A Multi-Threaded Debugger for Multi-Threaded Applications

Diplomarbeit
von
Martin Karsten
aus
Kempen-Hüls, jetzt Krefeld

vorgelegt am
Lehrstuhl für Praktische Informatik IV
Prof. Dr. Effelsberg
Fakultät für Mathematik und Informatik
Universität Mannheim

August 1995

Betreuer: Prof. Dr. Wolfgang Effelsberg

Vorwort/Preface

Diese Arbeit ist an der Universität von Waterloo, Kanada, erstellt worden, in Kooperation mit dem Lehrstuhl für Informatik IV an der Universität Mannheim, Deutschland.

Zuallererst möchte ich mich bei Professor Dr. Peter A. Buhr bedanken, für die Möglichkeit, diese Arbeit zu erstellen, sowie für seine sorgfältige Betreuung und Unterstützung. Zusätzlicher Dank gilt Professor Dr. Thomas Kunz und Dr. Bob Zarnke für ihre wertvollen Anregungen. Dankbar erwähne ich an dieser Stelle die Gastfreundschaft und finanzielle Unterstützung der Universität Waterloo.

Im weiteren gebührt mein Dank Professor Dr. Wolfgang Effelsberg und Professor Dr. Reinhard Männer von der Universität Mannheim für ihre Kooperation und Unterstützung.

Ein grosses "Dankeschön" geht an meine Familie, insbesondere an meine Eltern, Gisela und Richard Karsten, und meine Tante, Ursula Karsten.

Schliesslich, ich hatte viel Spass hier und möchte mich bei einer Reihe von Freunden dafür bedanken, mich immer wieder von der Arbeit abgelenkt zu haben.

This thesis was written at the University of Waterloo, Canada, in cooperation with the "Lehrstuhl für Informatik IV" at the University of Mannheim, Germany.

First and foremost, I wish to thank Professor Dr. Peter A. Buhr for the possibility to write this thesis and his thorough supervision and support. Additionally, I want to thank Professor Dr. Thomas Kunz and Dr. Bob Zarnke for providing valueable comments. I gratefully acknowledge the hospitality and the financial support of the University of Waterloo.

Furthermore, I want to thank Professor Dr. Wolfgang Effelsberg and Professor Dr. Reinhard Männer from the University of Mannheim for their cooperation and support.

A big "thank you" goes to my family, especially to my parents, Gisela and Richard Karsten, and my aunt, Ursula Karsten.

Finally, I had a good time and want to thank a number of friends for giving non-technical support. (Biraj, Jenni, Martin, Stacy, that's for you! :-))

Ehrenwörtliche Erklärung

Hiermit versichere ich, die vorliegende Diplomarbeit ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus den Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Püfungsbehörde vorgelegen.

Waterloo, den 13. August 1995 Martin Karsten

Abstract

An *interactive source-level debugger* is a tool that allows stepwise execution of an application with possible modification of the execution path or data flow. This level of control is essential to develop and verify hypotheses about faulty behaviour of an application.

Multi-threaded applications introduce two additional complications into debugging. First, traditional sequential debuggers are unable to recognize the existence of different threads of control in a program, and hence, are largely useless for concurrent debugging. Second, *data races* can cause the program to alter its behaviour from execution to execution, making it difficult to reproduce failure cases.

The goal of this thesis is to identify the key aspects of interactive source-level debugging of multi-threaded applications, which may execute in a shared address space or be spread over multiple address spaces. Central issues are: independent control of every thread in the target application, especially for shared-memory applications, and support for a broad variety of application models.

A general design for a debugger for multi-threaded applications is presented. Furthermore, the debugger itself is a multi-threaded program, which allows every thread in the target application to be controlled independently from the execution of other threads. It can properly deal with simultaneous events by the multiple threads in the application being debugged and the interactive requests of the programmer debugging the application. A prototype implementation for a debugger is constructed that fulfills these requirements. It is built using μC++, which extends C++ with new language constructs for concurrency, and it can be used to debug μC++ applications. This debugger is intended to be used stand-alone or to serve as a basis for a debugging environment that gives further support to handle data races.

Trademarks

Ada is a registered trademark of of the U.S. Government (Ada Joint Program Office)
Alpha AXP is a registered trademark of Digital Equipment Corporation
DCE is a registered trademark of Open Systems Foundation, Inc
MIPS is a registered trademark of MIPS Technologies, Inc.
Motif is a registered trademark of Open Systems Foundation, Inc
Solaris is a registered trademark of Sun Microsystems, Inc.
SunOS is a registered trademark of Sun Microsystems, Inc.
SPARC is a registered trademark of Sparc International, Inc.
UNIX is a registered trademark of UNIX System Laboratories, Inc.
X Window System is a registered trademark of X Consortium, Inc.

Abbreviations

BFD	Binary File Descriptor
COOL	Concurrent Object Oriented Language
CPU	Central Processing Unit
DCE	Distributed Computing Environment
EBBA	Event Based Behavioural Abstraction
GDB	Gnu's Debugger
IP	Internet Protocol
IPC	Inter-Process Communication
RISC	Reduced Instruction Set Computer
TCP	Transport Control Protocol

Contents

List of Figures

List of Tables

Chapter 1

Introduction

To use today's computer resources efficiently, it is necessary to design solutions that take advantage of concurrency and parallelism, either on a single machine, which may have multiple processors, or on multiple machines, which are connected by a network.

