pdb: A Network Oriented Symbolic Debugger

Paul Maybee

Solbourne Computer, Inc.

ABSTRACT

pdb is a symbolic debugger designed to support program debugging needs on a network of workstations. **pdb** runs under the X Window System[1], reads object files and symbol tables generated by existing (Sun[2]) compilers, debugs both local and remote processes, and works on multiple hardware types. **pdb** was constructed using object oriented programming techniques to improve its portability to new machine types, languages, and symbol table formats. **pdb** supports debugging at both the source and assembly level. It is entirely mouse and menu driven, with no command language or syntax outside of that required to formulate expressions in the source language.

1. Introduction

A workstation is a powerful desk side or desk top computer with a bitmapped graphics display and several megabytes of memory. It runs some flavor of UNIX[3], and may or may not have a local disk. A network of workstations may contain machines of only one type or, more commonly, machines of several different architectures. Often machines other than workstations are on the same network, for example, timesharing machines, scientific computers, or special purpose hardware like hypercube, database and Connection machines. The computers on the network typically share file systems, access to printers, and are usually under a single administrative control. Supporting debugging needs on such a heterogenous network is the goal of the **pdb** project.

The debugger takes advantage of the graphical capabilities of the workstation. The X window system was selected for the interface since it is perceived as the de facto standard workstation windowing system. Unlike other debuggers, **pdb** does not have a command language; there are no commands and no terminal-like windows to type commands into. The interaction with the debugger involves selecting source lines, expressions, and stack frames, popping up menus, and moving around icons.

pdb allows debugging of processes anywhere on the network. Debugging a remote processes has always been possible using a remote login to the host and then debugging normally. **pdb** does not require the remote login session. Debugging ''anywhere in the network'' on a network containing machines of differing architectures requires porting. **pdb** was designed for portability. The different aspects of porting were identified and an attempt was made to isolate the dependent code in specific objects. **pdb** contains objects that isolate the window system, architecture, language, object file format, core file format, and operating system interface. **pdb** has already been successfully ported to different architectures, operating system interfaces and core file formats.

Computers on a network cooperate to perform a variety of tasks. Network cooperation involves communication between processes residing on different machines. Even programs that

[1]**X Window System** is a trademark of MIT

[2]**Sun, Sun-3 and Sun-4** are trademarks of Sun Microsystems, Inc.

[3]UNIX is a registered trademark of AT&T

do not explicitly access remote machines may do so implicitly, because the network has made access invisible to the application. **pdb** supports debugging such multi-process programs, even when the machines are of different architectures.

Another program that **pdb** has been designed to help debug is the UNIX kernel. **pdb** supports operating system development by providing debugging assistance from the bootrom level to the application level. **pdb** communicates with system simulators, kernel co-resident debugging agents, and running kernels to support operating system debugging.

2. Architecture

pdb is composed of two processes. The Display Manager (DM) maintains the user interface and typically (but not necessarily) executes on the user's local host. The Remote Agent (RA) resides at the site of the process being debugged and provides process query and control services to the DM. The DM allows the user to simultaneously debug multiple processes running on different machines of dissimilar architectures. One RA executes for each process being debugged.

