulate these events. These tools consist of the following:

- an interprocess debugger,

- interprocess debugging demons and

- a transcriber of interprocess events on a per process basis.

The debugger provides the user with a debugging language for manipulating interprocess events. Debugging demons are an event-driven component for monitoring and manipulating events automatically. The transcriber provides for the recording of the event sequence associated with a process that can be replayed later for analysis.

In the following chapter (Chapter 2), we define our model for a communicating, multi-process system. This includes definitions for the interprocess objects and events used to support communicating, loosely-coupled processes.

In Chapter 3, we define our debugging tools: the debugger, the debugging demons and the transcriber. We also define the interprocess debugging language used for manipulating the objects and events of our model.

In Chapter 4, we describe the interprocess kernel. This kernel implements the objects and events of the interprocess model and implements our debugging tools.

Chapter 5 describes SPIDER, a program created to demonstrate our debugging techniques. SPIDER implements the objects and events of the interprocess model, as well as all our debugging tools. This chapter shows the use of SPIDER to monitor, control and test processes.

Chapter 6 concludes and discusses directions for future work.

An appendix is included at the end of this work that describes in much detail the environment of the SPIDER demonstration program. This includes the commands available to the user of the debugger, various communication and multi-process procedures available to a programmer and primitives used to create event-driven demons.

Another appendix is included that gives a complete description of all types of interprocess events used by our multi-process model.

Chapter 2
The Multi-process Model

2.1. Introduction.

In this chapter, we define our multi-processing model. This model includes all the objects and operations necessary to support communicating, loosely-coupled processes. We refer to the objects of this model as interprocess objects and describe them in section 2.2 below. We refer to the operations performed on interprocess objects as interprocess events and describe them in section 2.3 below.

The _kernel_ is the program that implements all our interprocess objects and the operations on them. Chapter 4 discusses this kernel in detail. The kernel provides system routines that can be called by programs to cause interprocess operations to occur. These system calls are described in section 2.3.1 below.

2.2. Interprocess Objects.

In this section, we define the interprocess objects of this model. Each object has a _reference_ used to name the object. Object references may be used as arguments in system calls and may be returned as results from these calls. Object references may also be sent in messages to processes. We will make some restrictions on how a process may use object references received from other processes. These restrictions are described in section 2.3.1 below. Note that all _borders_ mentioned in this section are defined in section 2.3.2 below.

A _process_ is an object composed of zero or more ports, an event execution parameter, a program, a program state, zero or more process state variables and a process border. All of these components are considered _local_ to the process. Components of other processes are considered _foreign_. Each of these components is defined below.

A port is an object composed of a queue of messages and a port border. Each port is a component of exactly one process. Messages are sent to ports (rather than to processes). Messages are received from ports. In particular, only the process that contains a particular port can receive its messages. A process can send messages to any port it has a reference to. Messages have the following partial ordering: all messages sent from a particular process to a particular port will arrive in the order sent. No other ordering occurs.

A name service is available for ports that allows processes to obtain port references of foreign ports using commonly known names. The scope of these names is global to all processes. This service is accessed through system calls.

The event execution parameter of a process has either the value "Run" or "Suspend". The value of this parameter specifies whether interprocess events will get executed (Run) or will not get executed (Suspend) for the process. (Interprocess events are defined in section 2.3 below.) The event execution parameter of a process is accessible only by the debugger cannot be modified by the process. The value of this parameter is not to be confused with the execution state of the process's program.

The program of a process is composed of a program border and a single address space containing code and data. The address space of this program exists on a machine whose characteristics are left unspecified. No data is shared between the program in one process and any other.

The program state of a process is one of NoCode, Running, Dead, ReceiveWait, or SystemCall. This state reflects the following about the program:

- NoCode, no code is running as the process's program,

- Running, code is running as the process's program and is currently executing anything but a system call,

- Dead, the code previously running as the program has died,

- ReceiveWait, code is running as the process's program, has called the system routine to receive a message from a port and is waiting for a message to be returned, or

- SystemCall, code is running as the process's program and has made a system call.

A message is represented by a parenthesized list of zero or more name/value pairs. A name/value pair is represented by a parenthesized list containing an alphanumeric string as its first component and either another alphanumeric string or a parenthesized list as its second component. For example,

```
( MessageIs Beware )
```

and

```
( Lunch ( Sandwich Peach ) )
```

are both name/value pairs and

```
( ( MessageIs Beware ) ( Lunch ( Sandwich Peach ) ) )
```

is a message containing both name/value pairs. Messages are objects local to programs. When a message is sent from one process to another, a copy of the message is made.

A process state variable is a name/value pair. These variables are created and accessed by programs with system calls. These variables allow a program to record interesting states or information about the process for the user of the debugger.

Figure 2.1 below shows a process as we picture it with all of its components.

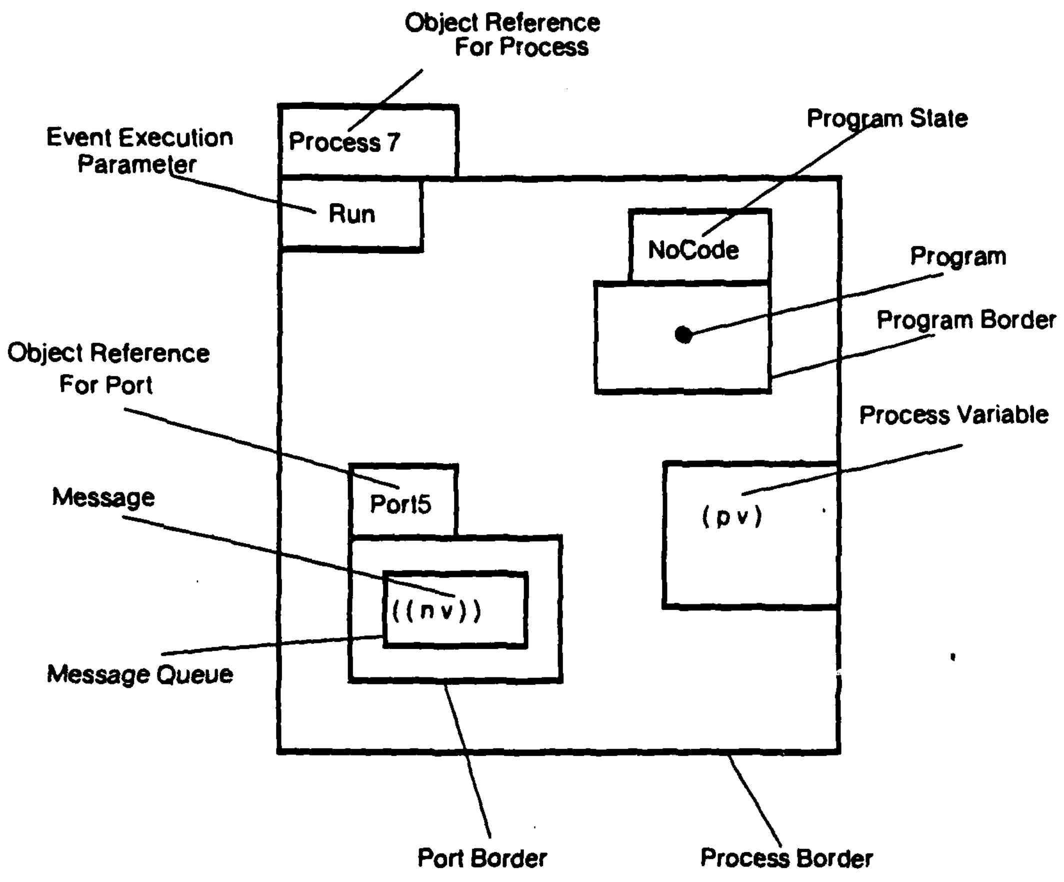


Figure 2.1. A process.

2.3. Interprocess Events.

This section describes the interprocess events of our model. These events represent the operations on interprocess objects provided by the kernel to support communicating, loosely-coupled processes.

An interprocess event is composed of an event class, an event type, zero or more event parameters and a possibly empty process reference. An event is created by the execution of a system call by a process's program, by the user of the debugger giving a command or as a side-effect of the execution of other events. Events are requests to perform some activity by the kernel. Events are executed by the kernel. After execution, each event is destroyed. Again, Chapter 4 describes how the kernel executes events in detail.

An event is composed of all information necessary to perform some activity. The event type is the name of the activity. The event parameters contain information dependent on the event type. We will define event types and their parameters as we need them in this discussion. A complete description of all event types and their parameters is contained in Appendix B.

The event class has the value "SystemCall", "MessageTransmission" or "DebuggingCommand". The class of an event specifies whether the event is a system call executed by the program of a process ("SystemCall"), a message being transmitted to a port ("MessageTransmission") or a command from the user of the debugger ("DebuggingCommand"). System call events are described in the next section. Message transmission events are described in section 2.3.2 below. Debugging events are defined in the next chapter.

As an example of an event, in Figure 2.2 below we show the event associated with a system call to send a message to a port by the program of a process. Note that the event type is "Send" and the event parameters include the message being sent and the destination port. Also note the process reference of the event is the process making the call. This event is created when the call is made. The message is actually sent when the kernel executes the event.

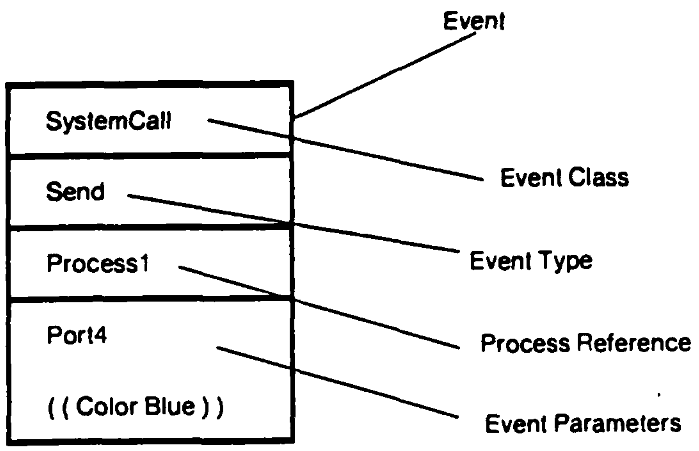


Figure 2.2. An event.

2.3.1. System Calls.

All interprocess activities of a process are controlled by a process's program. A process's program executes system calls to send and receive messages, to create ports and to create other processes. We assume this program also executes other code related to its internal data structures. We are not concerned here with any aspect of the internals of these programs, other than their connection with the multi-process environment through the system calls. For the sake of convenience of notation, we will occasionally say that a "process makes a system call" when we mean "the process's program makes a system call".

We make the restriction that process's programs cannot make interprocess system calls with foreign objects except:

- to send messages to foreign ports and

- to test the program state of processes created by the program ("child" processes).

In terms of capabilities, each process's program has all access rights to all objects local to the process and the right to read the program state of its child processes. A process's program may obtain the right to send messages to a foreign port through the name service or by receiving a port reference in a message. Such protection is possible since all access to interprocess objects, local or foreign, can only be made through system calls.

Each system call creates an event representing the call. Each such event has the following structure:

- the event class is "SystemCall",
- the event type is the name of the system call,
- the event parameters are the actual arguments of the system call provided by the calling program and
- the event process reference is the process containing the program making the call.

As a side-effect of the execution of every system call event, a result is returned to the program. (For some system calls, this result may be of no value, but the program will always wait for it to be returned.) Note that this means the program is effectively suspended for every system call until the kernel has executed the event and returned a result.

As another side-effect, some system calls may cause other events to occur. For an example of this, consider the system call "Receive", which takes as its only argument a local port reference. This system call creates an event of type "Receive". The execution of this event creates an event of type "ReceiveWait", whose event parameters include the port reference. If there is a message on the port's message queue, this event will remove the first message and return it to the program as the result of the original system call. Otherwise, the event creates another "ReceiveWait" event and returns nothing to the program. More details on message transmission is contained in the next section.

2.3.2. Message Transmission.

Message transmission activities are based on the concept of borders that exist around specific interprocess objects. Processes, ports and code objects have borders which much be crossed by messages during their transmission. Each border is bidirectional, with messages crossing in either the outgoing or incoming direction. The order of border crossings of a typical message transmission is shown in the Figure 2.3 below. A message M is sent from the program of Process1 to Port1 of Process2. This message:

- crosses the outgoing border of the program in Process1,

- crosses the outgoing border of Process1,

- crosses the incoming border of Process2 and finally

- crosses the incoming border of Port1 and is put on Port1's message queue.

Note that a copy of the message is sent and the original is left in the program in Process1.