While many approaches have been made to address multi-threaded programming and debugging of multi-threaded programs, there is currently no consensus on how to deal with concurrency in terms of programming languages and how to debug such programs effectively. Parallelism is often expressed through library calls to create several threads of control within a program or is achieved by writing several programs and having them explicitly communicate via shared memory or message passing, like applications using the TCP/IP protocol stack.

An alternate approach is to incorporate concurrency through new programming language constructs. Probably the most popular example is *Ada* [38]. A newer example is *μC++* [6], which also is a successful approach to integrate concurrency and object-orientation, a so-called *Concurrent Object Oriented Language (COOL)*. The μC++ project extends C++ with new language constructs to express parallelism and provides a run-time system that runs programs concurrently or in parallel when appropriate hardware is available.

Expressing concurrency through programming language constructs is useful and necessary to make the development of large multi-threaded applications efficient and to produce robust and maintainable software. This thesis deals with the problem of debugging concurrent applications programmed in a high-level concurrent programming language. While the focus is on concurrent programming languages, the underlying principles, from the standpoint of debugging, are applicable to several forms of parallelism.

1.1 Thesis Organization

Chapter 2 surveys different paradigms for multi-threaded systems and applications. The major differences between debugging of sequential and concurrent applications are outlined. It also establishes some basic terms that are used throughout this work.

Chapter 3 surveys related approaches for debugging multi-threaded programs and shows how this thesis fits in.

In Chapter 4, a classification of interactive source-level debuggers for multi-threaded applications is presented. Previous approaches for interactive source-level debuggers are examined and compared with the new ideas presented in this work.

General design goals are discussed in Chapter 5. The restrictions and additional complexity resulting from these goals are shown and special aspects of a multi-threaded debugger are discussed.

In Chapter 6, specific aspects of breakpoint implementation are described. As well, Chapter 6 discusses low-level issues that vary for different platforms.

Chapter 7 presents the design of the implemented prototype. Static aspects like separating the debugger into multiple modules are presented in detail. Also, the dynamic aspects, resulting from the fact that the debugger is itself a multi-threaded application, are discussed.

The algorithms for the core operations of the debugger that deal with the aspects from Chapter 6 are explained in Chapter 8. Because the presentation of these algorithms refers to the design that is introduced in Chapter 7, it is presented in a separate chapter.

Chapter 9 contains the user's guide for *kdb*, the prototype implementation built using μC++, for debugging μC++ applications.

Finally, Chapter 10 summarizes the thesis and gives ideas on how the this work can be used in future debugging environments.

Chapter 2

Multi-Threaded Systems

2.1 Architectures

There exists a large variety of different machine architectures that support multi-threaded programming. This variety ranges from uniprocessor machines, on which programs are run concurrently via context-switching among different processes up to loosely coupled networks, where an application can be distributed over different machines. Figure 2.1 gives an overview of the different architectures.

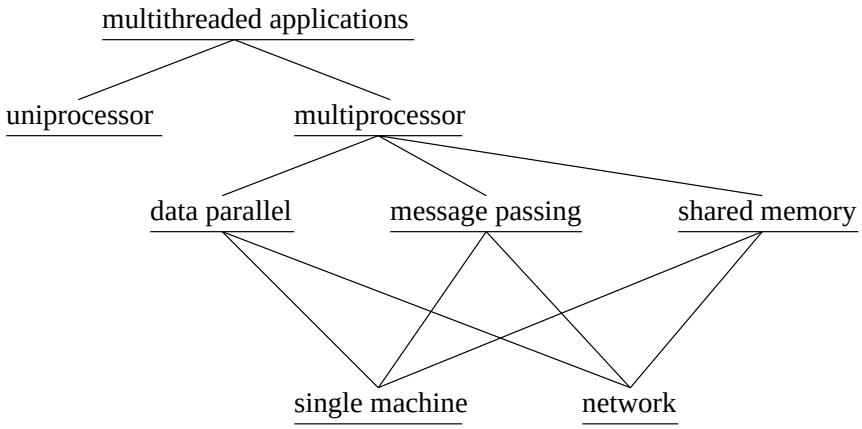


Figure 2.1: Machine Models

A major issue in distinguishing the different architectures, from the standpoint of debugging, is the type of inter-process communication (IPC) used to enable communication

among different threads of control. The two major approaches are message passing and shared memory, which both need to be supported by debugging tools.

Support for message passing mechanisms is needed because there will always be situations where shared memory does not exist among computers that need to communicate. Even if a distributed system provides a shared memory emulation, there will be situations where the efficiency of the emulation is unacceptable. Hence, there will always be applications that use message passing, and therefore, debugging tools are needed to support the development of those applications.

Whenever shared memory and a suitable synchronization environment is available, it should be the programmer's first choice for communication, because it provides the same data exchange model (and usually interface) for multi-threaded programming as for sequential programming. The general consensus is that a shared address space is the better mechanism to support high-level language constructs, because programmers should not be forced to deal with all the necessary communication details required by message passing. An examination of these aspects as well as numerous arguments for both kinds of communication can be found in [17].