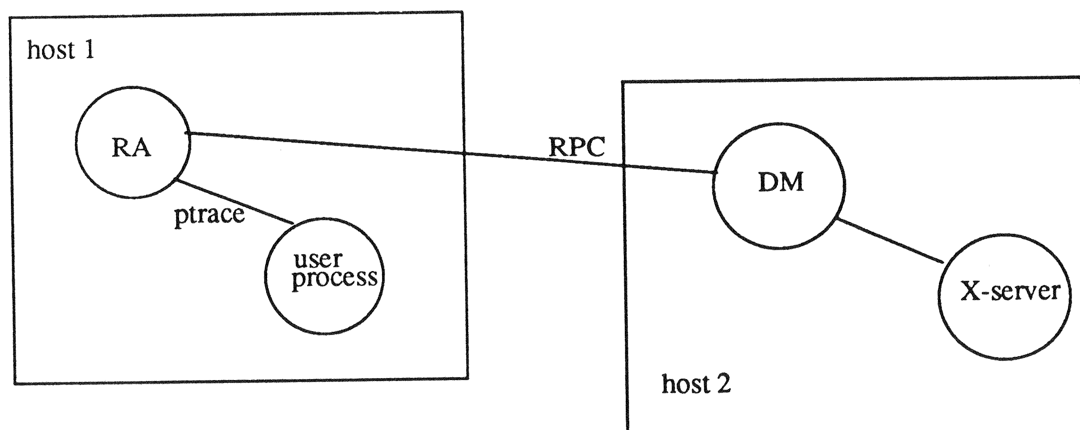


Figure 1: **pdb** architecture

pdb is written in C++ (except for C generated by a communication protocol compiler). The object oriented paradigm supported by C++ has proven invaluable in the construction of the debugger (see [Carg87] for a discussion of this topic in reference to the Pi debugger).

The DM is divided into two sets of objects:

(1) The first to implement the user interface.

(2) The second to implement the process interface.

The user interface is made up of multiple windows, similar to Pi [Carg85, Carg86], Saber-C [Kauf88] or LightspeedC[4] [THINK88], in the X environment. The process interface provides a process control abstraction for implementing debugging transactions. The user interface translates user input into combinations of calls to the process interface. The DM is independent of the architecture of the host on which the RA is executing. When a DM and RA begin communication the RA informs the DM of any required host specific information (e.g., register names

[4]LightspeedC is a registered trademark of Lightspeed, Inc.

and types). The DM can then construct menus and tables for presenting this information to the user.

The RA process has objects for process control, machine description, code instrumentation, low level operating system interface, core file access, expression evaluation and symbol table maintenance. The process control object implements the remote procedures used by the DM. The machine description object knows the types, sizes, and names of the registers, as well as function calling conventions. The code instrumentation object handles the setting and lifting of break and watch points. The low level interface object knows the operating system specific aspects of process control (e.g., **ptrace** or **/proc**). The core file object knows the file format and address translations necessary for reading a core image. The expression evaluator computes the values of expressions according to the language of the program being debugged. The symbol table object reads program symbol tables and builds a **pdb**-internal representation.

3. User Interface

pdb's user interface is composed of several tools. Each tool has its own window and displays information about a single process. When multiple processes are being debugged, multiple sets of windows are present. All windows referring to a particular process can be opened, closed, or deleted together (as well as separately). The set of tools includes a Process Window, Source and Assembly Displays, Expression Evaluator, and a Breakpoint Editor.

The Process Window (Figure 2) is divided into three regions: a control panel, a stack traceback, and a message log. The control panel contains pull down menus for file specification, processes control and for requesting additional tools. The stack traceback is updated every time the debugged process stops. It contains the current process stack or, for a core file, the stack at process termination. In figure 2 the stack traceback window shows four levels of calls in the **csh**, from *main* to *readc*. The solid arrow pointing to *readc* indicates that this is the current frame of focus for the debugger. All the tools will tend to display information for the frame under focus. The focal frame is changed by clicking on the line displaying the frame of interest. The message log window contains any messages that were logged by the debugger on behalf of the process. In figure 2 the debugger has displayed three lines related to reading the symbol table. Only about a third of the symbols were processed on reading the table because **pdb** does a lazy evaluation of the symbol table, i.e., it only reads as much of the table as is necessary to satisfy the current requests. There is a status line at the bottom of each tool. The status line for the Process Window indicates whether the process is executing or stopped, and if stopped, why. For this example the process stopped because it received a SIGSTOP signal.

The Source Display (Figure 3) has a control panel and a window with annotated program source. The control panel contains two buttons and three glyphs representing different types of code instrumentation: breakpoints (stop sign), conditional breakpoints (yield sign), and tracepoints (check mark). The annotations to the source text consist of these instrumentation marks as well as execution focus arrows; a solid arrow for the current point of execution and hollow arrows for call points down the stack. A breakpoint (tracepoint, conditional breakpoint) can be set by pulling a stop sign glyph (check mark, yield sign) from the control panel to a line of source. Process stepping and continuing is controlled by menu/button selections. The first button in the control panel, which in figure 3 is set to perform a continue operation, can be switched to any of the actions contained in the popup menu displayed. The "Control" button contains options for viewing other source text and for specifying directory search paths. Additional actions are available through clicking in the source window. Clicking on a variable causes its