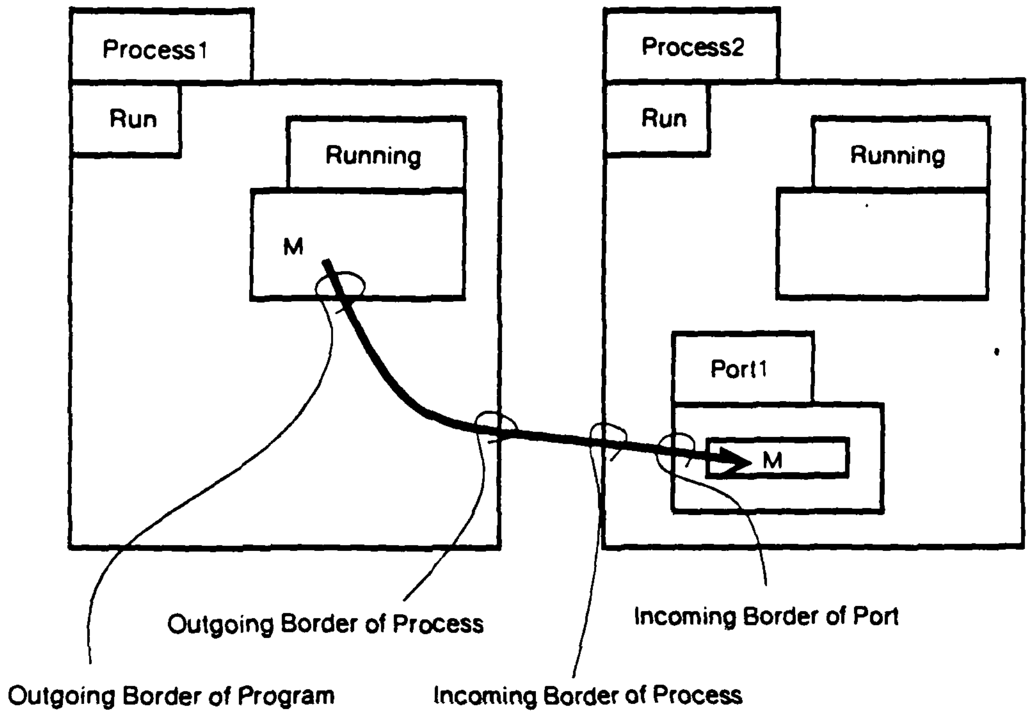

Process1
Run
Running
M
Process2
Run
Running
Port1
M
Outgoing Border of Process
Incoming Border of Port
Outgoing Border of Program
Incoming Border of Process

Figure 2.3. Message transmission: sending a message.

In Figure 2.4 below, we show the receipt of message M by Process2 from Port1. The message:

- is removed from Port1's message queue and crosses the outgoing border of Port1,

- crosses the incoming border of the program in Process2 and then

- is returned to the program.

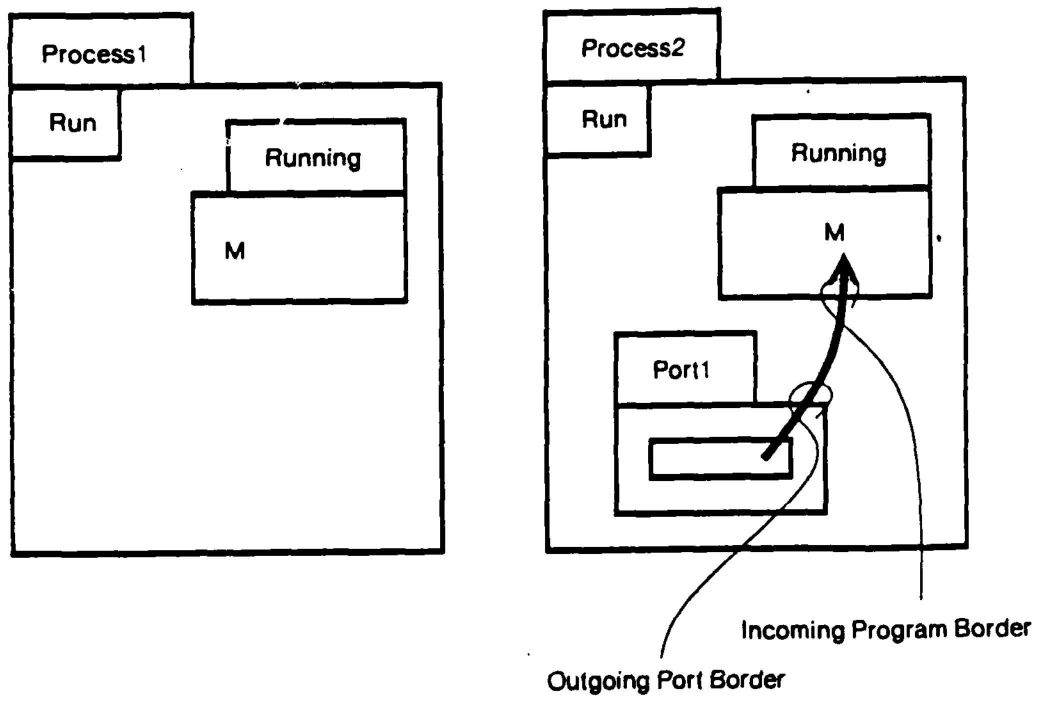


Figure 2.4. Message transmission: receiving a message.

Each border crossing is a separate event and has its own structure. We now give all the details of the above message transmission, beginning with the program of Process1 making the system call to send message M to port Port1. This call creates the event shown in Figure 2.5 below and changes the program state of Process1 to "SystemCall".

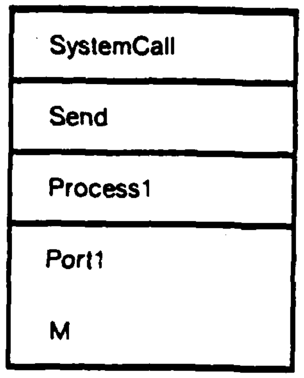


Figure 2.5. "Send" event.

The execution of this event changes the program state of Process1 to "Running", returns a result to the program and creates the first border crossing event. The structure of this event is shown in Figure 2.6 below.

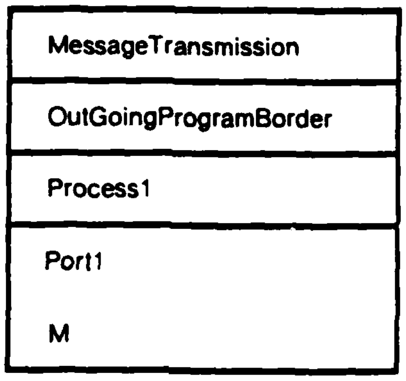


Figure 2.6. "CrossingOutGoingProgramBorder" event.

The execution of this event creates the next border crossing event, shown in Figure 2.7 below. Note for any object border being crossed, the event process reference is always the process containing that object.

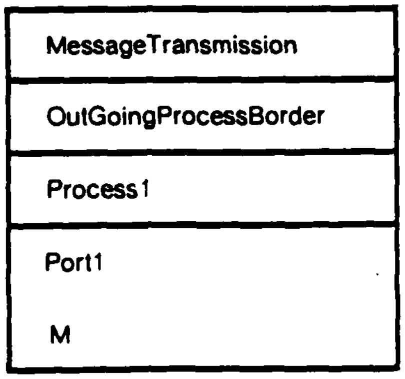


Figure 2.7. "CrossingOutGoingProcessBorder" event.

This event creates the next border crossing event, shown in Figure 2.8 below. Note that the event process reference has been changed.

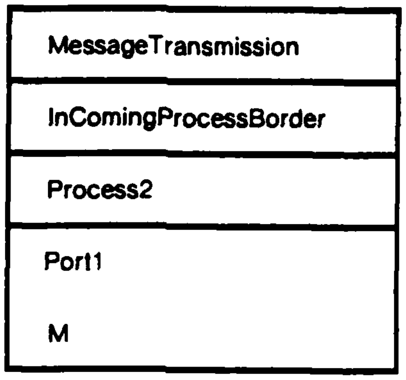


Figure 2.8. "CrossingInComingProcessBorder" event.

This event now creates the following final border crossing, shown in Figure 2.9 below.

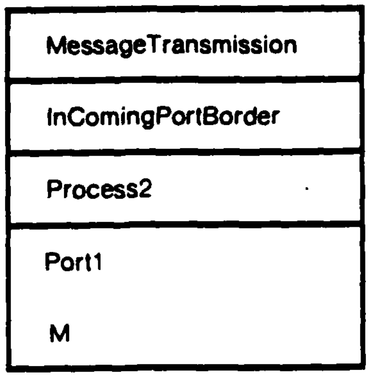


Figure 2.9. "CrossingInComingPortBorder" event.

The execution of this event causes the message M to be added to the end of Port1's message queue.

Now, suppose the program in Process2 executes the "Receive" system call on Port1. This call creates the event shown in Figure 2.10 below and changes the program state of Process2 to "SystemCall".

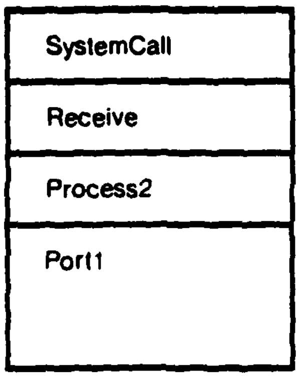


Figure 2.10. "Receive" event.

The execution of this event creates the "ReceiveWait" event shown in Figure 2.11 below and changes the program state of Process2 to "ReceiveWait". Note that no result will be returned to the program until the message is removed from the port and has crossed all the appropriate borders.

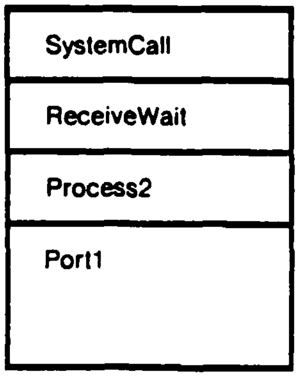


Figure 2.11. "ReceiveWait" event.

The execution of this event depends on the state of the message queue of the port reference found in the event parameter. If the message queue of this port is empty, the event creates an identical event. (Remember that events are destroyed after execution, so this event must create another one even though it is the same event.) Thus the system will continue executing these "ReceiveWait" events until a message appears in the message queue of the port. Since we have a message in the message queue of Port1, this event creates the border crossing event shown in Figure 2.12 below.

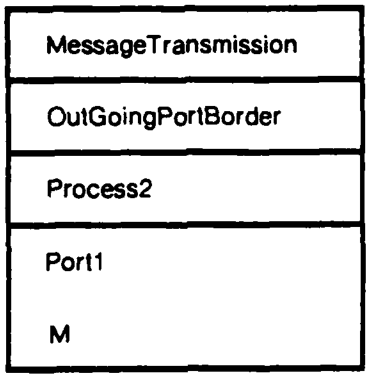


Figure 2.12. "CrossingOutGoingPortBorder" event.

The execution of this event creates the next border crossing event, shown in Figure 2.13 below.

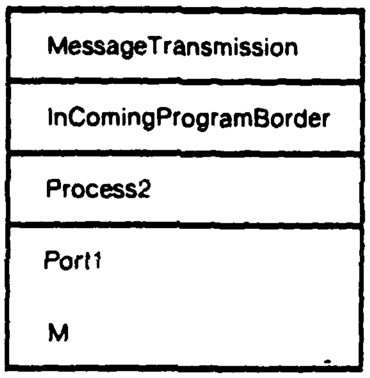


Figure 2.13. "CrossingInComingProgramBorder" event.

The execution of this event changes the program state of Process2 to "Running" and returns the message to the program.

Dividing message transmission into this fine a grain will be used later for debugging. Intuitively, we want to be able to detect and intercept classes of messages as determined by the borders they cross, such as any mess[illegible] at a process (incoming process border), any messa[illegible] a particular port (incoming port border), any messag[illegible] a program (outgoing program border), and so on. Sim[illegible] use of this feature will appear in Chapter 3, which describes our debugging tools and techniques, and in Chapter 4, which describes our demonstration program.

Chapter 3
Interprocess Debugging Tools and Techniques

3.1. Introduction.

In this chapter, we describe our tools and techniques for debugging communicating, loosely-coupled processes. The tools are based on the manipulation of interprocess events as defined in our model of the previous chapter. The techniques are based on the use of our tools to achieve the debugging goals of process control, process monitoring and process testing.

We describe in Chapter 4 how these debugging tools can be incorporated into a multi-process kernel. Chapter 4 also discusses how the kernel handles the execution of interprocess events for processes. In [illegible] 5, we give several sample debugging sessions using our SPIDE[illegible]stration program to show these debugging tools and techniques at work.

3.2. Debugging Tools.

This section describes our debugging tools for the multi-process domain. These tools manipulate interprocess events in various ways. The debugger is provided to allow a user direct control over interprocess events, such as modifying events, creating events and destroying events. Debugging demons are provided to give the user event-driven control over events. Finally, a transcriber is provided to record events executed for a process for later replay.

These tools provide control over the execution of processes at the grain of the interprocess event. In particular, this control interferes with the normal sequence and timing of events. This interference will not affect our debugging techniques as we use them in this work, but could keep these techniques from detecting sequence or timing related interprocess bugs.

3.2.1. The Debugger.

The debugger provides the user at a terminal direct control over interprocess events. Not only can the debugger inspect or alter events created by processes, but it can also introduce new events. Most of the debugger's abilities are provided by system calls on the kernel. These calls are not available to programs running as processes in the multi-process domain. Commands typed by the user are either translated into a system call or, for a few simple commands, executed directly. Appendix A details all commands available to the user of the debugger and all system calls available to programs.

In general, the user of the debugger can cause any action in the multi-process domain of system calls that a program can execute. A user can for example create a "Send" event for a process, making it appear to other processes that the process sent a message. Using the debugger, the user can:

- create, access and modify interprocess objects,
- preview, single-step and replace individual interprocess events,
- create, send and receive messages,
- start, stop and replace programs in processes,
- enable and disable debugging demons (see section 3.2.2 on demons below) and
- start and stop the transcribing of a process's interprocess events (see section 3.2.3 on the transcriber below).