In this work, shared memory is the focus, which either relies on hardware support or on software emulation, but certain aspects of non-shared memory are also considered.

2.2 Applications

In the literature, the terms *concurrent*, *parallel* and *multi-threaded* are used to describe non-sequential applications. A multi-threaded application creates and uses several threads of control to solve a given problem. Very often this term is used to denote the fact that the application resides in one address space and is executed by one hardware processor, but there is no inherent reason to restrict the term in this way. Parallelism can only occur when multiple hardware processors are involved in executing the application. Finally, concurrency denotes that an application *appears* to execute in parallel. However, all parallel systems are also concurrent systems and all parallel and concurrent applications contain multiple threads of control. Thus, the different terms do not denote differences in the logical structure of the program's control flow, and therefore, all three terms are used synonymously in this work.

Multi-threaded applications consist of a set of concurrent or parallel running threads. In this work, threads are distinguished by two properties: the *creation level* and the *address space*. The creation level indicates how the thread is created. If a thread is created by

the operating system kernel, it is called a *kernel thread*. If a thread is created by the runtime system of a programming language or a thread package in the user address space, it is called a *user-level thread*. The main difference is whether the operating system kernel is involved in the context-switches among them. The address space property indicates if multiple threads share the same address space.

	creation level	address space
UNIX processes	kernel	not shared
DCE threads	user	not shared
SunOS threads	kernel	shared
μC++ tasks	user	shared

Table 2.1: Thread Types

Table 2.1 shows typical examples for different combinations of threads and address spaces. This table is not exhaustive nor exclusive. For example, multiple DCE threads can be created in the same address space or an application can contain multiple μC++ modules, which are spread over multiple address spaces and communicate via message passing.

The address space of an application is usually separated into code memory and data memory. Throughout this work, whenever this distinction is important, the term *code image* is used to describe a separate section of code memory. If multiple threads share the same code image, they usually also share their data address space. The reverse is not necessarily true, multiple threads of control can have separate code images and share their data sections. In UNIX, every process has its own code image, but multiple processes can share their data memory.

On all systems, user-level threads are executed by kernel threads. The user-level threads context-switch among themselves through the runtime system provided by the programming language or the thread package, and the kernel threads context-switch among themselves in the operating system kernel. By time-slicing user-level threads across kernel threads, the user-level threads share each kernel thread's time-slice. In general terms, every kernel thread executes a group of user-level threads. A kernel thread that does not execute multiple user-level threads can be seen as executing only one user-level thread.

To achieve better performance or load-balancing among user-level threads on multiprocessor machines, n kernel threads may be aggregate in a logical unit, called a *cluster*.

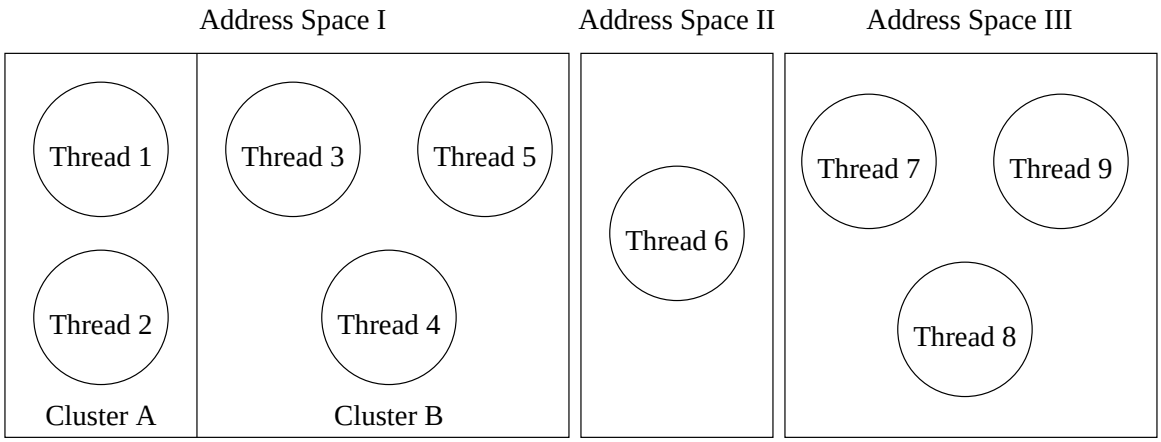


Figure 2.2: Threads and Address Spaces

The kernel threads that belong to the same cluster share one data address space and execute m user-level threads, where usually $m \geq n$. The kernel threads do not necessarily share one code image, but all are created from the same executable file, and hence, they have equal code images. Within a cluster, the distribution of user-level threads among the kernel threads can change arbitrarily. If a kernel thread is not aggregated with any other kernel thread, it can be seen as a cluster with only one kernel thread.

In general, an application consists of multiple user-level threads, groups of which execute on a particular cluster. At least one kernel thread is associated with each cluster and executes the user-level threads. Multiple clusters can share one address space. This model is general enough to cover most of the existing application models.

Figure 2.2 depicts a sample application, showing the model of threads and address spaces. The application consists of 9 user-level threads. Threads 1 and 2 are executing on cluster A, threads 3–5 are executing on cluster B, all of them sharing address space I. Thread 6 is executing alone in address space II. Within address space III, the threads 7–9 are executing. The kernel threads that execute the user-level threads are not shown in this picture.

