Implementing a Mach Debugger for Multithreaded Applications

Deborah Caswell and David Black

November 1989
CMU-CS-89-154

Deborah Caswell
Hewlett Packard Laboratories
Palo Alto, CA

David Black
Carnegie Mellon University
Pittsburgh, PA

To appear in the Conference Proceedings of *Winter 1990 USENIX Technical Conference and Exhibition*, Washington, DC, January, 1990.

Abstract

Multiple threads of control add new challenges to the task of application debugging, and require the development of new debuggers to meet these challenges. This paper describes the design and implementation of modifications to an existing debugger (gdb) for debugging multithreaded applications under the Mach operating system. It also describes the operating system facilities that support it. Although certain implementation details are specific to Mach, the underlying design principles are applicable to other systems that support threads in a Unix compatible environment.

This research was performed in part at Hewlett Packard Laboratories and supported by Hewlett Packard Company.

This research was sponsored by the Defense Advanced Research Projects Agency (DOD), monitored by the Space and Naval Warfare Systems Command under Contract N00039-87-C-0251.

1. Introduction

Applications that employ multiple threads of control add new challenges to the debugging problem that exceed the capabilities of most existing debuggers for single threaded applications. Not only do the debuggers lack the logic and data structures needed to deal with multiple threads in the target application, but often the operating system primitives they depend on (e.g. *ptrace*) are inadequate to the task. While a good operating system designer includes new primitives to support multithreaded debugging, it is up to the debugger implementors to put these primitives to good use. This paper describes one such effort in this area; the modification of an existing debugger for single threaded Unix[1] applications to support debugging of multithreaded applications under the Mach operating system. Although the operating system primitives are specific to Mach, the underlying design principles are applicable to any debugger for applications with multiple threads of control in a Unix-compatible environment.

This paper begins with an introduction to Unix debuggers for single threaded applications and the problems involved in adding support for multithreaded applications. We then address these problems and describe our solutions for the Mach debugger that we have implemented. We conclude with a short discussion of possible extensions to our work and its relation to other work in the field.

2. Unix Debuggers and Debugging-Related System Calls

On a Unix system, the debugger and an application being debugged are separate user processes that communicate via kernel-provided services, primarily the various options of the *ptrace* system call. In addition a debugged process undergoes special interactions with the Unix signal and process control mechanisms to facilitate debugging. Hence the implementation of debugging functionality is split across the user-kernel boundary and includes logic in both the debugger and the operating system.

2.1. *ptrace*

The *ptrace* system call implements most of the functionality required by a debugger to interact with the application being debugged. The various options to *ptrace* implement:

- Access to the application's registers and memory.
- Read access to certain kernel information about the application (e.g. the command it was invoked with).
- Continue or single step the application (with the option of delivering a signal). Also kill (terminate) the application.

Since Unix applications do not have access to the state or address space of another process, even when running in kernel mode, *ptrace* is implemented by having the debugged application perform the desired action on itself (e.g. if a register is to be read, the debugged application reads the register and returns that value to the debugger process via an internal kernel communication mechanism); the kernel code that the application executes to do this is in the function *procxmt*. The debugger process sleeps waiting for the application to wake it when the action has been completed.

[1] Unix is a registered trademark of AT&T Bell Laboratories

ptrace also includes a hook that allows the debugged application to identify itself to the kernel and request special treatment as a result. Typically a debugger forks a child process which calls this hook ('Please trace me') before *exec*'ing the application to be debugged; this hook sets the trace flag (STRC) for the debugged process. A process with this flag set is referred to as a "traced" process.

2.2. Signals

Signals form an important part of the interface between the debugger and the debugged application. Signals are caused both by external events involving the application and by exceptions (both hardware and software) occurring in the application itself. When a signal is delivered to a traced process, the process stops inside the signal code so that it can carry out actions requested by the debugger through the *ptrace* interface. The process remains in this state until a continue, single step, or kill action is requested; the first two allow it to continue execution, and the latter terminates it immediately. The signal itself is cancelled unless the debugger specifically directs that it be taken; the debugger may also substitute another signal if desired.