Some of these features could be accomplished with nothing more than access to the system calls normally available to programs. However, in order to provide for interprocess debugging with demons, for the transcribing of events executed in the kernel or for the replacement of process's programs themselves, we also make available more powerful, debugger-specific commands.

As described in the previous chapter, each process in our model has an event execution parameter whose value is either "Run" or "Suspend". This parameter specifies whether events will get executed ("Run") or will not get executed ("Suspend") for that process. This parameter cannot be set by a process's program and is intended for use only by the user of the debugger. With the debugger, the user has two commands, Resume and Suspend, for setting the event parameter to "Run" or "Suspend" respectively of a particular process. Events for suspended processes get added to a suspend queue rather than executed.

When a process is suspended, the user can single-step the process at the grain of interprocess events using the SingleStep command. This command causes the next suspended event for that process to be removed from the suspend queue and executed. With the PreviewStep command, the user can view this step before its execution. With the following restrictions, the user can also replace (or destroy) a step:

- a system call event of a program can only be replaced with a result to return to the program and

- a message transmission event can be destroyed or can have its message replaced.

Figure 3.1 below shows a simple program that we will bring up in a process and single-step at the interprocess level.

```
main()
{
    ...
    port = CreatePort(); /* 1 */
    mes = Receive(port);/* 2 */
    Type(mes);  /* 3 */
}
```

Figure 3.1. The program for a process.

We now show the sequence of debugger commands to bring the program up and step through a few of the events that occur for that process. In the following examples, the prompt for commands from the debugger is ">" and all system responses are indented several spaces to the right.

```
> CreateProcess("Suspend")
    created Process1
> StartProgram("main","Process1")
    OK
> PreviewStep("Process1")
    Event Class: SystemCall
    Event Type:  CreatePort
    Event Process:   Process1
    Event Parameters:
> SingleStep("Process1")
    Event Class: SystemCall
    Event Type:  CreatePort
    Event Process:   Process1
    Event Parameters:
> SingleStep("Process1")
    Event Class: SystemCall
    Event Type:  ReturnResult
    Event Process:   Process1
    Event Parameters:    Port1
> SingleStep("Process1")
    Event Class: SystemCall
    Event Type:  Receive
    Event Process:   Process1
    Event Parameters:    Port1
>
```

Figure 3.2. A user session, with single-stepping.

We now show a session where we send a message to a port in this process and single step through the message transmission. Note all the individual message transmission events. Also note the separate event for returning a result to a program and the "ReceiveWait" event used to represent the "state" of a process waiting for a message.

```
> Send("Port1","( ( A OK ) )")
    sent
> SingleStep("Process1")
    Event Class: MessageTransmission
    Event Type:  InComingProcessBorder
    Event Process:   Process1
    Event Parameters:    Port1  ( ( A OK ) )
> SingleStep("Process1")
    Event Class: MessageTransmission
    Event Type:  InComingPortBorder
    Event Process:   Process1
    Event Parameters:    Port1  ( ( A OK ) )
> SingleStep("Process1")
    Event Class: SystemCall
    Event Type:  ReceiveWait
    Event Process:   Process1
    Event Parameters:    Port1
> SingleStep("Process1")
    Event Class: MessageTransmission
    Event Type:  OutGoingPortBorder
    Event Process:   Process1
    Event Parameters:    Port1  ( ( A OK ) )
> SingleStep("Process1")
    Event Class: MessageTransmission
    Event Type:  InComingProgramBorder
    Event Process:   Process1
    Event Parameters:    Port1  ( ( A OK ) )
> SingleStep("Process1")
    Event Class: SystemCall
    Event Type:  ReturnResult
    Event Process:   Process1
    Event Parameters:    ( ( A OK ) )
>
```

Figure 3.3. Sending a message, with single-stepping.

With the debugger, our intention is to provide the user with direct control over interprocess events. This control over a set of programs at the grain of the interactions between processes is a powerful tool for interprocess debugging.

3.2.2. Interprocess Debugging Demons.

This sections describes interprocess debugging demons. Demons provide an event-driven component for our debugging tools. Demons can manipulate interprocess events just as the user of the debugger can. Each demon has a list of commands which are executed when a particular predicate becomes true. The list of commands come from those provided by the debugger. The predicate is an expression based on tests of conditions, objects and events in the multi-process domain.

An _interprocess debugging demon_ is composed of a class, a formal parameter list, a trigger and a command list.

The _class_ of a demon is one of "ReadWrite", "ReadOnlyFinal", "Replace" or "NoEvent".

The _formal parameter list_ is a list of untyped variable names. These parameters will be assigned values upon enabling (defined below). These parameters can be used in the demon trigger or in the command list.

The _trigger_ of a demon is a boolean expression involving tests for various interprocess states and events.

The _command list_ of a demon is a list of commands from those provided by the debugger. Any command executable by the user of the debugger can also be executed by a demon. Because of this, whenever we say the "debugger" can perform some function we also imply that demons can perform the same function. Several other commands are available to demons that are meaningless to the user of the debugger. These commands are used for synchronization with other demons or the debugger and also for certain forms of event manipulation.

The following figure shows a demon and its structure. Appendix A includes details on the syntax of demons.

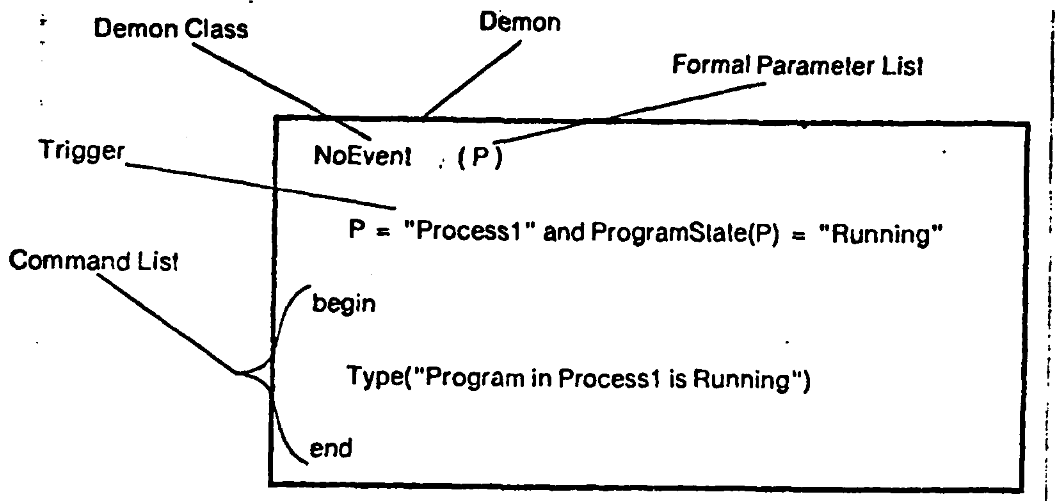

Demon Class
Demon
Formal Parameter List
Trigger
Command List
NoEvent (P)
P = "Process1" and ProgramState(P) = "Running"
begin
Type("Program in Process1 is Running")
end

Figure 3.4. A demon.

Demons are enabled with the debugger command "EnableDemon". This creates an instance of the demon that will have its trigger tested at the appropriate times by the kernel. A demon is removed from this testing by disabling. When the trigger becomes true, the demon is said to fire. When a demon fires, the command list of the demon is executed. After the execution of all commands is completed, the demon is re-enabled by the kernel.

Note that enabling and disabling only affects the testing of the trigger by the kernel. A demon that has fired and is executing can be disabled, but the execution of the demon will be completed. The kernel will not re-enable a demon that has been disabled. This feature ensures that demons will not be interrupted at an arbitrary place in their execution. It also allows demons to disable themselves for "one-time-only" execution.

When a demon is enabled, actual parameters are specified to match with the demon's formal parameters. These parameters allow several instances of the same demon to be used to watch for the occurrence of particular events on different objects.

Each demon is specified with a particular class. Demons can fire because of certain interprocess states or because of an attempt by the kernel to execute a certain interprocess event. All demons in the former case are of class "NoEvent". In the latter case, the demons must be in one of the other classes. The event is called the interrupted event because it will be suspended during execution of the demons. For several demons which fire on the same event, their class determines a partial ordering for their execution. This ordering is based on the need to let certain groups of demons perform their functions on the interrupted event before other demons execute. The demon classes and their intended usage is as follows:

- "ReadWrite" class is for demons that monitor events as they happen or want to modify events,
- "ReadOnlyFinal" class is for demons that want to monitor events after all demons have modified them,
- "Replace" class is for demons that want to replace or abort the event altogether and
- "NoEvent" class is for demons that do not fire on any particular event.

We define the trigger set as the set of all demons of a particular class that fire on a particular event. For each event, the trigger set of demons of the class "ReadWrite" is created. If the trigger set is not empty, each demon in this set is executed one at a time in an arbitrary order.

Next, the trigger set is created for all demons of class "ReadOnlyFinal" that fire on the possibly modified interrupted event. These modifications are permanent and the original event is effectively destroyed. The demons in this set are now executed one at a time in an arbitrary order.

Next, the trigger set is created for all demons of class "Replace" that fire on the event. This set can contain at most one demon. If the set is not empty, the demon is executed. This demon is allowed to replace or abort the event.

If the interrupted event has not been aborted, the event, as modified by the demons, is now executed. (This algorithm, including details on how other events get executed or suspended during this process, is described with the kernel description in Chapter 4.)

Note that although demons of the same trigger set are executed one at a time, there can be many demons executing at the same time from different trigger sets. These demons will have either fired on a different event or on no event at all.

Just before any demon trigger is to be tested, the execution environment of that demon is created. The <u>execution environment</u> of a demon is a set of name/value pairs. This set includes:

- name/value pairs formed by matching the values of the demon's actual parameters with the names of the demon's formal parameters,

- the names EventClass, EventType, EventProcess matched with the appropriate values from the interrupted event,

- other event names (based on the event type) whose values are derived from the interrupted event parameters.

All parameters in the execution environment can be used by the demon trigger to decide whether to fire. When the demon executes, it will also have available all of the execution environment. (Note that demons of the class "ReadWrite" that modify event parameters will each see the parameters from the original event. After all demons of this class have executed, the interrupted event is updated using the modified environments from the demons. This event is then used to fire demons of the next class.)

Names in the environment derived from the interrupted event parameters will depend on the particular event class and event type. For instance, for an interrupted event of class "MessageTransmission", the execution environment will include the names "Message" and "Port" for the message being transmitted and the destination port of the transmission respectively.

As an example of these rules for firing demons with classes, we give the three demons below and describe how they will respond to events. Note for the event type "Send", the event parameters DestinationPort and Message are the destination port and the message, respectively, of the "Send" system call.

```
ReadWrite ( P )
    EventProcess = P and EventType = "Send" and
    DestinationPort = "Port5"
begin
     DestinationPort ← "Port1"
end
```

Figure 3.5. FireOnPort.demon.

```
ReadWrite ( P )
    EventProcess = P and EventType = "Send" and
    ValueOfMessage(Message,"Color") = "Blue"
begin
    AddNameValuePair(Message,"( Color Red )")
end
```

Figure 3.6. FireOnMessage.demon.

```
Replace ( P )
    EventProcess = P and EventType = "Send" and
    DestinationPort = "Port1" and
    ValueOfInMessage(Message,"Color") = "Red"
begin
    Send(Message,"Port1")
    Send(Message,"Port2")
    ReplaceEventWithReturnResult("OK")
end
```

Figure 3.7. ReplaceSend.demon

The FireOnPort demon is intended to route all messages sent from process P to Port5 to Port1. FireOnMessage is intended to alter all messages sent from process P with name/value pair "(Color Blue)" to the name/value pair "(Color Red)". The ReplaceSend demon takes all messages sent from process P to Port1 containing the name/value pair "(Color Red)" and sends them to Port1 and to Port2. In particular, the action of FireOnPort and FireOnMessage on message transmission events will affect whether the ReplaceSend demon gets fired. Assuming all three demons above are enabled on the same process P, the following figure gives a chart indicating which demons will fire given the system calls (made by process P's program) as shown.

	FireOnPort	FireOnMessage	ReplaceSend
Send("Port5","((Color Green))")	X		
Send("Port4","((Color Blue))")		X	
Send("Port1","((Color Red))")			X
Send("Port5","((Color Red))")	X		X
Send("Port1","((Color Blue))")		X	X
Send("Port5","((Color Blue))")	X	X	X

Figure 3.8. How the demons fire.

Note that in the fourth and fifth cases, the demon of class "ReadWrite" will execute before the demon of class "Replace". In particular, the execution environment for the ReplaceSend demon will include the modifications made by the other demons. Thus, the ReplaceSend demon will fire because of the modification made to the event by the other demon.

In the sixth case of Figure 3.5, the demons of class "ReadWrite" will be executed in an arbitrary order before executing the other demon. Note that their modifications to the event parameters do not conflict. If there had been conflicts (that is, if they had both modified the same parameter), the event would have been updated with only one of the demon's modified parameters. Again, the ReplaceSend demon will fire because of the execution of the other two demons.

We use demons to provide a mechanism for monitoring and controlling interprocess objects and events automatically. The ability to fire on interprocess events and state changes plus the ability to execute any command available to the debugger makes demons a powerful mechanism for debugging at the interprocess level.

3.2.3. The Transcriber.

The transcriber is a tool that records all interprocess events executed for each process in a file called a transcript. In particular, when transcribing a particular process, this will record every event whose event process refers to that process.