Figure 2.3 shows how multiple user-level threads on a cluster are executed by kernel threads in μC++. A detailed description can be found in [6]. However, the internal scheduling mechanism in a cluster depends on the runtime system, hence, no general assumptions can be made about it.

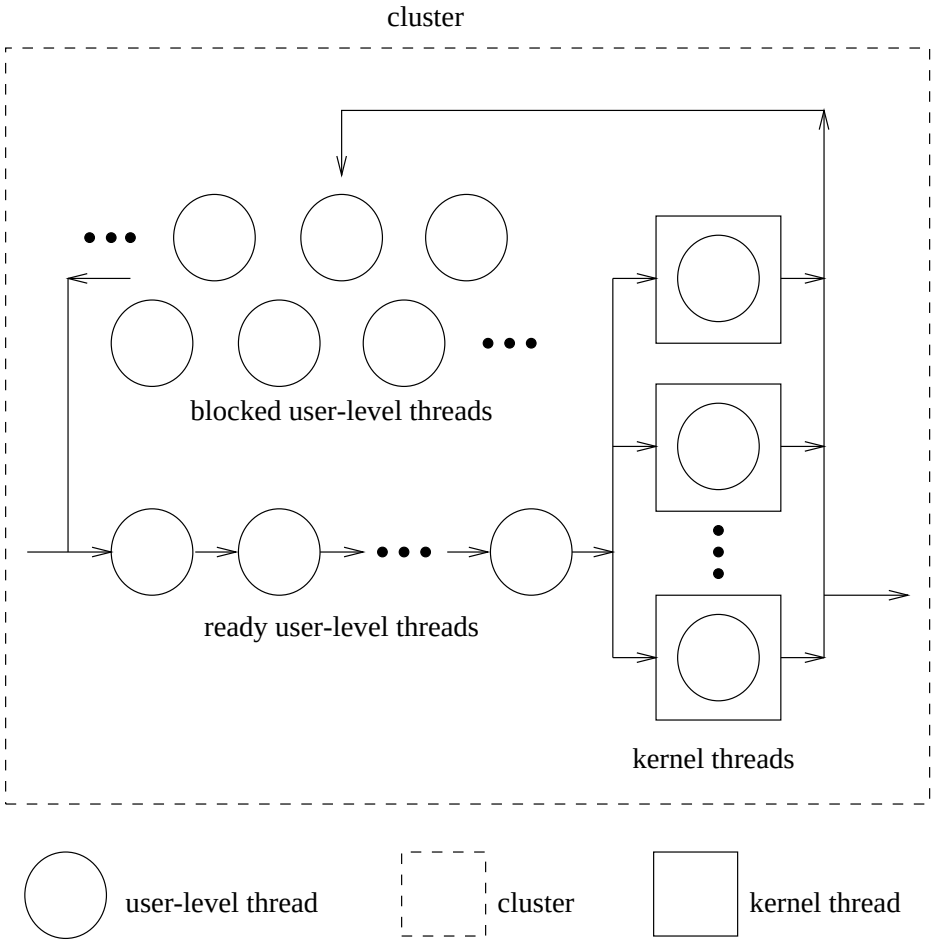


Figure 2.3: Threads on a Cluster

2.3 Debugging

In this work, debugging is considered as a cyclic process that consists of two phases [1]:

- **phase 1:** developing a hypothesis about faulty behaviour of the program
- **phase 2:** verifying this hypothesis with successive confirmation or rejection

If an application does not show the expected or predicted behaviour, formal or non-formal techniques are used to develop a hypothesis about the reason for the failure. Then, attempts are made to verify this hypothesis, and if the hypothesis is confirmed, the error

is located and can be fixed. An *interactive source-level debugger*, referred to as a *debugger*, is a tool that allows stepwise execution of an application with possible modification of the execution path or data flow. It supports the programmer during both phases of debugging.

The major difference between debugging of sequential programs and debugging of multi-threaded programs is that multiple threads of control are allowed to modify shared data. Furthermore, since most applications do not run under strong real-time constraints, the execution of each single thread is largely unpredictable due to clock tolerance, shared device access, interrupts, etc. [19]. Therefore, access to shared data must be synchronized properly to prevent *data races*. A data race occurs when at least two threads of control are accessing the same piece of data concurrently and at least one access is a write operation.

Whenever synchronization is insufficient due to errors in the algorithm or the implementation of the program, the program becomes *non-deterministic*, i.e., a data race causes the program to alter its behaviour from execution to execution. Therefore, it can be non-trivial to reproduce an observed failure of the program. On the other hand, it must be mentioned that there are a number of cases where non-determinism does not result from failures, but is the intended behaviour.

When using a debugger to monitor or control the execution of an application, the debugger itself usually influences the execution of a multi-threaded program, this phenomenon is called the *probe effect* [10]. The probe effect might be large enough to mask data races that only occur when the application is executing without a debugger. On the other hand, it might also point to the existence of errors that do not occur without any debugging activity. Even though the latter possibility can generate some surprising and unintended help, both cases are serious problems faced by developers of multi-threaded applications.