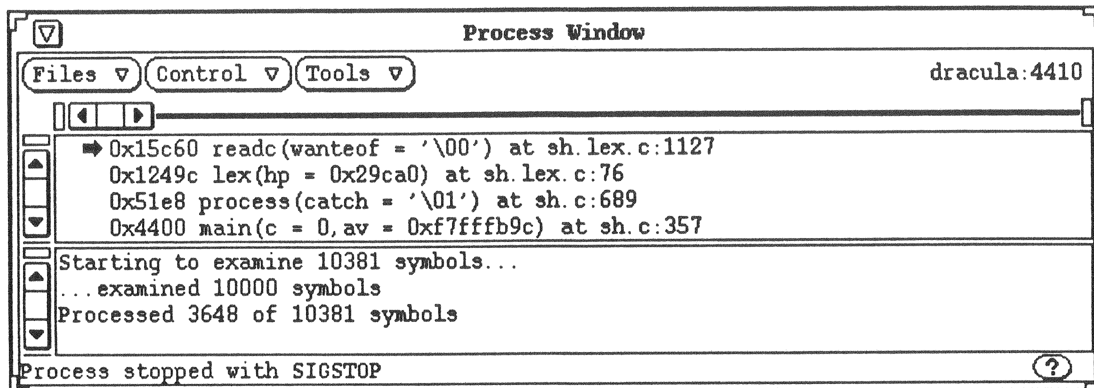


Figure 2: Process Window on **csh**

value to be printed, clicking on a function name switches the source window to display that function, and dragging the solid arrow to a new source line causes a *once-only* breakpoint to be set on that line and the process to be continued. The Assembly Display (Figure 4) is identical to the Source Display except that annotated disassembly is displayed rather than source.

The Evaluator (Figure 5) displays expressions in some program context. Expressions may be typed in directly to the Evaluator, selected from menus of local, global, and static variables,