The transcriber records only those events actually executed. Thus events will be recorded only after all demons have fired, executed and (possibly) modified the event. This will not record any events created by the debugger or by demons unless, as a side-effect, they cause events associated with particular processes to occur. For instance, if the debugger is used to send a message to a port in a process being transcribed, no events related to this transmission will be recorded until the message crosses the incoming border of the process.

The transcript can be replayed in a process in place of its program. This method of replay is based on the idea that the program of a process causes events to occur for that process. These events can then be manipulated by the debugger or cause event-driven demons to execute. Similarly, transcript replay causes events to occur for the process doing the replay. These events can be manipulated by the debugger (for instance, single-stepped) and they can be used to fire demons. Demons can access and even modify the event parameters as with all other events. In fact, these events act like any other event in all respects except that they are not executed by the kernel. Thus, the events cause no side-effects directly.

This ability, to use a previous process execution to create events is considered a powerful feature of the debugging techniques described here. Since all our tools are based on the concept of an event and on manipulating events in various ways, this gives us a consistent mechanism for both recording a process's behavior and using this record for other debugging purposes.

3.3. Debugging Techniques.

This section discusses the use of our debugging tools to achieve certain interprocess debugging goals. As we said in the first chapter, the user of a debugger for a message-based, multi-processing environment should have the ability to control, monitor and test processes. Our approach is to provide the tools described above for performing various manipulations on interprocess events and then define the desired techniques in terms of the use of these tools. For the definitions below, the "interprocess events associated with a particular process" are either the events created by a running program for that process, events replayed by a transcript, events created for that process by the debugger or events created for that process by demons.

Process control is the manipulation by the user of the debugger or by demons of the interprocess events associated with a particular process. This control allows the user (or demons) to specify completely the behavior of a process at the grain of interprocess events. In practice, this usually means that only a certain set of the activities of the total behavior of the process will be affected.

Process monitoring is the observation by the user of the debugger or by demons of the interprocess events associated with a particular process. We make the following differentiation between observation and manipulation: observation of a process implies no modification to a process's behavior will be made, manipulation implies that modifications can be made. This difference mostly affects the use of demons, which specify their class based on how they intend to interfere with interprocess events. Obviously, if a process is being controlled, it is also being monitored.

Process testing is the determination of the behavior of a process by process control and process monitoring. By using process control to specify the behavior of other processes and process monitoring to observe the events associated with a certain process, we can effectively test this process. Note that it is possible to use process testing to determine the behavior of a process that process control lets the user completely specify. This is of course a trivial use of process testing. In testing a process, process control is intended to allow the user to specify the behavior of _other_ processes.

Chapter 4
The Interprocess Kernel

4.1. Introduction.

In this chapter, we describe the interprocess kernel. Our kernel is responsible for creating and modifying interprocess objects, for implementing interprocess events and for implementing support for our debugging tools. Our description here will regard only that part of the kernel responsible for implementing the interprocess objects, events and debugging tools as described in Chapters 2 and 3. This kernel can be viewed as an extension to a standard multi-process operating system providing multiple process support (memory management, paging and swapping), a mechanism for programs for making system calls, a file system (for storing programs and demons) and terminal I/O support.

4.2. Basic Function of the Kernel.

The kernel implements the interprocess objects and all operations on them. Each interprocess object has a representation in the kernel containing all items as described in Chapter 2 for each object definition. Each interprocess event also has a representation containing the items given in Chapter 3 for the definition of an event.

In order for an event to be executed, it must first be put on the _event queue_. Figure 4.1 shows this event queue and the basic, top-level loop of the kernel. Events are added to this queue either by other events or by programs executing system calls. Note that since commands executed by the debugger for the user or for demons are treated as system calls, they also add events to this queue. However, to give quick response to the debugging functions, events added by system calls from the debugger take priority over other events.

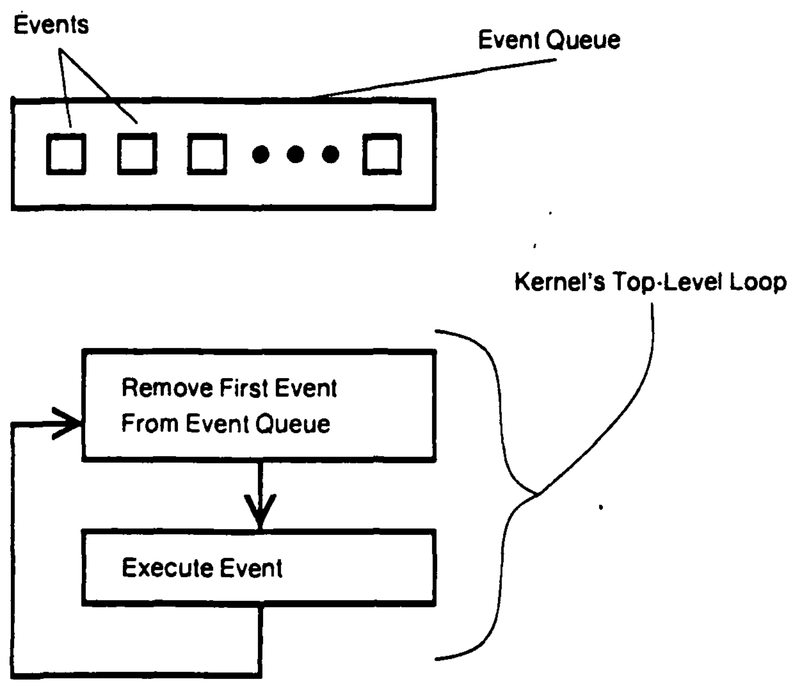


Figure 4.1. The basic kernel.

As shown in Figure 4.1 above, the basic kernel removes a waiting event from the front of the event queue, executes it and then repeats. When an event is executed, it may create or modify interprocess objects or it may add new events to the event queue. After an event has been executed, it is destroyed.

4.3. Modifications for the Debugger.

To the kernel described above, we add certain modifications to support our debugger. Each process has an event execution parameter (defined in Chapter 2) and each event has a process reference (defined in Chapter 3). We define the suspend queue of a process as an event queue associated with that process. (This suspend queue is another component of each process.) If the event execution parameter of the process referenced by an event is "Suspend", the event will be added to the suspend queue for that process instead of being executed. This action is shown in the following figure.

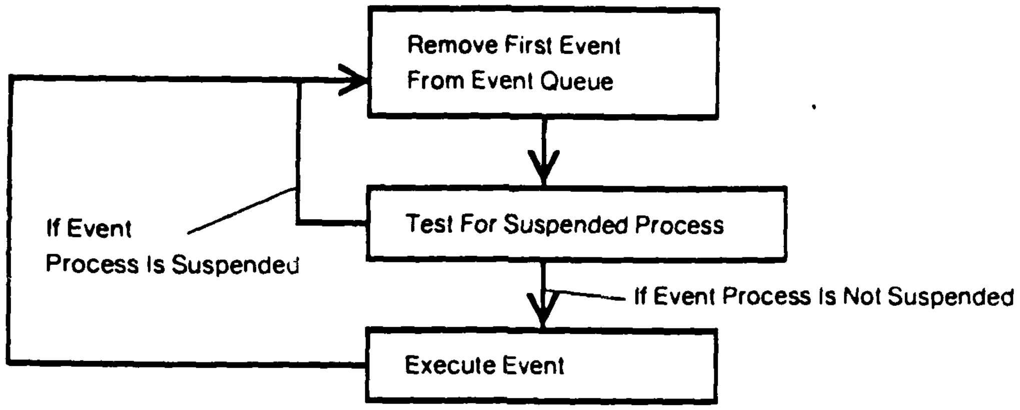

Remove First Event From Event Queue
Test For Suspended Process
If Event Process Is Suspended
If Event Process Is Not Suspended
Execute Event

Figure 4.2. Kernel modified to support the debugger.

Changing the event execution parameter of the process to "Run" causes the events in the process's suspend queue to be added to the end of the kernel's event queue.

4.4. Modifications for Demons.

Support for demons includes a demon trigger evaluator, a demon checker, a representation for trigger sets and various suspend queues for object borders. We define below the various modifications and procedures added to the kernel to support demons.

Each trigger set has a representation in the kernel. This representation includes the interrupted event, the names of all demons yet to be executed, the name of the currently executing demon, the names of all demons that have been executed, the demon class of this set, the initial execution environment and the environments of each executed demon.

The _demon trigger evaluator_ is a procedure that returns a trigger set given an event and a demon class.

The _demon check procedure_ is used by the kernel to manage the firing of demons. As shown in Figure 4.3 below, this procedure is called twice, once for every event that occurs in the kernel and once before any event is removed. Note in the second occurrence of the procedure call in this figure:

- if the procedure returns the null event, the kernel loops to the top,

- if the procedure returns an event, the kernel proceeds to execute it.

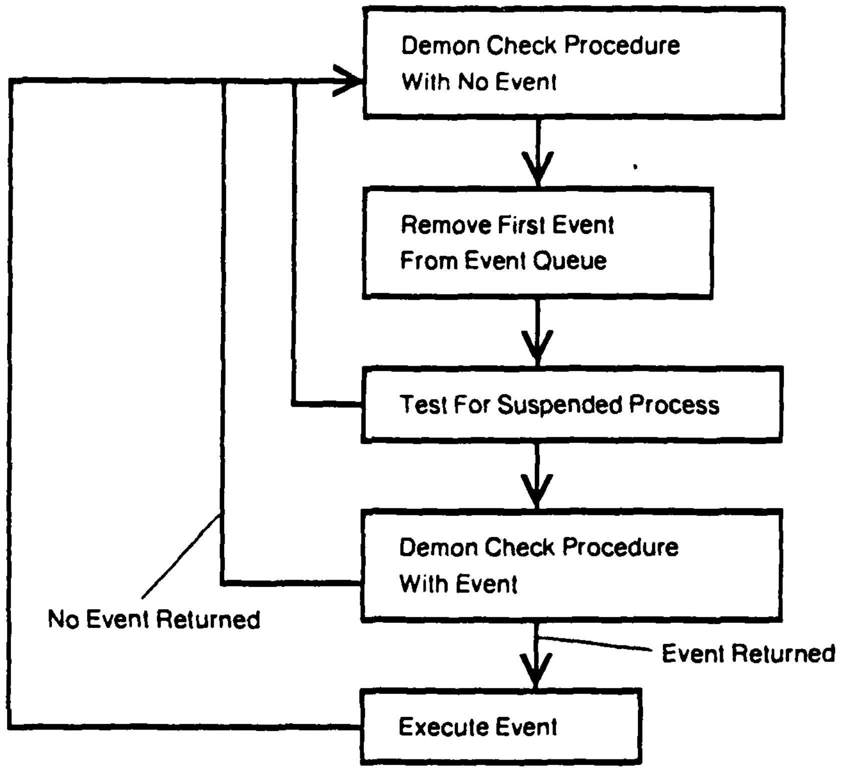

Demon Check Procedure With No Event
Remove First Event From Event Queue
Test For Suspended Process
Demon Check Procedure With Event
No Event Returned
Event Returned
Execute Event

Figure 4.3. Kernel modified to support demons.

When this procedure is called without an event, its task is fairly simple. The procedure calls the demon trigger evaluator with the class "NoEvent" to get the trigger set for this class. If the set is not empty, a demon is started from this set and the procedure returns the null event. If the set is empty, the procedure returns the null event.

When the demon check procedure is called with an event, the procedure does one of two things based on the event type:

- if the event type is "DemonDone", (that is, if a demon has finished executing), the procedure decides on the next demon to execute,

- otherwise, for all other events, the procedure determines the first demon to execute for this event.

To determine the first demon to fire for an event, the procedure calls the demon trigger evaluator with classes "ReadWrite", "ReadOnlyFinal" and "Replace", in that order, until either a non-empty trigger set is returned or no such set is found in any class. If no set is found, no demons are executed and the procedure returns the event. If a non-empty set is found for one of the classes, one of the demons of that set is started with the initial execution environment, the event is associated with the trigger set (and effectively suspended) and the procedure returns the null event.

When a demon finishes execution, it causes an event of type "DemonDone" to occur. When that event occurs, the demon check procedure must decide the next demon to execute. First the procedure looks at the set of names of demons not yet fired in the trigger set of this demon. If this is non-empty, a demon is chosen and started with the initial execution environment and the procedure returns the null event. If this set is empty (that is, if the last demon of this class has just executed), the action of the procedure depends on the demon class of the trigger set. If the demon class is:

- "NoEvent", the trigger set is destroyed and the procedure returns the null event,

- "ReadWrite", the final execution environments are used to update the initial execution environment of the trigger set and the trigger set for class "ReadOnlyFinal" is determined; if this set is not empty, a demon is started with the initial execution environment and the procedure returns the null event; otherwise the following case for class "ReadOnlyFinal" is performed;

- "ReadOnlyFinal", the trigger set for class "Replace" is determined; if this set is not empty, a demon is started with the initial execution environment and the procedure returns the null event; otherwise the following case for class "Replace" is performed;

- "Replace", the trigger set is destroyed and the procedure returns the event as left by the demons (i.e., the null event if the original was aborted by a demon, the possibly modified event otherwise).

Note for each of the cases "ReadWrite" and "ReadOnlyFinal" above, if the trigger set for the next class is empty, the step for that class is executed. In other words, the procedure acts exactly as if the last demon of the next class had just executed.