For the above reasons, the sole usage of an interactive source-level debugger may not be sufficient during phase 1 of debugging of multi-threaded programs and may have to be combined with some of the approaches that are discussed in Chapter 3. Nevertheless, interactive source-level debugging is the only applicable approach for phase 2 debugging. This work provides a new approach for the development of an interactive source-level debugger for multi-threaded programs that attempts to deal with the general application model described in Section 2.2.

Chapter 3

Related Work

There have been many different approaches for debugging multi-threaded programs. Usually the term *distributed debugging* is used to describe debugging of applications that are spread over a network and *parallel debugging* is used to describe debugging of applications that run on a single machine. However, a strict distinction between both definitions cannot be made, therefore the terms *debugging of multi-threaded applications* or *concurrent debugging* are used synonymously throughout this work.

A survey about debugging of multi-threaded programs is given in [21]. Approaches that try to deal with data races and possible non-determinism are *static analysis* techniques, event-based *dynamic analysis* techniques or combined approaches. Both approaches have to be combined with traditional *interactive source-level debugging* to be useful in debugging of multi-threaded programs.

3.1 Static Analysis

The term *static analysis* refers to the process of examining an application's source code to detect data races and coding schemes that are known to be dangerous, i.e., error-prone.

Emrath and Padua [8] present an approach to automatically detect non-determinism in parallel programs. The approach uses an analyzer to check whether a program is deterministic. If not, the analyzer tries to find the reasons for non-determinism. If eventually the program is shown to be deterministic, all other errors can be addressed with an interactive debugger. However, if a program is intended to be non-deterministic, this approach is restricted to only those parts of the program that are supposed to be deterministic.

The work of Schatz and Ryder [22] is another example of a static approach to detect data races. The idea is to generate a parallel program dependency graph and perform *program slicing* [39] on this graph to detect possible races. Static program slicing examines the text of a program and includes all statements that could affect the value of a given variable at a particular program point. Program slicing is done for any possible variable that is suspected of being involved in a data race and the output is processed by an analyzer module. Their approach contains a second step, *symbolic execution* of the program, using the dependency graph, to verify the potential races. Symbolic execution accumulates symbolic expressions for dedicated variables at any point along some program path. The various generated expressions are subsequently used to verify a presumed data race.

In general, there are two major restrictions in using only static analysis:

- Examining the source code is not realistic for large programs, since it often takes exponential time and/or space and produces huge amounts of output.
- Static analysis can give hints to error-prone parts of the program, but not everything that is detected is an error, and, most importantly, errors may remain undetected.

Thus, static analysis tools can support the development of multi-threaded programs, but are not powerful enough to be the only support for producing stable applications.

3.2 Dynamic Analysis

Dynamic analysis approaches try to overcome the influence of the probe effect and the possibility of non-determinism of a multi-threaded program in a general fashion. Programs are instrumented to generate events during communication operations and shared data access. An analyzing tool is able to capture these events during execution time and analyze them on-the-fly or post-mortem. Some tools are even able to simulate the environmental events for any thread.

The basic concept for dynamic analysis is to devise a scheme for ordering captured events [9] [19] to gain some insight about the causality of events, to collect information to re-execute a program run and to compare the actual with the predicted behaviour of the application.

Summers [28] presents an approach to reduce the huge amount of output generated during event capturing. He presents an algorithm to aggregate events into abstract events,

thus reducing the number of events that are shown in the output. Another algorithm is described that helps aggregating processes into groups of processes, called "clusters". (The term "cluster" is different in this context than discussed in section 2.2). In this work, the corresponding term is *group of threads*. All communication among processes within a "cluster" is not captured, which also reduces the size of the output. A realization of cluster abstraction is presented by Kunz in [18].

EBBA [3] is a tool-set that uses behavioural abstraction. Based on events captured from the application, behaviour patterns are recognized and compared with one or more behaviour models that are specified by the user.

Taylor [37] presents an implementation for a graphical visualizer that displays the events as points along time lines.

3.3 Combined Static and Dynamic Analysis

Zinn [40] proposed a combined approach of static and dynamic analysis that reduces the time complexity of previous approaches significantly. However, it does not provide an exact test for data races, and hence, only gives hints for locating a problem.

3.4 Interactive Source-Level Debugging

Several previous approaches exist that apply the interactive source-level debugging approach, as described in Section 2.3, to multi-threaded programs.

Node Prism

The *Node Prism* debugger [23] is targeted for massively parallel message-passing programs running on a multi-processor machine. The target application consists of multiple kernel threads, UNIX processes, each of them associated with one CPU. Each process is controlled by a dedicated simplified debugger process, which executes on the same CPU. A *central debugger* is running on another dedicated CPU. Each simple debugger controls its corresponding target process on behalf of the central debugger, using the UNIX debugging support. In the central debugger, processes can be aggregated into groups and debug operations can be issued for a group of multiple processes using only one group command. Node Prism features a partly graphical user interface with a scalable amount of detailed output.

LPdbx

Another approach is taken by the *LPdbx* debugger [25], a system that is targeted for loosely coupled parallel processors. In this approach there is also one simple debugger process attached to each target process and it controls the target process on behalf of a so-called *master debugger*. This system lacks facilities to group target processes for debugging purposes, but introduces a full graphical user interface with independent windows for each target process. Hence, the master debugger can be seen as a multi-threaded application, achieving concurrency by using the event handling facilities of the X Toolkit Intrinsics, but it is not explicitly programmed as a multi-threaded application.