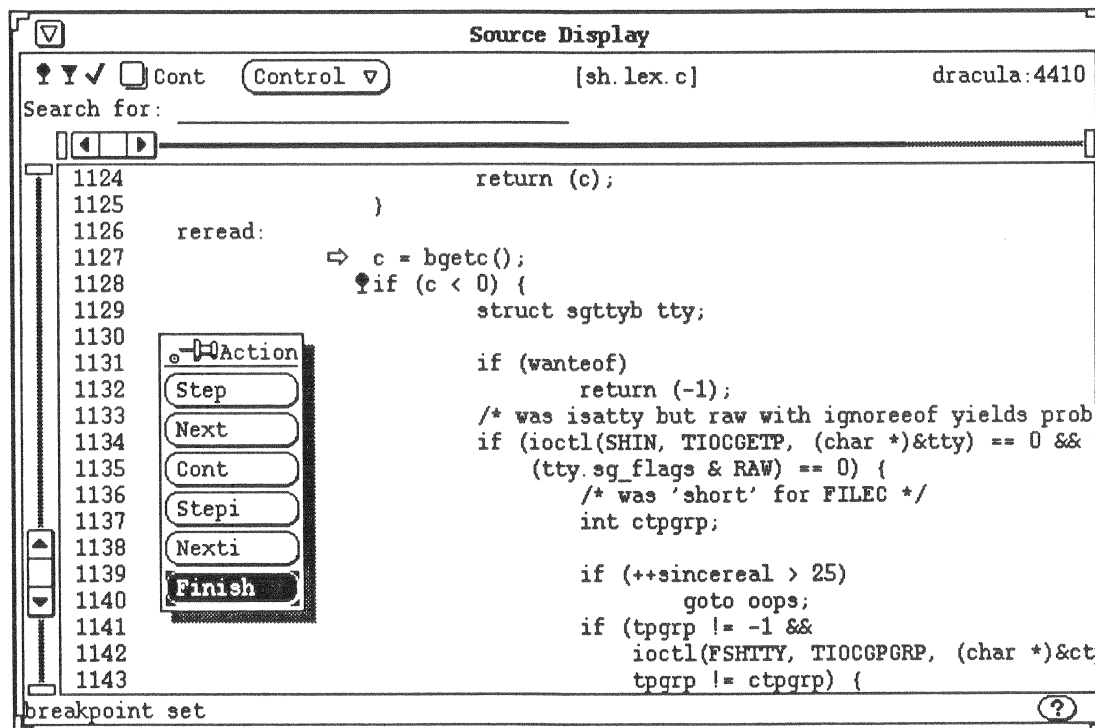


Figure 3: Source Display of **csh**

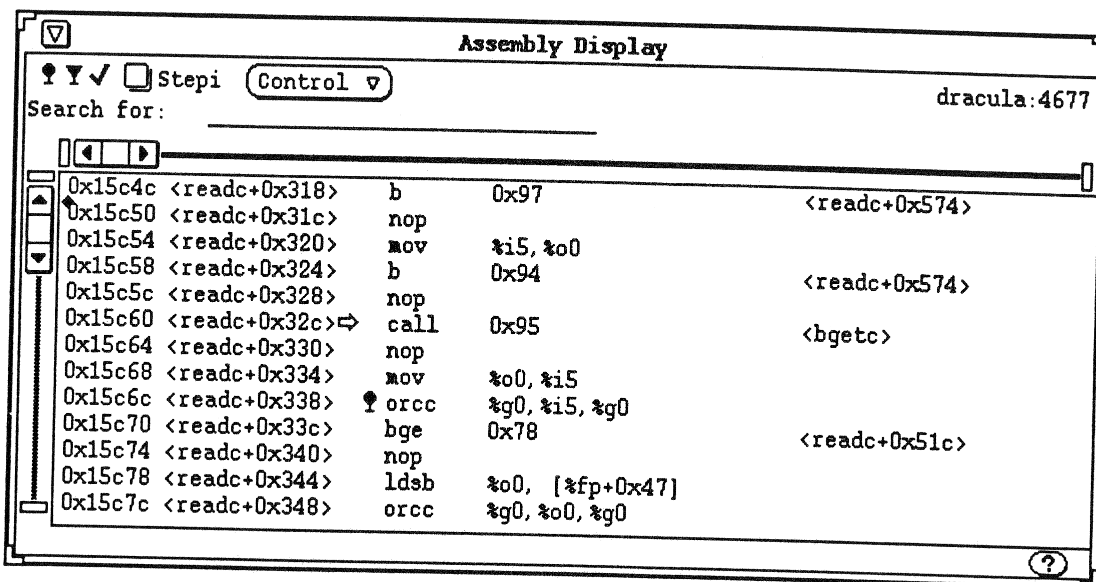


Figure 4: Assembly Display of **csh**

generated from previous expressions (e.g., by prefacing an existing expression with *, &, or (*type*)), or selected from source text. The syntax of expressions is that of the source language (only C currently). Expressions are always evaluated in the current context of the Evaluator. When the Evaluator context changes, in response to program execution or direct user specification, then the expressions for previous contexts are no longer visible. If execution returns to a previous context then the expressions for that context are redisplayed. If no context can be found (e.g., because the process is stopped in a module compiled without symbol table information) then a global context is selected. Expressions are re-evaluated every time that execution of the program stops.

The black arrow is used in the Evaluator to indicate the expression being focused on. When the menu for the tool is popped up the actions on it refer to this expression. Like the Source Display, the Evaluator also contains glyphs in the control panel that may be pulled down. The lock glyph, when placed on an expression, causes that expression to not be automatically re-evaluated each time execution stops. The check mark glyph indicates that the value for the expression should be written to the log every time it is re-evaluated. Any expression that contains an assignment operator, or invokes a program function will automatically be displayed with a lock.

The Breakpoints window (Figure 6) displays a list of all current program instrumentation. Elements of this list can be edited or deleted and new instrumentation can be added. If a breakpoint is deleted then any stop sign glyphs representing it in source and disassembly disappears. Similarly, if a new breakpoint is added then a stop sign will appear if corresponding source (disassembly) is being displayed.

pdb's user interface is built using a C++ X toolkit called OI (object interface) [Aitk89].