Note that the position of the demon check procedure in the kernel's top-level loop and the ability of this procedure to return either a null event or an event to execute allows events to be suspended, replaced or aborted altogether.

Another modification we make to the kernel is to suspend borders for trigger sets firing on events of class "MessageTransmission". Suspending a border involves adding four more suspend queues to each process object and two suspend queues to each port object in the kernel. For processes, the four queues are for the incoming and outgoing process borders and for the incoming and outgoing program borders. For ports, the two queues are for the incoming and outgoing port borders.

When a trigger set is created for an event of class "MessageTransmission", the appropriate border (based on the event type) is suspended. Any such event associated with an object with a suspended border will be added to the appropriate suspend queue for that object. Thus, until all demons have finished executing for a particular border crossing event, all other events that cross the same border in the same direction will be suspended. When the trigger set of the suspended event is destroyed, the border is resumed and the events on it are added to the event queue of the kernel.

4.5. Modifications for the Transcriber.

This section describes modifications made to the kernel for the transcriber. We modify the kernel to:

- transcribe events for a process,

- provide for the replay of a transcript.

The following figure shows all our modifications to the kernel in place, including support for the transcriber.

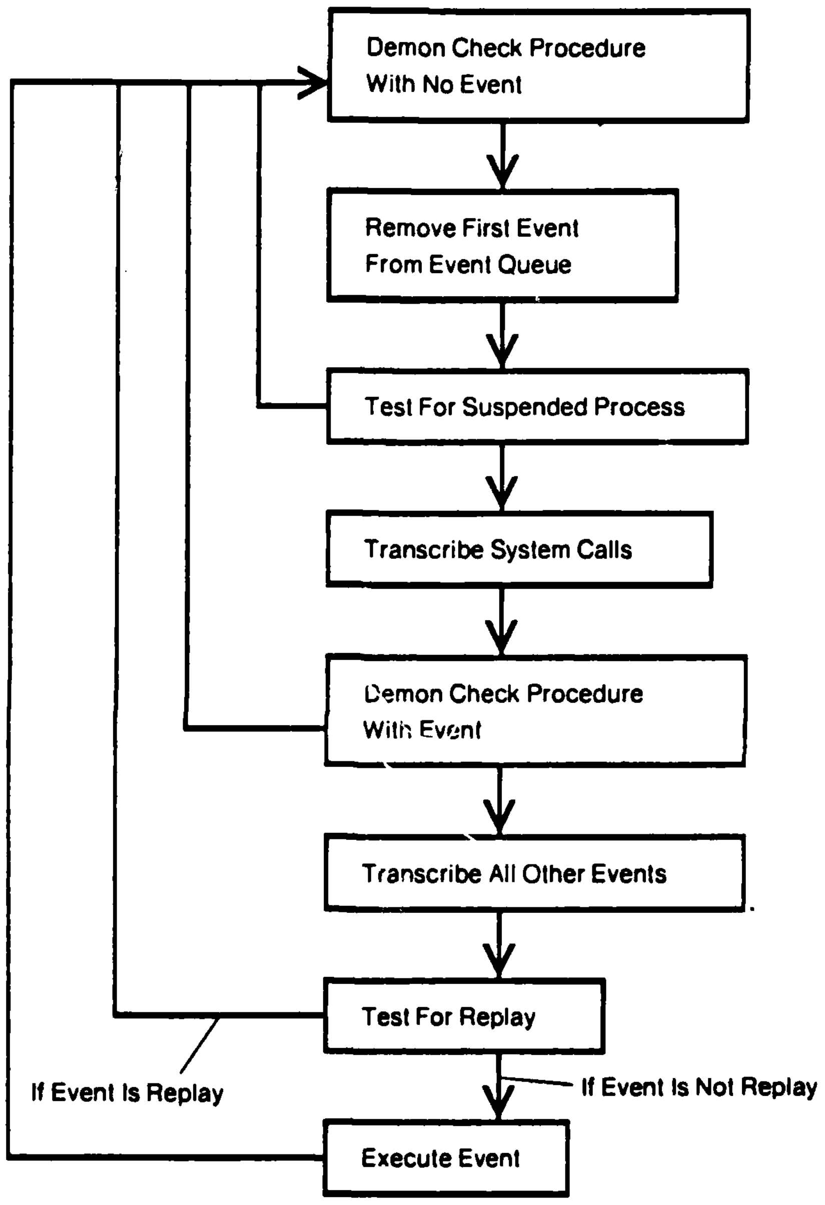

Demon Check Procedure With No Event
Remove First Event From Event Queue
Test For Suspended Process
Transcribe System Calls
Demon Check Procedure With Event
Transcribe All Other Events
Test For Replay
If Event Is Replay
If Event Is Not Replay
Execute Event

Figure 4.4. Kernel modified to support the transcriber.

Support for transcribing events for a process involves two changes. The primary addition is the transcribing of events after the demon checker but before event execution. However, under certain conditions this will miss an important event: whenever a program makes a system call and a demon fires on this event and replaces the call, the transcript will contain only the event regarding the result returned to the program. This will result in a transcript with an event for returning a result to the program unmatched with any system call. For this reason, we also transcribe all system calls before demons are checked.

To support transcript replay, the kernel is modified to read in events from a transcript instead of executing a program for a process. These events are marked as "transcript" events, but in all other respects look like actual events. The kernel loop is modified to destroy these events just before event execution would occur (shown in Figure 4.5 above). Note that transcript events may contain references to objects in the interprocess domain that do not exist. Thus, demons that fire on these events and use their parameters must be aware that the object references in the events are probably bogus.

Chapter 5
A Demonstration of Interprocess Debugging

5.1. Introduction.

In this chapter, we describe SPIDER: the multi-process system built for demonstrating the interprocess debugging tools and techniques of this thesis. SPIDER is a set of programs that together implement the multi-process model of Chapter 2 and our debugging tools of Chapter 3.

We also give several examples demonstrating the debugging tools and techniques of Chapter 3. This includes the basic control of processes (section 5.2), the monitoring of running processes or transcribed processes (section 5.3) and the testing of a new process (section 5.4).

5.2. The Implementation of SPIDER.

SPIDER is a set of programs implemented in C under the Berkeley Unix operating system running on a VAX/780. The core of SPIDER is the program that implements the multi-process kernel. This kernel effectively extends Unix to provide all the features of the message-based, multi-process model of Chapter 2.

To this kernel, we add our debugging tools. The debugger consists of a program that accepts commands from the user at a terminal and passes them on to the kernel. The kernel is modified to accept these commands and make debugging events out of them. Debugging demons are implemented by adding a mechanism to the kernel to test all demon triggers per event. Another program is created (actually a modified version of the debugger) for executing fired demons. For the transcriber, the kernel is modified to make transcripts of events on a per process basis and to replay transcripts.

When the system is running, a separate Unix process exists for each of the following:

- the kernel,

- the debugger,

- each program associated with a process and

- each executing demon.

The communication medium used between our Unix processes is the Interprocess Communication Facility (IPC) described in [Rashid80]. This medium is used to provide the semantics of interprocess system calls made between the separate address spaces of the kernel and programs. (The debugger's commands are implemented as additional system calls provided by the kernel not otherwise available to the programs of processes.) Figure 5.1 shows the IPC communication paths of the various processes involved. Note that only one kernel was built on one machine. Our model does not describe and SPIDER does not implement a truly distributed multi-process debugging system.

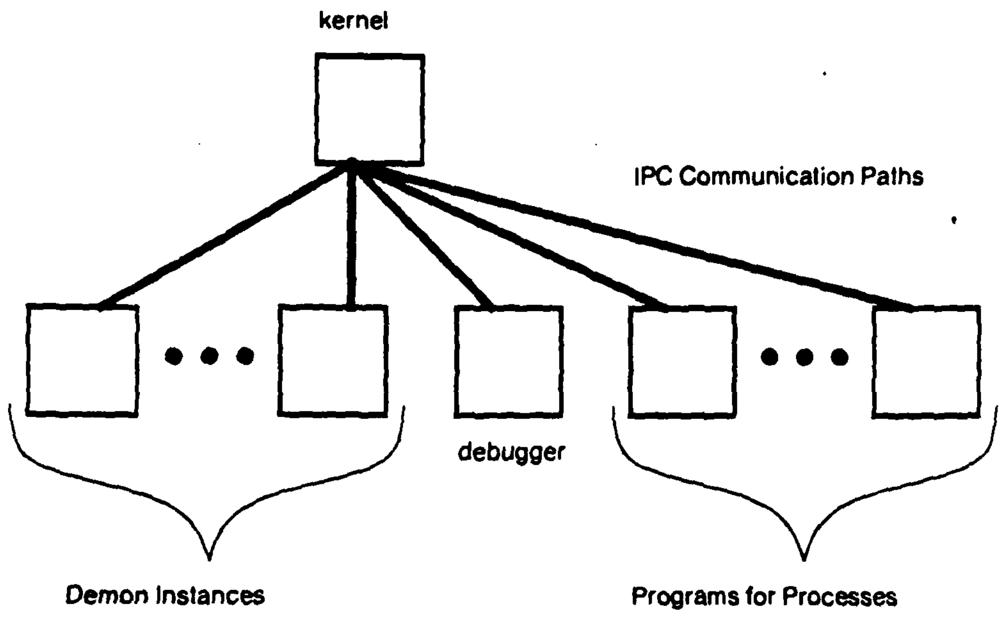


Figure 5.1. Communication paths among our Unix processes.

5.3. An Example: Process Control.

This section shows how we use our tools to create and control processes directly. First, using the debugger, we create a process and examine its state (Figure 5.2 below). Note that when a process is created, an initial value of the event execution parameter is given.

```
15:41:38> CreateProcess("Run")
          Process1
15:41:45> StatusOf("Process1")
          Status Of: Process1
          Event Execution Parameter: Run
          Program State: NoCode
15:41:55> Suspend("Process1")
          OK
15:42:12> StatusOf("Process1")
          Status Of: Process1
          Event Execution Parameter: Suspend
          Program State: NoCode
15:42:19>
```

Figure 5.2. Creating a process with the debugger.

Note this process has no program associated with it and will cause no events to occur by itself. Now we associate a program called the Sender with this process. Also in another process we associate a program called the Receiver. When both of these processes are running, the Sender will send messages containing either the string "Blue" or "Red" to the Receiver. The Receiver will count the number of "Blue"s and "Red"s received and record this in two process variables, "NumBlues" and "NumReds". This interaction is shown below in Figure 5.3. Note the use of the Resume and Suspend commands, which change the value of the event execution parameter of the process to "Run" or "Suspend", respectively.

```
15:44:23> StartProgram("Sender","Process1")
          OK
15:44:30> CreateProcess("Suspend")
          Process2
15:44:42> StartProgram("Receiver","Process2")
          OK
15:45:00> Resume("Process2")
          OK
15:45:21> Resume("Process1")
          OK
15:45:26> StatusOfAll()
          Status Of: Process1
          Event Execution Parameter: Run
          Program State: Running (Sender)

          Status Of: Process2
          Event Execution Parameter: Run
          Program State: ReceiveWait (on Port1) (Receiver)
          Process Variables: ( NumBlues 3 ) (NumReds 1 )
          Ports: Port1
          Message Queue (Port1):
15:45:53>
```

Figure 5.3. A process interaction.

We now bring the Receiver program down, clear the process variables and restart another Receiver in Process2. We let both these processes run for a while and then suspend both (Figure 5.4).

```
15:49:04> StopProgram("Process2")
          OK
15:49:29> StatusOf("Process2")
          Status Of: Process2
          Event Execution Parameter: Run
          Program State: NoCode
          Process Variables: ( NumBlues 7 ) ( NumReds 2)
          Ports: Port1
          Message Queue (Port1):
15:51:43> AssertProcessVariableInProcess("Process2","( NumReds 0 )")
          OK
15:51:59> AssertProcessVariableInProcess("Process2","( NumBlues 0 )")
          OK
15:52:15> StatusOf("Process2")
          Status Of: Process2
          Event Execution Parameter: Run
          Program State: NoCode
          Process Variables: ( NumBlues 0 ) ( NumReds 0 )
          Ports: Port1
          Message Queue (Port1):
15:52:22> Remove("Receiver")
          OK
15:52:59> StartProgram("Receiver","Process2")
          OK
15:53:10> Resume("Process1")
          OK
15:54:02> StatusOf("Process2")
          Status Of: Process2
          Event Execution Parameter: Run
          Program State: ReceiveWait (on Port2) (Receiver)
          Process Variables: ( NumBlues 9 ) ( NumReds 1 )
          Ports: Port1 Port2
          Message Queue (Port1): ( ( Color Blue ) )
          Message Queue (Port2):
15:54:21> Suspend("Process1")
          OK
15:54:22> Suspend("Process2")
          OK
15:54:22>
```

Figure 5.4. Another Receiver.