GDB

Gnu's Debugger, GDB [26], also offers some support to debug multi-threaded applications. GDB can recognize and deal with the existence of several kernel threads, which all share the address space of exactly one UNIX process. In particular, breakpoints may be set for a single thread and the target application is only stopped if this thread hits the breakpoint.

A Debugger for Multi-Threaded Applications

Jacobs [13] described and implemented a debugger for multi-threaded shared-memory applications, consisting of one kernel thread's execution that is split up and used by multiple user-level threads. With this debugger, it is possible to issue operations and set breakpoints for individual user-level threads.

3.5 Summary

Debugging of multi-threaded applications turns out to be considerably different from debugging sequential applications. The existing approaches for static and dynamic analysis present new algorithms to deal with the possibility of data races, non-determinism and the probe effect, as well as abstract views to deal with the huge amount of information these algorithms generate. None of the presented approaches is sufficient to produce error-free multi-threaded applications, neither does interactive debugging guarantee to find all errors in an application. Static and dynamic analysis might be extremely helpful for developing a hypothesis about erroneous behaviour of a multi-threaded application. However, to

effectively apply these approaches, there must always be an interactive source-level debugger, so that it is possible to perform stepwise execution of a program and verify such a hypothesis.

This work presents a design for an interactive source-level debugger that is not restricted to a certain subset of multi-threaded applications and is intended to eventually cooperate with tools that implement high-level analysis.

The surveyed debuggers from Section 3.4 are classified and discussed further in Chapter 4.

Chapter 4

Classification of Debuggers

Besides the timing aspects of debugging multi-threaded programs as described in Section 2.3, there is a need to distinguish approaches for debuggers based on certain technical lines, resulting from different architectures and application paradigms that allow multi-threaded programming. This chapter presents a classification of debuggers based on the concurrent structure of the target application (see Section 2.2): thread creation level (kernel or user) and address space (shared or non-shared). As well, one additional property is added: support for independent target thread control, i.e., the debugger must be able to independently control the threads in the application, whether they are kernel or user-level threads.

A debugger should have all these properties, so that it is a flexible tool to debug large, heterogeneous applications that make use of different forms of parallelism and communication. The following sections explain how the fulfillment of these properties places certain requirements on the design and implementation of a debugger. It is also examined whether and how previous approaches fulfill each property.

4.1 Thread Types

A debugger should be able to control applications that use multiple kernel threads and understand when the kernel threads are running multiple user-level threads. Furthermore, if a runtime system aggregates user-level threads into clusters, as described in Section 2.2, this also has to be dealt with by the debugger.

The traditional debugging primitives of UNIX are the `ptrace` [32] system call and the newer `/proc` filesystem [31]. Initially, `ptrace` was only intended to operate on processes

with one thread of control, but some work was done to enhance this functionality in Mach, to support multiple kernel threads in one address space [7]. The `/proc` filesystem supports multiple kernel threads in one address space.

With both mechanisms, even though multiple kernel threads can be recognized and several operations can be issued on single kernel threads, there is no support for user-level threads, since these mechanisms are part of the operating system kernel and cannot anticipate the design of a user-level thread package.

LPdbx, NodePrism and GDB rely on the UNIX primitives, hence they only allow debugging of applications consisting of multiple kernel threads and do not handle the existence of user-level threads. Jacob's system supports user-level threads, but is restricted to one kernel thread. None of the surveyed approaches show any awareness about user-level clustering mechanisms.

4.2 Address Space

To debug shared memory programs, a debugger has to deal with multiple threads that execute in the same address space. In UNIX, a shared address space scenario can be achieved by creating multiple user-level threads within a single process or having multiple processes sharing parts or all of their address space using the `mmap` [30] system call. In the *Mach* operating system, each "task" provides an address space and communication rights. A kernel thread is a locus of control within a task. A UNIX process is represented by a task with a single thread of control. Additional threads can be created within a task [4], building an application where multiple threads share the same address space. Several UNIX versions provide kernel thread libraries that follow the Mach model.

As well, a debugger should be able to control targets that are executing in multiple, non-shared address spaces. As stated previously, multi-threaded applications may communicate via message-passing as well as via shared memory to achieve optimal efficiency. It is realistic to consider applications that are spread over different address spaces and still have multiple threads of control executing in each address space. It is desirable to have control for all threads aggregated for debugging purposes in one debugger instance. Hence, a debugger should not be targeted for either shared *or* non-shared address spaces, but for both. The requirement to support multiple address spaces, as well as sharing of address spaces exists for the code images and for the data sections of an application.

GDB as well as Jacob's debugger support multiple threads within one address space, but do not support applications using multiple address spaces. LPdbx and NodePrism are

targeted for message passing systems, hence they do not support multiple threads in one address space but instead support multiple address spaces with one thread executing in each.