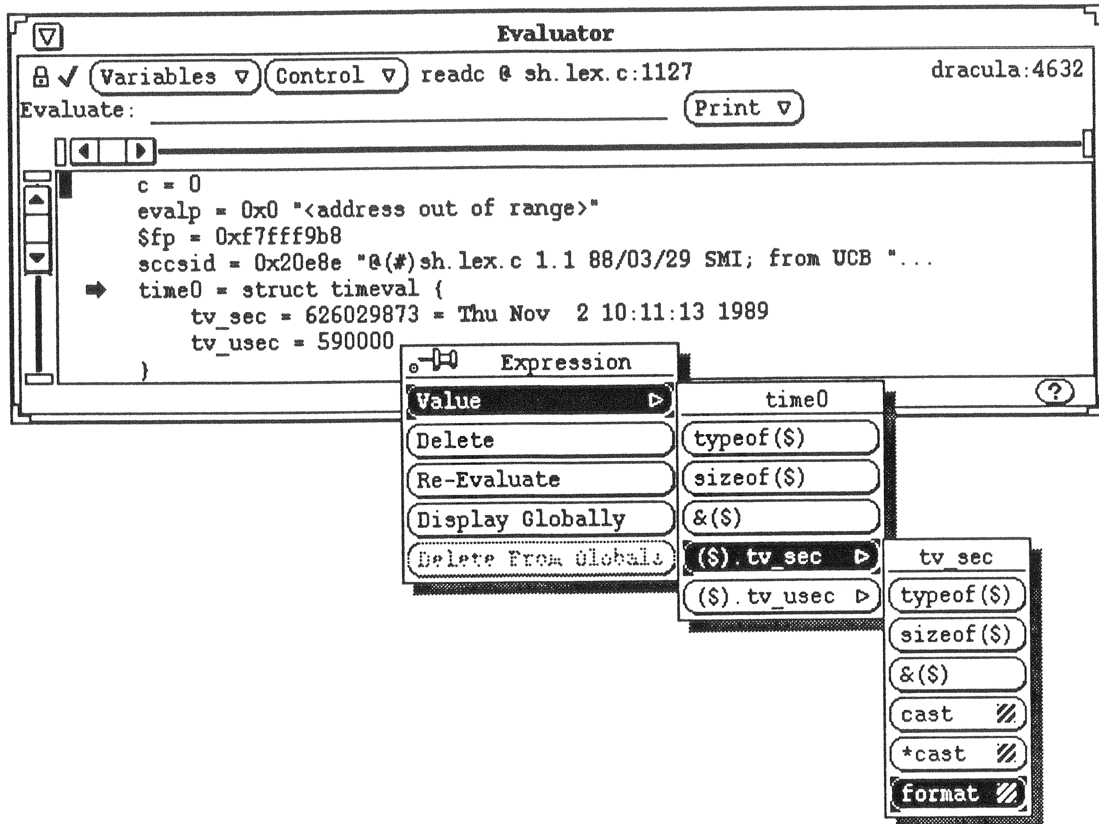


Figure 5: Evaluator displaying **csh** expressions.

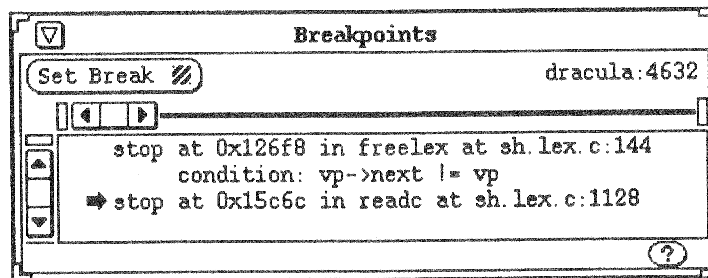


Figure 6: Breakpoints in **csh**.

The OI implements the look-and-feel of both OpenLook[5] and Motif[6]. The desired look-and-feel is chosen based on the state of a runtime flag. All pictures of **pdb** for this paper were taken using the OpenLook setting.