As we have said, the event execution parameter specifies whether events will or will not be executed for a process. With both processes now suspended, no events are being executed. We now single-step the Receiver process (not the Receiver program) to show some of the individual events (Figure 5.5).

```
15:58:58> SingleStep("Process2")
          Event Class: SystemCall
          Event Type: ReceiveWait
          Event Process: Process2
          Event Parameters: Port2
15:59:18> SingleStep("Process2")
          Event Class: MessageTransmission
          Event Type: InComingProcessBorder
          Event Process: Process2
          Event Parameters: Port2 ( ( Color Blue ) )
15:59:25> SingleStep("Process2")
          Event Class: MessageTransmission
          Event Type: InComingPortBorder
          Event Process: Process2
          Event Parameters: Port2 ( ( Color Blue ) )
15:59:34> SingleStep("Process2")
          Event Class: MessageTransmission
          Event Type: OutGoingPortBorder
          Event Process: Process2
          Event Parameters: Port2 ( ( Color Blue ) )
15:59:37> SingleStep("Process2")
          Event Class: MessageTransmission
          Event Type: InComingProgramBorder
          Event Process: Process2
          Event Parameters: Port2 ( ( Color Blue ) )
15:59:47> SingleStep("Process2")
          Event Class: SystemCall
          Event Type: ReturnResult
          Event Process: Process2
          Event Parameters: ( ( Color Blue ) )
15:59:50> SingleStep("Process2")
          Event Class: SystemCall
          Event Type: AssertObject
          Event Process: Process2
          Event Parameters: ( NumBlues 5 )
15:59:53> SingleStep("Process2")
          Event Class: SystemCall
          Event Type: ReturnResult
          Event Process: Process2
          Event Parameters: OK
15:59:56> SingleStep("Process2")
          Event Class: Sys emCall
          Event Type: Receive
          Event Process: Process2
          Event Parameters: Port2
```

Figure 5.5. Single-stepping Receiver.

5.4. An Example: Process Monitoring.

In this section, we use our tools to monitor processes. This will involve monitoring a previous process execution (from the above example) and monitoring a running process.

For the first monitoring example, we monitor a process that has already finished executing by using a transcript of its execution. We assume the user has made a transcript of the Receiver process (from the above section). This transcript is taken while the Receiver works on a set of messages from the Sender. We now monitor the transcript with the demon shown in Figure 5.6 below. This demon will notify the user when a message is sent containing both strings "Blue" and "Red". In Figure 5.7 below that, we show the session at the debugger to perform this monitoring.

```
ReadWrite ( P )
    EventType = InComingProcessBorder and
    EventProcess = P and
    ValueOfInMessage(Message,"Color") = "( Blue Red )"
begin
    i = ValueOfProcessVariable(P,"MessageCount")
    AssertProcessVariable(P,MakeNVPair("MessageCount",i + 1))
    Type("Got message with both")
end
```

Figure 5.6. MonitorBoth.demon.

```
16:19:40> CreateProcess("Suspend")
          Process1
16:19:56> StartTranscript("T","Process1")
          OK
16:20:33> EnableDemon("MonitorBoth.demon",("Process1"))
          Demon1
16:20:46> Resume("Process1")
          OK
16:20:55>
          Demon1:
          Got message with both
16:20:59>
          Demon1:
          Got message with both
16:21:04>
          Demon1:
          Got message with both
16:21:08> SuspendAll()
          OK
16:21:11> StatusOf("Process1")
          Status Of: Process1
          Event Execution Parameter: Suspend
          Program State: (Transcript T)
          Process Variables: ( MessageCount 3 )
16:21:18>
```

Figure 5.7. Monitoring messages with the debugger.

Note that this count is kept in the process variable "MessageCount". Also note that since this is a transcript replay, none of the original system call events recorded in the replay that created process variables are executed, so the process shows no other variables.

For our second monitoring example, we actually run the Sender and Receiver programs and again monitor the messages using the MonitorBoth demon from Figure 5.6 above. The debugging session to perform this monitoring is shown below that in Figure 5.8.

```
16:23:01> CreateProcess("Run")
          Process1
16:23:10> CreateProcess("Suspend")
          Process2
16:23:27> StartProgram("Sender","Process1")
          OK
16:23:32> StartProgram("Receiver","Process2")
          OK
16:23:41> EnableDemon("MonitorBoth.demon",("Process2"))
          Demon1
16:24:00> Resume("Process2")
          OK
16:24:28>
          Demon1:
          Got message with both
16:24:43>
          Demon1:
          Got message with both
16:24:51>
          Demon1:
          Got message with both
16:24:59>
          Demon1:
          Got message with both
16:25:05>
          Demon1:
          Got message with both
16:25:15>
          Demon1:
          Got message with both
16:25:21> SuspendAll()
          OK
16:25:23> StatusOf("Process2")
          Status Of: Process2
          Event Execution Parameter: Suspend
          Program State: ReceiveWait (on Port1) (Receiver)
          Process Variables: ( NumBlues 5 ) ( NumReds 3 )
                    ( MessageCount 6 )
          Ports: Port1
          Message Queue (Port2): ( ( Color Blue ) ) ( ( Color ( Blue Red ) ) )
16:25:32>
```

Figure 5.8. Another monitor session.

5.5. An Example: Process Testing.

This section shows an example of process testing. Process testing uses a mix of the above techniques of process control and monitoring to achieve the goal of convincing the user his processes work.

For this example, we will test a new version of the Receiver process, Receiver2. Receiver2 counts messages with "Blue"s and "Red"s but also counts messages that contain both, recording this count in "NumBoth". A message containing both will cause just NumBoth to be updated. We show the session below that brings up this program and runs the process until its program state becomes "ReceiveWait". In Figure 5.9 below, we show the demon used to notify the user when this has occurred. In Figure 5.10 below that, we show the session with the debugger for this.

```
NoEvent ( P )
    ProgramState(P) = "ReceiveWait"
begin
    Suspend(P)
    Type(P," in ReceiveWait")
    Type(P," suspended")
    DisableDemon(MyName)
end
```

Figure 5.9. WatchForReceiveWait.demon

```
16:31:20> CreateProcess("Suspend")
          Process1
16:31:30> StartProgram("Receiver2","Process1")
          OK
16:31:52> EnableDemon("WatchForReceiveWait.demon",("Process1"))
          Demon1
16:32:09> Resume("Process1")
          OK
16:32:16>
          Demon1:
          Process1 in ReceiveWait

16:32:19>
          Demon1:
          Process1 suspended
16:32:19>
```

Figure 5.10. Bring up Receiver2.

Now we send this process messages with various "Blue"s and "Red"s and watch all three counts until we are satisfied the program works. The debugger session for this is in Figure 5.11 below.

```
16:32:20> Resume("Process1")
          OK
16:32:30> StatusOf("Process1")
          Status Of: Process1
          Event Execution Parameter: Run
          Program State: ReceiveWait (on Port1) (Receiver2)
          Ports: Port1
          Message Queue (Port1):
16:32:40> Send("Port1","( ( Color Blue ) )")
          OK
16:33:05> Send("Port1","( ( Color Blue ) )")
          OK
16:33:30> Send("Port1","( ( Color ( Blue Red ) ) )")
          OK
16:33:54> StatusOf("Process1")
          Status Of: Process1
          Event Execution Parameter: Run
          Program State: ReceiveWait (on Port1) (Receiver2)
          Process Variables: ( NumBlues 2 ) ( NumReds 0 ) ( NumBoth 1 )
          Ports: Port1
          Message Queue (Port1):
16:36:11> Suspend("Process1")
          OK
16:36:22> Send("Port1","( ( Color ( Blue Red ) ) )")
          OK
16:36:34> SingleStep("Process1")
          Event Class: SystemCall
          Event Type: ReceiveWait
          Event Process: Process1
          Event Parameters: Port1
16:36:40> SingleStep("Process1")
          Event Class: MessageTransmission
          Event Type: InComingProcessBorder
          Event Process: Process1
          Event Parameters: Port1 ( ( Color ( Blue Red ) ) )
```

Figure 5.11.a. A final debugger session.

```
16:36:44> SingleStep("Process1")
          Event Class:  MessageTransmission
          Event Type:  InComingPortBorder
          Event Process:  Process1
          Event Parameters:  Port1  ( ( Color ( Blue Red ) ) )
16:36:48> SingleStep("Process1")
          Event Class:  MessageTransmission
          Event Type:  OutGoingPortBorder
          Event Process:  Process1
          Event Parameters:  Port1  ( ( Color ( Blue Red ) ) )
16:36:51> SingleStep("Process1")
          Event Class:  MessageTransmission
          Event Type:  InComingProgramBorder
          Event Process:  Process1
          Event Parameters:  Port1  ( ( Color ( Blue Red ) ) )
16:36:53> SingleStep("Process1")
          Event Class:  SystemCall
          Event Type:  ReturnResult
          Event Process:  Process1
          Event Parameters:  ( ( Color ( Blue Red ) ) )
16:36:57> SingleStep("Process1")
          Event Class:  SystemCall
          Event Type:  AssertObject
          Event Process:  Process1
          Event Parameters:  ( NumBoth 2 )
16:37:00> SingleStep("Process1")
          Event Class:  SystemCall
          Event Type:  ReturnResult
          Event Process:  Process1
          Event Parameters:  OK
16:37:03> SingleStep("Process1")
          Event Class:  SystemCall
          Event Type:  Receive
          Event Process:  Process1
          Event Parameters:  Port1
16:37:12> SingleStep("Process1")
          Event Class:  SystemCall
          Event Type:  ReceiveWait
          Event Process:  Process1
          Event Parameters:  Port1
```

Figure 5.11.b. A final debugger session (cont.).

Chapter 6
Conclusions

6.1. Contributions and Conclusions.

The evolution of program debuggers has resulted in systems that describe the particular activities of a computer in terms of the abstractions used by the programmer to design and create his programs. Research into debugging systems has led to the use of programming concepts (such as dynamic and static typed variables, linked-lists, record structures, procedure instances, objects and object constraints) as models for displaying the behavior of a running program to the user. Such systems share many common characteristics, including:

- the use of a debugging language consistent with the user's programming language,

- mechanisms to filter out irrelevant information,

- dynamic program monitoring facilities and

- the ability to test parts of programs without involving the entire program.

In this work, the programming model is that of communicating, loosely-coupled processes and it is exactly the abstractions of this model that form the basis of our debugging system. The use of communicating, multiple processes as a model of computation provides a clean "cut" of what information about the activities of the system is interesting to the user and what information is not. In particular, we define a set of <u>interprocess</u> <u>events</u> that describe all possible interprocess activities of our model. Our debugging system has various facilities that allow the user to monitor and control these events.

Our multi-process model defines interprocess objects and activities. We use these concepts to design a debugging language that expresses the manipulations a user can make on the interprocess domain. This language allows the user to control and monitor interprocess activities in terms of the interprocess model. We also describe debugging tools for accessing and manipulating interprocess events. Using our interprocess debugging language as an interface, these tools provide the user with a view of his programs strictly in terms of their interprocess behavior. (This behavior is, of course, defined in terms of the interprocess events executed by the individual processes.)

Our debugging tools and techniques provide for the monitoring and controlling of individual events, the filtering of irrelevant information, the recording and replaying of interprocess events and the isolation and testing of individual processes. The tools consist of a <u>debugger</u> program that provides a direct interface to the user for giving commands in our interprocess debugging language, <u>interprocess debugging demons</u> that provide automatic, event-driven debugging commands and a <u>transcriber</u> that records all interprocess events executed for a single process. These tools are used to achieve the debugging techniques of <u>process control</u>, <u>process monitoring</u> and <u>process testing</u>.

We claim that the debugging of single programs is better understood because programming languages and the programming abstractions provided by these languages are also better understood than distributed programming abstractions. The development of more sophisticated programming languages and the combined experiences of programmers with such languages has provided an important motivation for the development of more sophisticated debugging systems. In particular, research that has <u>followed</u> the development of programming languages has resulted in debugging systems that provide the user with better representations of his programming abstractions. Experience gained with these debugging systems has provided further insight into future directions for such research.

The development of the programming model of message-based, communicating multiple processes to organize large sets of programs has provided a prime motivation for the research reported here. Multi-processing makes the understanding of the behavior of each single program in the system more difficult since a program's behavior depends on the behavior of others. Our debugging tools help reduce this added complexity by providing the user with the means to monitor and control the behavior of his programs at the level of their interprocess interactions. Thus, we believe this approach to solving the debugging problems of communicating, loosely-coupled processes is justified.

6.2. Future Work.

Our use of borders around certain interprocess objects serves as a "syntactic sugar" to create a particular set of interprocess events to express message communication events between processes, ports and programs. It is reasonable to assume that the choices made in this work for the placement of borders will not satisfy all future needs of multi-process programmers. A more general idea would be to allow user-defined borders. A user could place such a border around a set of any arbitrary processes. Message traffic in or out of this set would create an event related to the corresponding border. Note this implies the event types of these border events would also be user-defined.

[Lampson80] suggested an even broader approach to the classification of "interesting" events: the user would specify as an event any kind of system activity, from the access of individual program variables to the activities between sets of large numbers of processes. The use of any system activity as an event implies that our debugging system could be used on other types of multi-process systems. For instance, multiple processes that communicate through a Hoare monitor could use the tools and techniques of this work by having procedures in the monitor cause events instead of executing their code. Only when our kernel (under the control of the debugger) decided an event should be executed, would the actual code for the procedure be executed. In general, any system-detectable activity, which includes message transmission, calls on any procedure, references to any program variable or structure and so on, can potentially create a corresponding event that our event-oriented and event-driven debugging tools can be used to monitor and control.

The kernel algorithm described in detail in Chapter 4 could also be made more general by introducing a "meta-demon" approach to the debugging modifications shown. The only modification to the kernel algorithm would be a mechanism to fire meta-demons. A meta-demon would consist of a trigger and a command list, much like an ordinary debugging demon. The trigger would be a boolean expression of tests on interprocess information and event parameters, but the command list would contain instructions to transcribe or suspend the current event, fire a set of debugging demons, or cause other such debugging-related activity. Thus, the primary function of these meta-demons would be to give the user more control over what parts of the system were debugged and what were not.