4.3 Independent Target Thread Control

To interactively debug multi-threaded applications, it is necessary to have the ability to handle the different threads of control independently, but through a single debugger. Consider the following example, illustrated by Figure 4.1: Two threads, A and B, use a

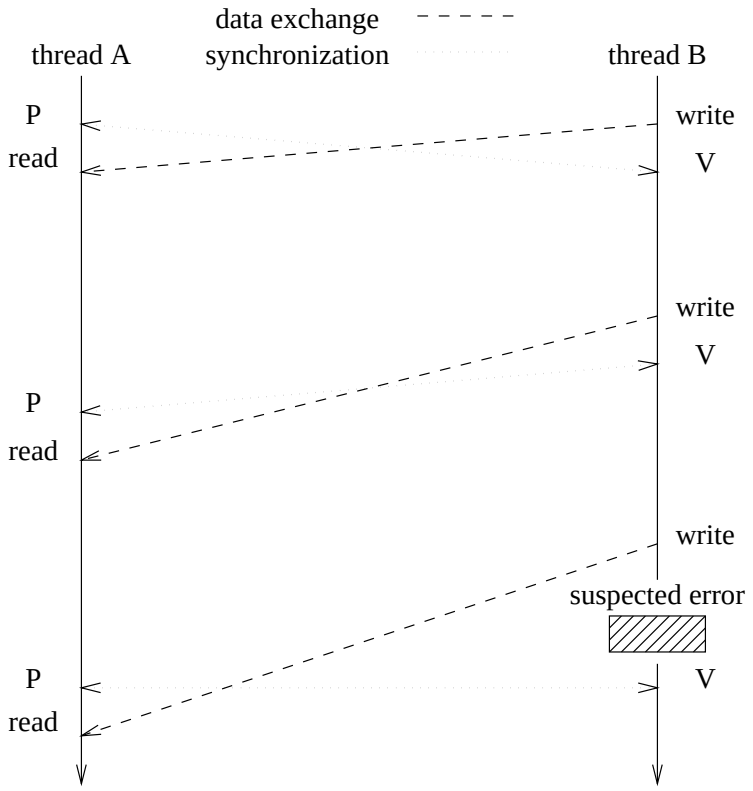


Figure 4.1: Example Problem

binary semaphore to synchronize during execution, and each time B delivers data to A. Initially the semaphore's value is 0, and at a synchronization point, thread A calls the `P` operation to wait for B. Thread B performs the `V` operation, whenever it can deliver the appropriate data. Consider a hypothesis that there is a problem before a certain

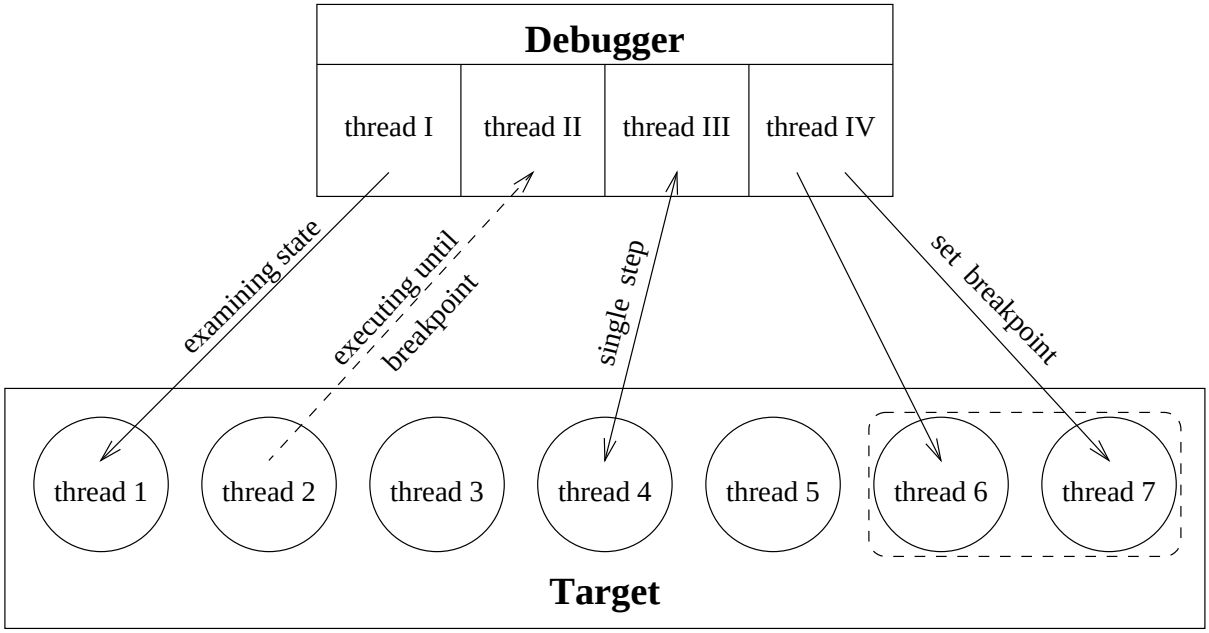
synchronization point, related to the `V` operation executed by thread B. Then, a possible debug approach is to start thread A and let it wait at the `P` operations. Meanwhile, thread B is executed stepwise and the system's behaviour can be inspected, since the stepwise execution does not prohibit thread A from automatically resuming execution whenever possible.