[5]OpenLook is a trademark of AT&T

[6]Motif is a trademark of the Open Software Foundation

4. Communication

The Display Manager and Remote Agent communicate via remote procedure call (Sun RPC) over TCP/IP connections. The DM causes the RA to be created via the internet daemon and then acts as its client. However the relationship between the two **pdb** processes is not strictly client-server. The RA sometimes needs to inform the DM of the debugged process's activity. When this is necessary the RA makes a call back to the DM bearing the news (e.g., the process has hit a breakpoint, the symbol table is loaded, or the process has executed a traced instruction). These calls are made asynchronously (i.e., no response is expected or waited for) over a separate TCP/IP link.

The set of calls that the DM makes is similar to a standard debugger command set (Table 1). There are six groups of calls: process and object file selection calls, process query and control calls, symbol table query calls, expression evaluation calls, front end configuration calls, and low-level access calls. The process and object file selection calls tell the RA which processes to debug and which files to search for symbol table information. The process query and control calls perform code instrumentation and execution tasks as well as process image querying for stack state and disassembly. The symbol table query calls fetch lists of files, functions, types and variables, and provide a source-to-object location mapping. The expression evaluation calls compute values of expressions in context. The front end configuration calls, at this point, are limited to providing register names and types. Finally, the low-level access calls allow for the transfer of raw (i.e., uninterpreted) data transfer between RA and DM.

The Remote Agent calls (Table 2) indicate the detection of an event by the agent. These are generated in situations for which the DM does not (or cannot) specifically wait. The DM does not wait for the symbol table to load since this may take a long time. The DM cannot block waiting for a process to stop because some processes never stop without input from the user (e.g., sending a signal to the running process). Since these calls are delivered using a reliable communications protocol, and since they do not require a response, the DM does not respond and the RA does not wait for a response. This asynchrony allows for notification of events (especially trace messages) with minimum overhead.

A typical debugging session might proceed as follows. A **pdb** user begins debugging by executing the **pdb** command (the DM process). A process to debug is then specified using the process's host name and pid. If the user wishes to debug a program from the beginning, rather than intercept it while it is running, then the **pause** command can be used. This command starts the process, but suspends it before it executes its first instruction. At this point **pdb** can start to debug. **pdb** makes an ATTACH call to the process's host machine. The RPC number for **pdb** is registered with that host's internet daemon (*inetd*). The inetd receives the request and forks a RA process that handles the call. The RA issues an attach request to the process, responds to the Display Manager's ATTACH call, and waits for the process to stop. When the process stops the RA gains control and makes a PROCESSSTOP call to the DM.

The DM then issues a USEOBJECT call containing the name of a file to be read for symbol table information. The RA responds to this call after it has successfully opened the file, but before it has processed the symbols therein. When it has completed it will notify the DM via a SYMTABLEREAD call. For large images symbol table reading takes quite a while and the user often gets anxious about the progress being made. For this reason the RA notifies the DM after every 10000 symbols have been processed with a TRACEMESSAGE that the DM then displays to the user (see Figure 2 above).

Call	Definition
	Process and Object Selection
ATTACH	debug a process
REATTACH	debug a different process
USEOBJECT	use symbols from object file
DETACH	cease debugging a process
	Process Query and Control
TRACEBACK	get a stack trace
SETBREAK	set a breakpoint
DELETEBREAK	delete a breakpoint
SETWATCH	watch a memory location
DELETEWATCH	stop watching
ADVANCE	execute the process
SIGNAL	send a signal to process
HANDLE	handle signals
FETCHASM	get disassembly
	Symbol Table Query
FETCHFILES	get list of files
FETCHFUNCS	get list of functions
FETCHTYPES	get list of types
FETCHVARS	get list of variables
FILLLOCATION	get source/object location info
	Expression Evaluation
EVAL	evaluate an expression
EVAL_RES	get an EVAL result
	Client Configuration
REGDESC	get description of registers
	Low-Level Access
FETCHRAWMEM	fetch raw memory
PUTRAWMEM	write raw data to memory
FETCHRAWREG	fetch raw registers
PUTRAWREG	write raw data to registers

Table 1: Display Manager RPC interface.

Call	Definition
PROCESSSTOP	a process has stopped
EXPREVAL	an expression value is available
SYMTABLEREAD	symbol table reading is complete
TRACEMESSAGE	trace message

Table 2: Remote Agent RPC interface.