The two most common uses of signals during debugging are breakpoint/trace detection and interruption of a looping application. Both breakpoints and trace traps generate the signal SIGTRAP; the debugger will usually allow this signal to be cancelled after gaining control via *ptrace*. Similarly, interruption characters at the keyboard also generate signals (e.g. control-C generates SIGINT). A user can take advantage of this to regain debugger control of a runaway application; typing control-C on the terminal causes the application to stop in the signal code and the debugger to regain control. The only remaining detail to be explained is how the debugger detects that the application has stopped.

2.3. *wait*

wait is the most common mechanism used by debuggers to determine that a debugged application has stopped[2]. The *wait* system call is normally used to wait for an exited process and return its exit code, but *wait* also returns if it finds a traced process that has stopped in the signal code. Thus a debugger can use wait to simultaneously detect the two occurrences that it is most interested in:

1. The application has taken a signal (e.g. hit a breakpoint) and stopped.
2. The application has exited.

The status returned by *wait* indicates whether the application has stopped or exited, and in the former case also returns the signal that caused it to stop. Once *wait* has indicated that the debugged application has stopped, the debugger can safely use *ptrace* to manipulate the application.

2.4. Breakpoints and Single Stepping

The actual mechanisms used to implement breakpoints and single stepping are hardware dependent, but the software interface to them (via *ptrace*) is very similar across most machines. A breakpoint is usually implemented by a special machine instruction which causes a trap that the kernel immediately identifies

[2]Used by virtually all BSD debuggers; some System 5 debuggers may use the SIGCLD signal for this purpose.

as a breakpoint trap. Similarly, single stepping is often implemented by a 'trace bit' which causes a trace trap at the end of the next instruction. In both cases the kernel identifies the trap and generates the SIGTRAP signal; this notifies the debugger via the kernel mechanisms discussed above.

There is one subtlety involved in actually using breakpoints. Since it is not possible to insert the special breakpoint instruction between adjacent instructions, it is necessary to replace part of the application's machine code[3]. As a result, it is necessary to execute the replaced code when continuing from a breakpoint; this is usually done by replacing the original code, single stepping, and replacing the breakpoint before continuing the application.

3. Multiple Threads and Debugging

Mach splits the familiar Unix **process** abstraction into the **task** and **thread** abstractions [9]. A **task** is an address space along with associated communication rights; it is a passive entity that is not capable of execution by itself. A **thread** is a locus of control within a task; it consists of a register context, a kernel stack, and usually its own user stack (assigned under user control). A Unix process is represented by a task with a single thread under Mach. Additional threads can be created within a task for concurrency and parallelism; the threads share all of the task's resources (memory and communication rights). Threads execute and are scheduled independently by the operating system kernel, so that a blocking operation executed by one thread does not affect others. A debugger normally executes in a task distinct from the application that is being debugged for protection reasons.

Using existing process-oriented debuggers to debug applications with multiple threads exposes a number of problem areas in both the debuggers and the underlying operating system support. Among the more significant of these problems are:

- The debugger lacks internal data structures and process control logic for tracking and controlling the states of the application threads.
- The existing *ptrace* interface and process control logic are inadequate because they assume that there is exactly one application thread.
- Concurrent exceptions (e.g. three threads hit a breakpoint before the debugger gets control) are beyond the ability of both the debugger and the operating system. The exceptions will be serialized, causing potentially bizarre behavior (e.g. a thread appears to hit a breakpoint that was just removed).

We attacked these problem areas individually by improving *ptrace* to deal with multiple threads, replacing the signal and ptrace logic that serializes concurrent exceptions, and enhancing the debugger to understand multiple threads and concurrent exceptions. The following sections discuss our implementation of this strategy.