The meta-demon feature, for example, would make it possible to record one transcript for the events executed by a set of processes. Irrelevant events could be filtered out of a potentially long transcript by meta-demons. Meta-demons could also be used to create the additional message transmission events related to border crossings, (user-defined or otherwise). In particular, this would also provide a mechanism for decreasing the overhead incurred in the creation of events that are used just for debugging. That is, the user would not only have control over the debugging of interesting system activities, but he would also have control over what system activities create what events in the first place.

Distributing processes over separate computers that communicate via networks adds still more complexity to the debugging of multiple processes. We have specifically addressed the problems encountered in accessing and manipulating interprocess objects and activities for a single, multi-process kernel on a single machine. Distributing the processes over separate computers implies distributing the kernel functions used by our debugging tools.

One obvious way of using our system for debugging distributed processes would be to have all remote kernels contact an instance of our kernel before performing any interprocess events. This centralized approach would involve implementing an extra (but trivial) protocol layer between the kernel used for debugging and the kernels on remote machines. This method could be implemented relatively quickly and would probably work but would also probably be far too expensive for practical applications.

Distributing the kernels implies distributing at least part of the debugging tools. By simply making our kernel modifications to all kernels and implementing an extra (but trivial) protocol layer between the remote kernels and our debugger, many of our same debugging functions would immediately become available. In particular, debugging commands, demons (or meta-demons) and transcripts that apply to processes local to one machine (and thus one kernel) would work as always. Such a system could also be implemented relatively quickly and would incur less overhead than one using a central debugging kernel.

However, even this approach still leaves open questions about events occurring across machine boundaries. For instance, there is no place in these schemes to determine when to fire debugging demons that refer to events or information between machines. Nor is it exactly clear what it means for events to occur between machines: having a demon that fires on particular values of two process state variables on two different machines implies also having an interpretation for "the simultaneous occurrence of two events on two separate machines". The location of the observer (or the debugger) in such a situation will of course determine whether the events appear to occur simultaneously or in some order. We claim that a greater understanding of distributed programming and of the programming abstractions necessary to deal with the distributed domain must occur before anyone can begin writing better distributed debuggers.

Work on distributed debuggers can only follow future work in distributed architectures, distributed operating systems and language constructs for distributed programming. A system such as ours or the ones we proposed here will provide the basis for more research by giving people experience in actually writing distributed programs. Further work into the area of debuggers and program development tools for the multi-process and distributed process domain can only occur after enough experience has been gained with programming in such a domain.

Bibliography

[Ball75] Ball, J. E., J. A. Feldman, J. R. Low, R. F. Rashid, P. D. Rovner, RIG, Rochester's Intelligent Gateway: System Overview, IEEE Trans. Software Eng., Vol. 2, No. 4, December 1975, 321-328.

[Ball78] Ball, J. E., E. Burke, I. Gertner, K. Gradischnig, R. F. Rashid, personal conversations, 1978.

[Balzer73] Balzer, R. M., An Overview of the ISPL Computer System Design, CACM, February 1973, 117-122.

[Baskett77] Baskett, Forest, John H. Howard and John T. Montague, Task Communication in Demos, Proceedings of the Sixth ACM Symposium on Operating Systems Principles, November 1977, 23-31.

[Bobrow72] Bobrow, D. G., J. D. Burchfiel, D. L. Murphy, R. S. Tomlinson, Tenex, a Paged Time Shared System for the PDP-10, CACM, March 1972, 135-143.

[Borning79] Borning, Alan, ThingLab -- A Constraint-Oriented Simulation Laboratory, SSL-79-3, July 1979, Xerox Corp., 100 pp., Stanford Ph.D. thesis.

[Feldman79] Feldman, Jerome A., High-level Programming for Distributed Computing, CACM, Vol. 22, No. 6, June 1979, 353-368.

[Gertner80] Gertner, Ilya, Performance Evaluation of Communicating Processes, TR-76, May 1980, Dept. of Computer Science, University of Rochester.

[Lampson80] Lampson, Butler, personal conversation, November 1980.

[Lantz80] Lantz, Keith A., Uniform interfaces for Distributed Systems, TR-63, May 1980, Dept. of Computer Science, University of Rochester.

[Masinter80] Masinter, Larry M., Global Program Analysis in an Interactive Environment, SSL-80-1, January 1980, Xerox Corp.,105 pp., Stanford Ph.D. thesis.

[Model79] Model, Mitchell L., Monitoring System Behavior In a Complex Computational Environment, CSL-79-1, January 1979, Xerox Corp., 179 pp., Stanford Ph.D. thesis.

[Myers80] Myers, Brad A., Displaying Data Structures for Interactive Debugging, CSL-80-7, June 1980, Xerox Corp., M.I.T. Masters thesis.

[Philips81] Philips, Doug, Black-Flag, Draft, June 1981, Dept. of Computer Science, Carnegie-Mellon University.

[Rashid80] Rashid, R. F., An Inter-Process Communication Facility for UNIX, CMU-CS-80-124, June 1980, Dept. of Computer Science, Carnegie-Melon University.

[Reiser75] Reiser, John F., BAIL, A Debugger for SAIL, SAIL Memo AIM-270, Stanford Computer Science Department Report STAN-CS-75-523, October 1975.

[Sandewall78] Sandewall, Erik, Programming in an Interactive Environment: the LISP Experience, Computing Surveys, March 1978, 35-71.

[Satterthwaite75] Satterthwaite, Edwin H., Jr., Source Language Debugging Tools, Stanford Computer Science Report CS-75-494, May 1975, Ph.D. thesis.

[Smith75] Smith, David Canfield, Pygmalion: A Creative Programming Environment, Stanford Artificial Intelligence Laboratory Memo AIM-260, June 1975, Ph.D. thesis.

[Teitelbaum79] Teitelbaum, Tim, The Cornell Program Synthesizer: A Syntax-Directed Programming Environment, Dept. of Computer Science, Cornell University, SIGPLAN Notices, October 1979.

[Winograd75] Winograd, Terry, Breaking the Complexity Barrier Again, SIGPLAN Notices, January 1975, 13-22.

Appendix A
User's Manual for SPIDER

A.1. Introduction.

This appendix describes the commands and operation of the SPIDER debugging demonstration program. Many of these commands require as an argument (or return as a result) an object reference to some interprocess object such as a process or port. Object references are implemented in SPIDER as a unique string that identifies the object in the kernel. Nothing prevents the user or the program from making up object references, so each must avoid operations that might create references to mythical objects.

A.2. Debugger Commands.

This section details all commands available to the human user at the debugger. A majority of these commands actually create events in the kernel to perform their action. Remember, these are exactly the commands available for execution by demons. The following is the syntax for these commands:

```
<UserCommand> ::= <LocalVariableAssignment> |
          <ProcedureCall> |
          ! <UNIXShellCommand>

<LocalVariableAssignment> ::= <LocalVariable> = <LocalVariable> |
          <LocalVariable> = <Constant> |
          <LocalVariable> = <ProcedureCall>

<LocalVariable> ::= <AnyNumberOfAlphaNumericCharacters>

<Constant> ::= "<AnyNumberOfAlphaNumericCharacters>"

<ProcedureCall> ::= <CommandName> ( <ArgumentList> )

<ArgumentList> ::= <Argument> | <Argument> , <ArgumentList>

<Argument> ::= <LocalVariable> | <Constant> | <ProcedureCall> |
```

(<ArgumentList>)

All commands for non-terminal <CommandName> are listed and described below.

A local variable is a name/value pair local to the debugger. A local variable is created on the first assignment to it. Subsequent assignments overwrite the old value. The system will complain only if you attempt to access the value of a local variable that has never been assigned a value. If <AnyNumberOfAlphaNumericCharacters> is just numeric characters, it is always considered a constant.

All commands return some value, which will be printed out unless assigned to a local variable. If no return value is shown with the command definition below, the command returns "OK". An error message will be returned by a procedure any time there is an error of any sort, regardless of whether the command is shown to return a value. Since errors abort the interpretation of commands, the debugger will never assign an error message to a local variable.

A.2.1. Processes.

This section describes commands that operate on processes. These commands provide for the creation of processes, the examination of the interprocess state of processes and control over the execution of each interprocess event for a process.

process CreateProcess(event-execution) - this command creates a process with the event execution parameter as given. This parameter must be either "Run" or "Suspend". The process will contain no ports, no process variables and no program, and will have a program state of "NoCode. This procedure returns a reference to the process created.

interprocess-step PreviewStep(process) - this command returns a description of the first interprocess event waiting on the process's suspend queue. The process must be suspended and there must be at least one interprocess event awaiting execution for this process or an error will be returned. The interprocess event returned will be the first interprocess event to be executed when the process is resumed (or single-stepped, see "SingleStep" below).

process ProcessOf(port) - this returns a reference to the process containing the given port.

Resume(process) - this changes the event execution parameter of the process to "Run". Any interprocess events previously suspended for this process are put on the end of the kernel's event queue.

interprocess-step SingleStep(process) - this resumes a process for exactly one interprocess event. The process is left suspended during and after the step's execution. The command returns a description of the step executed.

Suspend(process) - this changes the event execution parameter of the process given to "Suspend". The kernel will not execute any interprocess events for a process that is suspended. This will not stop the UNIX process running the process's program unless it executes a system call.

SuspendAll() - this changes the event execution parameter of all processes to "Suspend". (This is provided for panic stops.)

port WaitingPort(process) - if the given process's program state is "ReceiveWait" on a port, this will return a reference to that port, else this will return an error.

A.2.2. Ports.

This section describes commands that operate on ports. These commands provide for the creation of ports, examination of and control over their message queues and control over each port's global name.

Assert(global-name,port) - this asserts the global name given for the port. If the name is already in use for another port, an error is returned. If the port already has a global name, it is quietly written over.

port CreatePortInProcess(process) - this creates a communication port in the given process, returning a reference to the port.

message GetFromPort(port) - this removes and returns the first message in the message queue of the port without causing any border crossings.

port Locate(global-name) - this returns a reference to the port with the global name as given. An error message is returned if no port has the given global name.

integer NumberOfMessagesInPort(port) - this returns the number of messages in the message queue of the port.

message Preview(port) - this returns the first message in the message queue of the port given. The message is not removed from the message queue and no borders are crossed.

PutInPort(port,message) - this inserts the message onto the end of the message queue of the port. The message will not cross any borders to get into the port.

Remove(global-name) - this removes the global name from the port it was associated with.

A.2.3. Messages.

This section describes commands for operating on messages. This includes the creation of messages, creation of name/value pairs, the insertion and removal of name/value pairs from messages and control over the sending and receiving of messages

message AddNVPairToMessage(message,name/value pair) - this adds the name/value pair to the message returning the resulting message. The message passed as an argument is not modified.

message CreateMessage() - this returns an empty list and is intended to be assigned to a local variable. This is more useful to demons than to the human user, but is provided for completeness.

name/value-pair MakeNVPair(name,value) - this returns the name/value pair made out of the name and value given.

message Receive(port) - this command returns the first message in the message queue of the port given. This simulates the action of a Receive from within the program of the process that contains the port. In particular, the message will cross the border of the port and the border of the program. Any program running in the process will not see the message.

message RemoveNVPairFromMessage(message,name) - this removes the name/value pair with the name as given from the message given returning the resulting message. The message passed as an argument is not modified.

Send(port,message) - this sends the message to the given port The message crosses the incoming border of the destination process, crosses the incoming border of the port and is added to the message queue of the destination port. This is a Send without a source process.

SendFrom(process,port,message) - this sends the message to the port from the process. The process given is used as the source process, the process containing the port is destination process. This simulates the action of a Send from the program of the source process. The message will cross the outgoing border of the source program, the outgoing border of the source process, the incoming border of the destination process, the incoming border of the destination port and is then added to the message queue of the destination port.

value ValueOfInMessage(message,name) - this returns the value of the name/value pair with the given name as found in the message.

A.2.4. Programs.

This section describes commands for operating with programs. (These are not the system calls available to programs. System calls are described in section A.3 below.) This includes commands for starting and stopping programs in processes and for examining the program state of the process.

StopProgram(process) - this shuts down the UNIX process running the program of the given process (if it hasn't died already of its own accord) and sets the program state of the process to NoCode. Even if the program of a process dies, no new program can be started in the process until a StopProgram has been performed.

program-state ProgramState(process) - this returns the program state of the process. This will be one of "NoCode", "Running", "Dead", "ReceiveWait" or "SystemCall".

StartProgram(run-file-name,process) - if no program is running in the given process (program state is NoCode) then this establishes run-file-name as the process's program. A UNIX process is spawned to run this code.

A.2.5. Transcripts.

This section describes commands for operating on transcripts. This includes the starting and stopping of transcribing for a process and the playback of a transcript in a process by the kernel.

StartTranscribing(process,output-file-name) - this establishes output-file-name as the transcript file for the process. Every interprocess event executed by the kernel for the given process from this point on will be written into output-file-name.

StartTranscript(transcript-file-name,process) - if there is no program running in the process, this starts a transcript playback with the transcript-file-name as input.

StopTranscribing(process) - this closes the transcript file for the process and stops transcribing.

StopTranscript(process) - this shuts down the transcript replay in the process.