All the remaining calls, except ADVANCE and EVAL, execute as normal RPC calls. The call is made by the DM, the RA processes the request, returns a result, and waits for the next call. ADVANCE also executes normally, but the process is executing following its completion. Thus the next activity performed by the RA may be either to service another RPC call or to notify the DM that the process has stopped executing again (PROCESSSTOP). The EVAL call also executes as a normal RPC call, most of the time. The exception occurs when the evaluation of the expression causes the process to execute. The happens when the expression contains a user function call. In this case EVAL returns after the process has begun to execute. When the expression evaluation is complete (i.e., after the process has stopped) the RA informs the DM with an EXPREVAL. The RA then uses an EVAL_RES call to retrieve the result.

The ADVANCE and EVAL calls do not execute synchronously since they rely on user process execution, and thus may take an arbitrarily long time to complete. Because the DM does not wait the user interface is active during process execution, allowing the user to continue to perform actions - stopping execution, for example.

The **pdb**-internal form of the symbol table is generated lazily from the object file. The symbol table query calls allow the front end to generate menus of file, functions, types and variables. **pdb** requests the file, function and global variable lists immediately upon starting up, thus the symbol table is initially processed to the extent that these lists can be created. Lists of types and local variables are requested on an as-needed basis, i.e., as the user demands information about the different local contexts of the program. The object file is processed for this information in response to these requests. In a typical debugging session most of the local contexts may never have to be generated [Carg87].

Architectural and implementation specific information is supplied to the DM by the RA via the configuration calls. In the current implementation only a list of register names is supplied. The DM uses the list to generate the Evaluator's registers menu. This group of calls will be expanded (or perhaps a more general specification will be supplied for a single call) to provide more detailed configuration information describing the specific RA.

The low-level access calls read (write) raw data from (to) the process being debugged. The data are transferred by the RA as uninterpreted bytes. These calls are currently unused but may be useful in the context of a raw memory viewer/editor. **pdb** does not now have such a viewer, but one is planned for the future.

5. Operating System Debugging

Software simulators for new hardware are typically ready months in advance of the hardware. Thus, long before new hardware is ready, the process of porting system software can begin. **pdb** will allow the developers to debug the operating system from the first rom instruction executed on the simulator. Later, when hardware is ready, **pdb** can help in debugging the

running system through a stand-alone debugging agent interface and it can help in postmortem analysis of failure by reading kernel core dumps. From the first simulation of the bootrom code to modifying the **csh**, **pdb** provides a uniform environment for debugging support.

pdb can be used to debug running processes, core files, running kernels, kernel core files, and programs running in simulation - all from the same RA image. The **pdb** user selects the object to debug and the DM then makes an ATTACH call for that object. The RA contains a base class, *process*, and derived classes for each of the possible objects to debug. The derived class for running processes uses **ptrace** for access and control of the UNIX process image. The core file class does not allow any execution or breakpoint operations; access to symbols and stack state is done by reading the core file. A running kernel is debugged over either a SLIP or UDP conversation with the stand-alone debugger executing on a remote machine [Rud89]. Kernel core images are accessed via the kernel virtual memory utilities supplied under Sun OS (*libkvm*). Finally, the derived class for simulator debugging communicates over a TCP/IP connection with a hardware simulator. Both the stand-alone agent and the hardware simulator implement a **ptrace** interface.

The Solbourne operating system supports multiple processors executing a common kernel in which each processor maintains its own stack. **pdb** displays these multiple stacks as preliminary support for debugging shared address space parallel programs. When debugging a multi-stack program, **pdb**'s Process Window displays a "Next Stack" button. Selecting the button will rotate the tool through the stacks. This mechanism is usable for the small numbers involved with a multi-processor kernel, but is not sufficient for debugging programs with large numbers of stacks. Plans for the future include a better method for selecting the stack to display and for displaying multiple stacks simultaneously.

6. Current State and Future Directions

pdb's goal was to meet the debugging needs in the network environment. Its current implementation meets these goals in only a limited fashion. Processes anywhere in the network may be debugged (if the host belongs to the class of supported machines), X windows is used to construct the user interface, and a informative debugger interface has been made available. Development is continuing in three important areas: multi-process debugging, multi-language support, and development of a "standard" RPC interface.