4. Improving *ptrace*

Our major goal in improving the implementation of *ptrace* was to eliminate *procxmt* and the associated logic that resumes the debugged application. The interaction of *ptrace* with signals requires that the entire

[3]This is not strictly true for pipelined machines with address queues that allow out of order instruction sequencing (e.g. Hewlett Packard's Precision Architecture). The breakpoint and trace trap mechanisms for such machines can be rather complex.

application be resumed, with the result that the threads not stopped in the signal code continue to execute, possibly altering state needed for the debugging process. The new implementation takes advantage of existing Mach features to replace the *procxmt* code (executed by the application to perform actions on itself) with code that allows the debugger process to directly perform the actions itself. This has the beneficial side effect of eliminating two context switches and the associated synchronization from every call to *ptrace*. The following sections describe these changes and also indicate how the ptrace actions could be replaced by other Mach kernel facilities available to a debugger.

4.1. Access to Memory and Registers

The *ptrace* actions that read or write the application memory can be replaced by the Mach *vm_read* and *vm_write* kernel operations that directly access memory of another task. These operations are somewhat inefficient in accessing the small amount of data that ptrace normally transfers. This inefficiency can be alleviated by directly using *vm_read* and *vm_write* in the debugger to examine or write larger amounts of data, or by system-specific extensions to ptrace for transferring larger amounts of data; in both cases the number of kernel operations required to access large amounts of data is reduced.

Similarly, the *ptrace* actions that read or write the application registers can be replaced by the Mach *thread_get_state* and *thread_set_state* kernel operations which also read and write registers. Since user registers are saved on the kernel stack during kernel operations, this improvement is made possible by the fact that Mach places the kernel stacks of threads in kernel memory proper, so that all stacks are accessible to the kernel at all times (in contrast Unix usually forbids access to kernel stacks of other than the current process).

4.2. Other Kernel Information

The *ptrace* action that reads kernel information from the *user* area in Unix indicates which information it wants by an offset into the user structure. Since the user structure has been divided into task and thread specific components in Mach, it was necessary to incorporate kernel code to reassemble a fake user structure from these components so that the offset can be interpreted to find the desired data. This information is also available via the various options to the Mach *table* syscall.

4.3. Execution Control

The *ptrace* continue action can be replaced by the Mach *task_resume* kernel call. Finer control over execution can be obtained by using *thread_suspend* and *thread_resume* to control which application threads are to execute. Single stepping can be implemented by accessing the process control register(s) of the desired thread(s) to set the trace bit (as described in the previous section) before performing the *task_resume*.

There is, however, no direct Mach replacement for the termination action of *ptrace* due to the need to clean up and deallocate Unix state and data structures. It is still necessary for the application to perform this itself by calling *exit* either within or from outside the kernel. The debugger can cause the application to do this reliably by sending SIGKILL to it.

4.4. Mach ptrace

We have applied the above changes to the Unix implementation of *ptrace* to yield the Mach implementation. The resulting implementation does not resume the debugged application in order to examine or modify its state. Although the above changes have been described in terms of the calls that could be made by a debugger outside the kernel, we have taken advantage of the efficiencies afforded by an in-kernel implementation. (e.g. modify only the register requested, rather than performing a *thread_set_state* to modify all of them.). The resulting *ptrace* makes it possible to use existing single threaded debuggers for multithreaded applications because it does not resume the application in order to examine or modify its state. This does not constitute complete support for multithreaded debugging due to lack of support for concurrent exceptions[4], but it is a distinct improvement over the original implementation which is essentially useless for this purpose.