A.2.6. Demons.

This section describes commands for operating on demons. This includes enabling and disabling of demons and control over the continuation of waiting demons.

ContinueDemon(demon) - this continues a waiting demon (see the Wait command in section A.4 below). Nothing is done if the demon is not waiting.

DisableDemon(demon) - this disables the demon. If the demon is not currently a member of any trigger set, it is removed from the list of enabled demons. If the demon is currently a member of a trigger set, it is flagged to destroy itself when re-enabled.

demon EnableDemon(demon-file-name,actual-parameters) - this enables the trigger of demon-file-name and associates the actual-parameters given with the formal parameters of the demon. Note the the command EnableDemon has exactly two arguments. The actual parameters are a list of arguments enclosed in parentheses (see the grammar for commands above). The actual parameters must match in number the formal parameters in the demon.

A.2.7. Miscellaneous Commands.

This section describes some miscellaneous commands. This includes commands to get general status of the system, to execute command files, to examine and control process variables in processes and to operate on lists.

AssertProcessVariableInProcess(process,name/value-pair) - this asserts the name/value pair in the process as a process variable. If a process variable already exists in the process with the same name, its value is changed to the value in the given name/value pair.

process-status StatusOf(process) - this returns a description of the given process. This includes the value of the event execution parameter, the program state of the process and (if a program is running) the name of the program, a list of all ports in the process and the entire contents of their message queues, a list of all process variables and (if transcribing) the name of the transcript file.

list StatusOfAll() - this returns a the StatusOf all processes.

Type(value) - this types out the value given. Mostly useful to demons, this is provided for completeness.

value ValueOfInProcess(process,name) - this returns the value of the process variable in the process with the name as given.

A.3. System Calls.

The syntax of these calls follows the conventions for C procedure calls. The following is the definition in C for each procedure, showing the type of the procedure and the types of the formal parameters. The procedure types "name", "list", "process", "port", "message", "run-file-name", "string", etc., are each a pointer to an array of characters. (Some syntactic liberty has been taken to describe the types of the formal parameters.)

A.3.1. Processes.

This section describes procedures that operate on processes. These procedures provide for the creation of processes, the examination of interprocess state and some control over spawned processes.

int CodeIsAlive(process) - this returns the C constant "true" if and only if the program state in the process given is not "Dead", otherwise "false". The process must be a child process of the caller.

int ReplaceProgram(process, run-file-name) - if the program state of the process is anything but "Dead", this will return "false". If the program state of the process is "Dead", this will establish run-file-name as the new program of the process.

process StartProgram(run-file-name) - this routine creates a new process with event execution parameter "Run" and establishes run-file-name as the program in that process. (The kernel will spawn a UNIX process with the code found in run-file-name.) This also returns the process name of the new process. Note there are only two calls which will respond to the use of this process name: CodeIsAlive and ReplaceProgram. The latter call will only execute when the former returns "false".

A.3.2. Ports.

This section describes procedures that operate on ports. These procedures provide for the creation of ports, examination of and control over their message queues and control over each port's global name.

int Assert(global-name,port) - this asserts global-name as the global name for the port. If the name is already in use for another port, "false" is returned. If the port already has a name, it is quietly written over and the call returns "true". The port must be contained in the process running the program that makes this call.

port CreatePort() - this creates a communication port in the process running the program, returning a reference to the port.

port Locate(global-name) - this returns a reference to the port with the global name as given. The port does not need to be contained in the program's process.

int NumberOfMessagesInPort(port) - this returns the count of the number of messages in the message queue of the port as a string. The port must be contained in the process running the program.

message Preview(port) - this returns the first message of the port given. The message is not removed from the message queue and no borders are crossed. The port must be contained in the process running the program.

Remove(global-name) - this removes the global name from the port it was associated with. This port must be contained in the process running the program.

A.3.3. Messages.

This section describes procedures for operating on messages. This includes the creation of messages, creation of name/value pairs, the insertion and removal of name/value pairs from messages and control over the sending and receiving of messages

message AddNVPairToMessage(message, list) - this adds the name/value pair nvpair to the message returning the resulting message. If the name already exists in the message given, its value is overwritten. The message passed as an argument is not modified.

message CreateMessage() - this returns the empty message "( )".

list MakeNVPair(name, name-or-list) - this returns the name/value pair made out of the name and value given.

message Receive(port) - this returns the first message in the message queue of the port given. This port must be contained in the process running the program. The message will cross the outgoing border of the port and the incoming border of the program. The program state of the process will be "ReceiveWait" until the message is returned to the code after crossing all the borders.

message RemoveNVPairFromMessage(message, name) - this removes the name/value pair with the name as given from the message returning the resulting message. The message passed as an argument is not modified.

int Send(port, message) - this sends the message to the given port. The message crosses the outgoing program border of the program that makes the call, the outgoing process border of the containing process, the incoming process border of the destination process, the incoming port border of the destination port, and is then added to the message queue of the port. This routine returns "false" if the port does not exist or the message is not in the correct form, otherwise it returns "true".

value ValueOfInMessage(message, name) - this returns the value of the name/value pair in the message with the name as given. This value will either be a name or a list.

A.3.4. Miscellaneous Procedures.

This section describes some miscellaneous procedures. This includes procedures to examine and control process variables and to operate on lists.

AssertProcessVariable(name/value-pair) - this asserts the name/value pair given as a process variable in the process running the program. If there exists a process variable with the same name, its value is overwritten.

value ValueOf(name) - this returns the value of the process variable in the process running the program with the name as given.

A.4. Demons.

This section describes interprocess debugging demons. The syntax for demons and demon triggers is as follows:

```
<Demon> ::= <Class> ( <Formal-parameter-list> ) <DemonTrigger>
          begin <DemonCommandList> end

<Class> ::= ReadWrite | ReadOnlyFinal | Replace | NoEvent

<Formal-parameter-list> ::= <LocalVariable> |
          <LocalVariable> , <Formal-parameter-list>

<DemonTrigger> ::= <BooleanExpression>

<BooleanExpression> ::= <ParenthesizedExpression> |
          <ParenthesizedExpression> and <BooleanExpression> |
          <ParenthesizedExpression> or <BooleanExpression>

<ParenthesizedExpression> ::= <Expression> | ( <BooleanExpres  on>
)

<Expression> ::= <VarOrProc> |
          <VarOrProc> = <VarOrProc> |
          <VarOrProc> <> <VarOrProc> |
          <VarOrProc> > <VarOrProc> |
          <VarOrProc> >= <VarOrProc> |
          <VarOrProc> < <VarOrProc> |
          <VarOrProc> <= <VarOrProc>

<VarOrProc> ::= <LocalVariable> | <Constant> |
          <DemonTriggerProcedure>

<DemonCommandList> ::= <UserCommand> |
          <UserCommand> <return> <DemonCommandList>
```

The non-terminal <DemonTriggerProcedure> has the same syntax as <ProcedureCall> in the debugger syntax. The non-terminal <UserCommand> is also defined in the debugger syntax.

The non-terminal <LocalVariable> can take on any name from the demon's execution environment, which includes:

- any name from the formal parameter list of the demon,
- the names EventClass, EventType and EventProcess,
- other names based on the event parameters of the interrupted event.

A.4.1. Demon Trigger Procedures.

The following are some procedures for testing various parameters.

int CodeIsAlive(process) - this returns "true" if and only if the program state of the process is not "Dead", otherwise "false".

program-state ProgramState(process) - this returns the program state of the process, which will be one of "NoCode", "Running", "Dead", "ReceiveWait" or "SystemCall".

event-execution EventExecutionParameter(process) - this returns the value of the event execution parameter of the given process, either "Run" or "Suspend".

port Locate(global-name) - this returns the name of the port with the global name as given.

int NumberOfMessagesInPort(port) - this returns the count of the number of messages in the message queue of the port given.

message Preview(port) - this returns the first message of the port given. The message is not removed from the message queue of the port and no borders are crossed.

process ProcessOf(port) - this returns a reference to the process that contains the given port.

value ValueOfInProcess(process,name) - this returns the value of the process variable in the process with the name as given.

port WaitingPort(process) - if the program state of the process is "ReceiveWait" on a port, this returns the name of that port, else this will return an error.

A.4.2. Demon Commands.

Within the its command list, a demon can execute any command available in the debugger. In addition, the following procedures are provided:

Wait() - this suspends this demon instance until someone (another demon or the user) executes a ContinueDemon command on the name of the waiting demon. The interrupted event for this demon and all demons in the associated trigger set will be suspended until this demon is resumed.

ReplaceEventWithReturnResult(result) - this causes the demon to stop executing and specifies to the kernel that the interrupted event is to be aborted and the result given with the call is to be used to create a event. This command can only be called by a demon of class Replace. If the interrupted event is a system call, the result is used to create an event of type "ReturnResult". If the interrupted event is of class "MessageTransmission", and the result is "nil", the event is aborted completely. If the interrupted event is of class "MessageTransmission", and the result is a message, the result will replace the message in the interrupted event to make a new event. This never returns to the demon.

DemonDone() - this command is always executed by default if the demon command list is exhausted, but can be called by the demon. This causes the demon to stop executing and never returns.

Appendix B
Event Types

B.1. Introduction.

This appendix describes all the event types of our model. Each of these types are implemented in the SPIDER demonstration program and use the various parameter names listed below. The event types are listed below by the event class that they occur with. Also included in this description are the names corresponding to the event parameters (for system call and message transmission events) that will be included in demon execution environments if the event is interrupted to fire a demon.

B.2. Debugger Commands.

The event types of event class "DebuggingCommands" are caused by commands executed by the debugger either under the control of the human user or a demon. Each of these event types has the same name as the corresponding debugger command. These commands are listed in section A.2 and A.4 of Appendix A. For each type, the event parameters are the arguments to the call (if there are any) or null. (We are not naming the event parameters for the debugger commands here, since we do not use the debugger to watch its own events.)

B.3. System Calls.

The event types described in this section are caused by system calls by programs. System calls are described in section A.3 of Appendix A.

AddNVPairToMessage - this is caused by the system call AddNVPairToMessage. The event parameters are a message and a name/value pair, named respectively "Message" and "NameValuePair".

Assert - this is caused by the system call Assert. The event parameters are an arbitrary string and a port reference, named respectively "GlobalName" and "Port".

AssertProcessVariable - this is caused by the system call AssertProcessVariable. The event parameter is a name/value pair called "NameValuePair".

CodeIsAlive - this is caused by the system call CodeIsAlive. The event parameter is a process reference and is named "Process".

CreateMessage - this is caused by the system call CreateMessage. There are no event parameters.

CreatePort - this is caused by the system call CreatePort. There are no event parameters.

Locate - this is caused by the system call Locate. The event parameter is an arbitrary string called "GlobalName".

MakeNVPair - this is caused by the system call MakeNVPair. The event parameters are a name and a value, called respectively "Name" and "Value".

NumberOfMessagesInPort - this is caused by the system call NumberOfMessagesInPort. The event parameter is a port reference called "Port".

Preview - this is caused by the system call Preview. The event parameter is a port reference called "Port".

Receive - this is caused by the system call Receive. The event parameter is a port reference called "Port".

ReceiveWait - this is caused by the event type "Receive". The event parameter is a port reference called "Port".

ReturnResult - this is caused by the execution of system calls. The event parameter is a result called "Result".

Remove - this is caused by the system call Remove. The event parameter is an arbitrary string called "GlobalName".

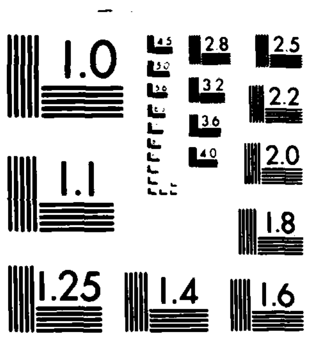
RemoveNVPairFromMessage - this is caused by the system call RemoveNVPairFromMessage. The event parameters are a message and a name, called respectively "Message" and "Name".

ReplaceProgram - this is caused by the system call ReplaceProgram. The event parameters are a process reference and the name of an executable program, named respectively "Process" and

AD-A127 117 DEBUGGING TECHNIQUES FOR COMMUNICATING LOOSELY-COUPLED PROCESSES(U) ROCHESTER UNIV NY DEPT OF COMPUTER SCIENCE E T SMITH DEC 81 TR-100 N00014-78-C-0164 2/2

UNCLASSIFIED F/G 9/2 NL

END
DATE
FILMED
5-83
DTIC


1.0
2.8
2.5
3.2
2.2
3.6
4.0
2.0
1.1
1.8
1.25
1.4
1.6

"ProgramName".

Send - this is caused by the system call Send. The event parameters are a port reference and a message, named respectively "DestinationPort" and "Message".

StartProgram - this is caused by the system call StartProgram. The event parameter is the name of an executable program and is called "ProgramName".

ValueOf - this is caused by the system call ValueOf. The event parameter is a name called "Name".

ValueOfInMessage - this is caused by the system call ValueOfInMessage. The event parameters are a message and a name, called respectively "Message" and "Name".

B.4. Message Transmission.

The event types of class "MessageTransmission" are related to the transmission of message across object borders. Each message transmission event has two event parameters, the message being delivered and the destination port. These are named respectively, "Message" and "DestinationPort". The six event types of this class are: InComingProcessBorder, OutGoingProcessBorder, InComingProgramBorder, OutGoingProgramBorder, InComingPortBorder and OutGoingPortBorder.