The current multi-process support consists of the ability to have windows on multiple processes open at the same time and to debug a restricted class of multi-stack programs. There is no support for debugging forked processes or for controlling or querying multiple processes without switching focus between separate windows. Future versions of **pdb** will include support for debugging forked processes as well as breakpoints that stop more than one process, monitoring of interprocess communication, and collecting and filtering trace output in search of anomalous program behavior.

All users do not program in C. Several languages are typically in use by programmers on any sizable network. More complete C++ debugging support will be added as a first step in making **pdb** a multi-language debugger. Support for other languages (e.g., FORTRAN, Pascal) will then be added as the need arises.

pdb's portability to new operating systems and architectures involves deriving new objects for those portions of the interface that have changed. With the exception of the code for the disassembler, the Sun-3 specific portions of the RA comprise about 700 lines of code, the Sun-4

specific code totals about 1100 lines. Of course both of these implementations use nearly identical operating systems (SunOS 4.0 and its derivatives), so the differences in that aspect are small. The only architectural differences that the RA needed to inform the DM of was the names of registers. Future ports may require expanding the client configuration information to include:

high or low byte word addressing
list of signals
location watching capabilities
list of unimplemented RPC calls

The last of these indicates that the RA does not have to implement the complete interface. The DM is designed to handle failure of any RPC call. Failures of some calls are catastrophic, however others only result in the inability to generate menus, disassembly, etc. A partial implementation of the RPC interface can still be useful. For instance, the low-level access calls, coupled with ADVANCE and ATTACH, are sufficient for implementing a remote debugger. However the DM process then needs to contain all the architecture-specific code for interpreting the process and object. This functionality was found to be very useful in the early stages of **pdb**'s development.

A completely new RA implementation requires that a small set of RPC calls (currently 25) be constructed. We feel that the interface is already too complex and are working towards a smaller, cleaner interface for basic debugger functionality and a general mechanism for allowing implementations to provide more complex or system dependent features. The overall goal of supporting network debugging cannot be considered complete without the adoption of such a remote agent interface "standard."

7. Acknowledgements

This work would not have been possible without the help of Karen Meyer-Arendt, Andrew Gerber and Andrew Rudoff. Karen conspired in the initial design of the project and constructed the symbol table and expression evaluation modules. Andrew Gerber is responsible for the current state of the user interface, including the port to OpenLook. Andrew Rudoff's suggestions, comments, and complaints were a continuing source of inspiration and ideas.

References

[Aitk89]
Aitken, Gary, *OI: A Model Extendable C++ Toolkit for X Windows*, in submission.

[Carg85]
Cargill, Thomas A., *Implementation of the Blit Debugger*, Software--Practice and Experience, Vol. 15(2), 153-168 (February 1985)

[Carg86]
Cargill, T.A., *The Feel of Pi*, Proceedings Winter 1986 USENIX Technical Conference, Denver, CO (January 15-17, 1986)

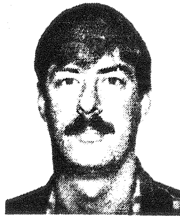
[Carg87]
Cargill, T.A., *Pi: A Case Study in Object-Oriented Programming*, C++ Workshop Proceedings, Santa Fe, NM, USENIX Assoc. (Nov 9-10, 1987)

[Kauf88]
Kaufer, Stephen, Russel Lopex, and Sesha Pratap, *Saber-C An Interpreter-based Programming Environment for the C Language*, Proceedings of the Summer 1988 USENIX Conference, San Francisco, CA, (June 20-24, 1988)

[Netw88]
Network Programming, Sun Microsystems, Inc., Revision 50 (19 January 1988)

[Rudo89]
Rudoff, Andrew, *Remote Debugging Kernels on Multiprocessor Systems*, in preparation.

[THINK88]
THINK'S LightspeedC User's Manual, Symantec Corporation, 1988



Paul Maybee
Solbourne

Paul Maybee received his B.A. in Mathematics and M.S. in Computer Science from the University of Colorado, Boulder. He is currently an engineer at Solbourne Computer, Inc. In his copious free time he is also working on his Ph.D. at the University of Colorado.