5. Handling Concurrent Exceptions

The interaction of *ptrace* with signals poses a major obstacle to multithreaded debugging because signal semantics require that exceptions be handled serially. This is a reasonable restriction for external event signals and a single-threaded application, but it is unreasonable for exception signals generated by a multithreaded application because it hides some of the application's state. In particular, it is impossible for a debugger to determine the complete state of a multithreaded application that has hit multiple breakpoints in different threads because only one breakpoint can be reported to the debugger. If the user were to remove some of the other breakpoints before continuing the application, some of the pending breakpoint signals would appear to come from non-existent breakpoints. This and similar problems are a direct result of the serialization imposed by signals and can only be alleviated by using a different mechanism to report exceptions to the debugger.

5.1. The Mach Exception Handling Facility

Mach provides a message-based exception handling facility to overcome the limitations imposed by Unix signals. Converting exceptions to messages allows a debugger to receive and record all of the exceptions that have happened to the application since it was last continued, and thus obtain a complete picture of the application's state. The exception handling facility is based on a remote procedure call (rpc) implemented by two Mach ipc messages. The occurrence of an exception causes a message to be sent that identifies the exception, the thread that caused it, and the task containing that thread. The thread waits for a reply to this message; the reply indicates whether the exception was successfully handled. Further details on the use and implementation of the exception handling facility can be found in [4].

A debugger uses the exception handling facility to detect exceptions in the debugged application, including breakpoint and trace traps. The debugger enables this by setting the **task exception port** of the debugged task to a port on which the debugger expects to receive exception messages. Then any exception in the application that is not handled by an application-specific error handler initiates an exception rpc to the debugger. This mechanism replaces signals and the serialization restriction imposed by them; multiple concurrent exceptions result in multiple messages being queued for the debugger. The debugger

[4]This also causes problems with single stepping in the presence of outstanding breakpoint events because the signal received after initiating the single step action may be from a pending breakpoint instead of completion of the single step.

can dequeue and examine all of these messages without continuing the application and thus determine the complete state of the application, including which threads are at breakpoints.

5.2. Unix Compatibility

Unix compatibility is provided by an internal kernel exception handler (the ux_handler) that converts exception messages to signals and sends them to the appropriate threads. This handler sets up its port as the task exception port of /etc/init; the inheritance of the task exception port across task creation (including forks) ensures that exceptions are converted to signals by default unless there is an application that is interested in handling the exceptions, *e.g.* a debugger. A debugger can use the ux_handler to convert exceptions to signals for debugging purposes; forwarding an initial exception message or sending a new one to the ux_handler causes this to occur. The ux_handler's port can be obtained as the original task exception port of the application before the debugger changed it.

The signal compatibility code in the Mach kernel has been extended to ensure that signals generated by exceptions are handled by the thread that caused the exception. This behavior is applied to the nine Unix signals caused by exceptions; SIGILL, SIGTRAP, SIGIOT, SIGEMT, SIGFPE, SIGBUS, SIGSEGV, SIGSYS, and SIGPIPE. As a result, the ux_handler can be assured that the thread to which it delivers an exception signal (i.e. the thread that caused the original exception) will handle the signal.

6. A Multithreaded Debugger

We have developed a debugger for multithreaded applications that utilizes the Mach kernel's debugging support. This debugger is an enhanced version of *gdb* from the Free Software Foundation [8]. This section describes the enhancements and the additional functionality they provide.

6.1. User Interface Extensions

We extended gdb's user interface to allow the independent examination and modification of threads in the application. New commands were added to identify threads and to select a thread to manipulate. Our debugger adopts the concept of a *current thread* to simplify the interface and maintain maximum compatibility with the single threaded version of *gdb*. Although our enhanced *gdb* can only manipulate one thread at a time, it allows the user to select any thread as the *current thread* to be manipulated. Hence the same *gdb* commands that modified the state of a Unix process perform the same modifications to the selected thread. We also changed the single step command to single step only the current thread while preventing all other application threads from running.

In order to isolate aberrant behavior such as race conditions, it is vital to be able to continue only selected portions of a multithreaded application. To support this we also added debugger commands to individually suspend and resume threads. A *suspended* thread does not execute when the application is continued, and remains in this *suspended* state until it is *resumed*. These commands also support options to suspend or resume all threads to allow faster selection of the desired portion of the application (e.g. one can select two threads out of a collection by suspending all of the threads and then resuming the two desired threads). The debugger can resume a thread, but the thread remains suspended until the entire application containing the thread is continued.

6.2. Debugging and Concurrent Events

The major new complication posed by debugging multithreaded programs is that instead of one event (a breakpoint), many events can happen concurrently when the application is continued. Due to parallelism and concurrency within the operating system itself, these events may not be reported to the debugger in chronological order (e.g. two threads hit breakpoints on a parallel processor, but a clock interrupt causes the first thread to be delayed in preparing its exception message so that the second exception message is sent first). In particular, the Mach exception facility makes no guarantees that messages will be delivered in the order that the exceptions occurred. In most cases the order of the events is not of major interest; for the case of breakpoints, the user is probably more interested in the fact that two threads just hit breakpoints (and which breakpoints they hit), than in which breakpoint occurred first.

6.3. Managing Thread Execution

Our debugger uses a six state model to manage the execution states of threads in the application it is debugging. This model underlies the debugger logic for dealing with concurrent events. The six possible execution states for a thread being managed by our debugger and the associated state transitions are shown in Figure 1. The reader should note that the Mach suspend and resume mechanisms are based on reference counts (e.g. a thread that is suspended twice must be resumed twice) and that the task and thread suspension mechanisms are independent (e.g. the effects of a *task_suspend* cannot be reversed by a *thread_resume*). A thread may run only if both it and its task are not suspended. For clarity, only the initial suspend (either task or thread) that makes a thread not runnable, and the final resume that makes the thread runnable, are shown in Figure 1. No other suspends and resumes cause state transitions.

With this state diagram, we can now explain how the debugger manages thread execution. All of the threads for a running application are in the **run** state; events that invoke the debugger cause one of two possible transitions out of this state:

1. The event is an exception, causing the thread to send a message to the debugger. The thread waits in the **exc** state for a reply.
2. The event is the delivery of a Unix signal. The thread is stopped by the Unix signal code and in the **ptrace** state where it can be manipulated using *ptrace*.

In response to the first exception detected (from a set of concurrent exceptions), the entire application is suspended, causing threads in the **run** and **exc** states to move down to the corresponding **suspend** states. The debugger proceeds to obtain information about the remaining exception events and clears the exceptions by sending reply messages (moving the corresponding threads from the **exc + suspend** state to the **suspend** state). The debugger is now prepared to interact with the user, with all of the application threads in either the **ptrace** or **suspend** states. The Unix signal logic guarantees that at most one application thread can be in the **ptrace** state.

When the user requests that the application be continued, the debugger continues the application by using either *task_resume* (if the debugger suspended the task) or the continue action of *ptrace* (if the kernel ptrace code suspended the task). Threads individually suspended by the user remain in the **suspend** state; all others enter the **run** state and resume execution. For single stepping, the debugger suspends all but the thread to be single stepped, arranges to single step that thread by setting the trace bit (using *thread_set_state*), and resumes the task; only the thread to be single stepped enters the **run** state.

Figure 1: Application threads state diagram

6.4. Processing Application Events

Our debugger utilizes two sources to obtain information about events that occur in the application, the status returned by the *wait* system call and the contents of the exception messages. The status returned by the *wait* system call indicates not only whether a thread has stopped in *ptrace*, and why, but also whether the application has terminated. Exception messages identify the thread that caused the exception, and precisely identify the exception as well. This enables our debugger's low level event detection code to treat breakpoints and trace traps differently because they are used for different purposes.

Trace traps need not be reported directly to the user by the low-level event detection code because they are used for exactly two purposes: user-requested single stepping, and debugger-initiated single stepping as part of continuing from a breakpoint. In both cases higher-level debugger code can figure out if the user should be notified and how, so the events are passed directly to that code without reporting the trap directly to the user. The higher level debugger logic will return control to the user or continue the application as appropriate. In the second case, the user is not notified about the internal debugger actions required to continue from a breakpoint. Similar reasoning applies to breakpoints used to implement stepping over functions, subroutines, or procedures as part of a language-level (e.g. C) single step.

In contrast, breakpoint traps caused by user-requested breakpoints are reported directly to the user by the the low-level event detection code. This is so the user is aware of which threads have hit breakpoints since the last time the application was continued. The only breakpoints not reported to users are those used as part of a language-level single-step as indicated above.

Our debugger must use a polling low-level event detection algorithm due to the incompatibility of the sources of event information. The polling interval is implemented by a timeout if no message is received within the timeout interval, along with the use of a nonblocking variant of *wait*. Upon detecting an event, the algorithm loops to obtain information about all pending events; this guarantees that the state reported to the user contains no stale events (e.g. the problem of hitting a deleted breakpoint is eliminated). Our actual event detection algorithm is:

```
1: Attempt to receive an exception message with a timeout.
If there are any exceptions waiting to be processed
   If this is the first exception since we continued last time,
        suspend the task
   If the exception is a trace trap, process it and come back for more
   If the exception is a breakpoint trap, announce it to the user
   Continue in this loop until no more exceptions (i.e., goto 1)

2: See if any thread stopped with a Unix signal.  If so, process it
   If no Unix signals and we found an exception above, process the last
       exception found
   If no exceptions were found, start the entire algorithm over again
       (i.e., goto 1)
```

In this description, the term ‘process’ should be read as ‘invoke the higher level debugger code and set the current thread to this thread.’

When the first exception is received, the debugger suspends the entire task. It continues to receive exceptions which have queued up on the exception port and deals with them appropriately, but no further suspension is necessary. It immediately sends a reply message to each thread which hit an exception. The thread is free to continue running upon receipt of the debugger's reply, but since the task is suspended, the thread won't execute until the debugger resumes the task. When control finally passes up to the user, the *current thread* in the user interface will be set to the the thread found stopped with a Unix signal if there is one (because this signal must be processed in order to reset *ptrace*), otherwise it will be the last thread for which an exception message was received.

6.5. Continuation Logic

A fundamental design decision in our *gdb* makes it impossible to handle more than one breakpoint per continuation without extensive modifications. This restriction is due in part to the *current thread* concept that permits us to utilize existing code; that code can only recover from a breakpoint in the current thread. Our enhanced *gdb* provides the user with workarounds for this problem; multiple breakpointed threads can be single-stepped (clears the breakpoint) or suspended (defers action on the breakpoint until later) before continuing the application. Of course a thread must be the *current thread* before it is single stepped.

6.6. Deadlock

The ability to suspend individual threads in an application creates potential deadlock scenarios. Continuing an application that has no runnable threads causes deadlock because the debugger is waiting for the application to do something and the application is waiting for the debugger to resume one or more of its threads. Due to a feature of Unix signals, the only way to recover from such a deadlock is for the user to send SIGKILL to the application[5] (other signals will not work because they do not resume the application). To avoid this type of embarassing inconvenience, our debugger checks to make sure that at least one thread is runnable before resuming the application; if no thread is runnable, it complains to the user and will not continue the application until at least one thread is resumed. Thread suspension can cause additional deadlocks in the application if all of the running threads need to synchronize with threads suspended by the debugger; these deadlocks can be interrupted by typing control-C because there are threads in the application that are not suspended, and therefore able to handle signals.

6.7. Delivering Signals

Signal delivery forms an important part of the debugging of complex Unix applications because it allows signal handlers to be tested and monitored under controlled conditions. Most Unix debuggers implement the delivery of arbitrary signals by using the *ptrace* "continue with signal" action. This is not as straight-forward for our debugger because the desired thread may not be stopped in *ptrace*, and signals sent to a multithreaded task are handled by the first thread that notices them. We solve this problem by forcing the desired thread into *ptrace* by sending it a signal. To ensure that the thread we are interested in receives the signal, we send a message to the *ux_handler* specifying that thread and suspend all other threads in the task. Upon continuing the application, that thread will execute, receive the signal from the *ux_handler*, and stop in *ptrace*. At this point *ptrace* can be used to deliver the desired signal to the thread.

This mechanism can also be used to force a selected thread to initiate a core dump by sending it a fatal signal that causes a core dump. Since resuming the application resumes all of the threads (not just

---

[5] e.g. kill -9 *pid*

the thread that will generate the core dump), the user must suspend all other threads in order to preserve the application's memory state. Due to single-threaded limitations in the core file format, core dumps for multithreaded programs currently contain state only for the thread that initiated the core dump. Switching to a new core file format that allowed multiple thread contexts to be dumped would avoid this limitation.

7. Debugger Attachment

A debugger that attaches to a running process has been implemented using the techniques described in Section 4 for replacing *ptrace* with other kernel operations. Before using these techniques, the debugger must first obtain task and thread ports for the application. All Mach tasks and threads are represented to the outside world by a port; a user process must have the corresponding port in order to perform operations on a task or thread. Our debugger currently uses the Mach **task_by_unix_pid** kernel call to obtain the task port for a given Unix process id (pid); this call checks the Unix uid's of the application and debugger to prevent a protection violation. The thread ports for threads in the task can be obtained by using the Mach **task_threads** kernel call. Because the *wait* call cannot be used to detect termination of the application in this case (the debugger is not the parent of the debugged application), a notify mechanism based on the death of the task port (caused by application termination) is used to detect application termination.

Attaching to and detaching from an application is transparent because the debugger does not rely on *ptrace* or the STRC "traced" bit to alter application behavior. The debugger attaches by obtaining the application's task port and suspending the application to examine it. It detaches by resuming the application and deallocating the task port from the debugger. The deallocation happens implicitly if the debugger exits.

Signal behavior is slightly different from the normal case because the application process is not "traced". External event signals no longer cause the application to stop for the debugger, but instead invoke their handlers or carry out the corresponding default actions. Interruption of a runaway process is implemented by deploying a signal handler for SIGINT in the debugger; a control-C invokes this handler to wake up the debugger which then suspends the application for further debugging. Signal delivery to the application is implemented by using the normal Unix kill() mechanism to send signals; this may not override pending signals (unlike the *ptrace* continue with signal mechanism).

8. Related Work

[7] describes CodeView 2.2 which provides debugging support for multithreaded programs in OS/2. The OS/2 kernel provides an enhanced *ptrace* interface for interaction between the debugger and the application making possible a clean debugger design and user interface. The major enhancements to *ptrace* are the ability to manipulate any thread in the application (an additional argument specifies the desired thread), and the decoupling of *ptrace* from the semantics of signals [5,6]. These improvements involve changing the *ptrace* call itself and are made possible in part by the absence of a need to maintain Unix compatibility. Concurrent breakpoints are not mentioned in any of these sources. This might be because OS/2 only runs on uniprocessors where it is possible to avoid concurrent breakpoints via scheduling enhancements (e.g. run debugger immediately in response to the first breakpoint). We did not have this choice because concurrent breakpoints cannot be avoided on multiprocessors that support parallel execution of threads.

The Parasight project at Encore Computer has been following a different approach for debugging multithreaded applications on multiprocessors. [2] Parasight inserts debugging “parasites” into the application to directly monitor its execution state; these “parasites” execute as independent threads on spare processors (not in use by the application) to continually check assertions. Parasight relies heavily on this technique of modifying the application to perform debugging (e.g. breakpoints are implemented by a branch to debugging code inserted into the application by Parasight). In contrast, we have retained the traditional Unix debugger model which avoids modifying the application except for breakpoint insertion.

9. Future Enhancements

One major area of work we chose not to attack was a major redesign of *gdb* to incorporate support for multiple threads throughout its code. Such a redesign would include saving state information about every thread so that multiple breakpoints can be handled by a single continue. This would make it possible to support conditional breakpoints and breakpoints that execute commands and continue without user intervention which cannot be supported in our current design.

In addition, the exception handling processing should be written to take advantage of the detailed information received through the MACH Exception Handling Facility. For example, when the debugger receives a breakpoint trap exception from MACH, it knows whether the exception was caused by a real breakpoint or by a trace trap. Using the *ptrace/wait* combination for finding out an application thread’s stop status, all the debugger knows is that the thread stopped on a SIGTRAP which could be either a breakpoint or a trace trap. Our *gdb* is currently implemented to convert the detailed information back to a SIGTRAP so that the original *gdb* code can interpret what to do. Much of the complexity of that code could be simplified by knowing up front what kind of trap occurred.

10. Acknowledgements

The Mach kernel features for debugging support (Mach ptrace and the exception handling facility) were implemented by David Black at Carnegie Mellon University. Some initial work on adapting gdb to multiple threads was done by Karl Hauth at Carnegie Mellon University. The bulk of the work in restructuring gdb’s internals to properly deal with multiple threads was done by Deborah Caswell at Hewlett Packard Laboratories. The attachment feature was implemented by Richard Sanzi at Carnegie Mellon University. The authors would like to thank Hewlett Packard for supporting this work. Further detail on the Mach system and the various kernel calls used by our debugger can be found in [1] and [3].

References

[1] Accetta, M., R. V. Baron, W. Bolosky, D. B. Golub, R. F. Rashid, A. Tevanian, Jr. and M. W. Young. “MACH; A New Kernel Foundation for UNIX Development", *Proc. USENIX 1986 Summer Technical Conf. and Exhibition*, 9-13 June 1986, 93-112.

[2] Aral, Ziya, Ilya Gertner, and Greg Schaffer. “Efficient Debugging Primitives for Multiprocessors", *ASPLOS-III Proceedings (Third International Conference on Architectural Support for Programming Languages and Operating Systems)*. Boston, MA, April 3-6, 1989. pp. 87-95.

[3] Baron, R. V., D. L. Black, W. Bolosky, J. Chew, D. B. Golub, R. F. Rashid, A. Tevanian, Jr. and M. W. Young. *MACH Kernel Interface Manual*, Dept. of Computer Science, Carnegie-Mellon Univ., 15

Feb. 1988.

[4] Black, D. L., D. B. Golub, K. Hauth, A. Tevanian, Jr. and R. Sanzi. “The MACH Exception Handling Facility", *Proc. A.C.M. SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging, SIGPLAN Notices* 24,1 (May 1988), 45-56.

[5] Letwin, Gordon. *Inside OS/2* Redmond,Wa.: Microsoft Press, 1988. 289 pages.

[6] Microsoft Corporation. *Microsoft OS/2 Programmers Reference, Volume 3*. Redmond, Wa.: Microsoft Press, 1988. 430 pages.

[7] Petzold, Charles. “Techniques for Debugging Multithread OS/2 Programs with CodeView 2.2", *Microsoft Systems Journal*, September 1988, pp. 21-29.

[8] Stallman, R.M., *GDB Manual; The GNU Source-Level Debugger, 1st ed., GDB Version 2.0*, Free Software Foundation, Jan. 1987.

[9] Tevanian, Jr., R. F. Rashid, D. B. Golub, D. L. Black, E. Cooper and M. W. Young. “MACH Threads and the UNIX Kernel; The Battle for Control", *Proc. USENIX 1987 Summer Technical Conf. and Exhibition*, 8-12 June 1987, 185-197.