Hence, to debug concurrent applications, a debugger not only needs to be aware of the different threads of control, but must allow interactive requests from the user to control single threads or groups of threads, while other threads are executing. This objective could be achieved by a sequential debugger that only handles one thread at a time or by having copies of a sequential debugger controlling each target thread of interest. However, the cleanest solution is for the debugger itself to be a multi-threaded application with different threads managing the threads of the target application, so that a dedicated debugging thread can be created for every target thread that is being inspected. As well, a single thread in the debugger can control a group of threads in the target. The general design of a multi-threaded debugger can be seen in Figure 4.2.

As an immediate consequence, the communication mechanisms between the debugger and the target must be asynchronous at the application level, so that the necessary communication for the control of a single thread neither blocks the whole debugger nor the whole target application. On the other hand, the communication between a thread in the target and the corresponding thread in the debugger can be synchronous or asynchronous, depending on the actual situation. For example, the target thread's communication is synchronous, i.e., either the thread is stopped and waiting for incoming messages or running and able to send messages. For the debugging thread of control, it is an advantage to have asynchronous communication, so that certain operations can be issued on a running target thread.

Unfortunately, the traditional debugging primitives of UNIX, the `ptrace` [32] system call or the `/proc` filesystem [31], only offer a synchronous communication interface (see Section 4.5 for details) at the application level. Hence, they are only partly suitable for a multi-threaded debugger.

From the surveyed debuggers, only LPdbx allows independent control of every target thread. Asynchrony is achieved with a dedicated UNIX process for each target process that synchronously controls the target on behalf of the central debugger and communicates with the central debugger in an asynchronous fashion, using UNIX `sockets`. There is no indication, whether NodePrism allows to control different processes independently, but since NodePrism also uses a dedicated debugging process for each target process, and hence, overcomes the synchrony restriction of the UNIX primitives, it should be possible



An application is shown with 7 threads executing. Within the debugger, there are currently 4 threads, which control and examine selected target threads. Target threads 6 and 7 are grouped for debugging purposes and controlled by debugging thread IV, whereas threads 3 and 5 execute independently of the debugger.

Figure 4.2: Multi-Threaded Debugger

to provide this capability.

However, the creation of a complete UNIX process for every thread in the target application is an unsatisfying approach, especially in a shared environment, where multiple users use the same machine and resources are limited. Instead, a debugger implemented using user-level threads can provide independent target thread control more efficient.

GDB and Jacob's debugger do not show any asynchrony at all, thus, there is no possibility to independently control target threads.

4.4 Comparison

Table 4.1 summarizes how each surveyed debugger fulfills the discussed properties. The design that is proposed in the following chapter is able to fulfill each property, but the implemented prototype does not completely support targets that are operating in multiple

	NodePrism	LPdbx	GDB	Jacobs	new prototype
user-level threads	-	-	-	●	●
multiple kernel threads	●	●	●	-	●
multiple code images	-	●	-	-	○
multiple data sections	●	●	-	-	○
shared address space	-	-	●	●	●
independent control	●	●	-	-	●

Table 4.1: Properties of Interactive Source-Level Debuggers

address spaces. In detail, applications that use multiple address spaces can be handled, if all address spaces are on the same machine, but applications that are spread over multiple machines cannot be handled by a single debugger instance. Instead, an instance of the debugger has to be started on each machine. Partially achieving a property is denoted by an empty circle.

4.5 Technical Consequences

In this section, it is shown how the discussed objectives for a debugger for multi-threaded applications places certain requirements on the design of such a debugger.

All surveyed debugger solely rely on the usage of UNIX debugging primitives to communicate with the target application. The major reason for the usage of operating system services for debugging is safety. It is considered to be a general requirement for a debugger that neither code nor data of the debugger resides in the address space of the target application. A debugger should be independent from the target application, otherwise a bug in the application can cause the debugger's data to be overwritten and can prohibit further debugging.

It turns out that partial fulfillment of the postulated requirements is possible solely through usage of the UNIX debugging primitives. The UNIX debugging primitives operate on UNIX processes, hence, if an application consists of multiple code images generated by multiple UNIX processes, multiple system calls are necessary to wait for events in the target. Since these system calls are synchronous and events from multiple processes cannot be multiplexed, this can only be handled by creating a dedicated control processes for each target process, as done in LPdbx and NodePrism. However, in the case of threads sharing

a code image, a debugger cannot communicate with those threads asynchronously, because each access to the UNIX process stops all kernel threads, which in turn stops all user-level threads, thus the threads cannot be controlled independently. Therefore, it is impossible to support the full requirements without inventing new approaches.

When using even the extended `ptrace` call [7] or the `/proc` filesystem [31], the entire debugger is blocked when it is waiting to receive events from the target, like encountering a breakpoint. It is impossible to asynchronously deliver such events to the debugger and there is no mechanism for the debugger to poll for events in a non-blocking fashion. Instead, the debugger has to issue either the `wait` system call [33] (when using `ptrace`) or an `ioctl` system call [29] with a specific request, `PIOCWAIT` [31] (when using the `/proc` filesystem). Although for `/proc` there is another request type to wait for a specific kernel thread's events, this does not solve the problem, since multiple system calls cannot be multiplexed.

Besides the fact that `ptrace` and the `/proc` filesystem only offer a synchronous communication interface and that they are bound to one UNIX process, they cannot properly deal with the existence of user-level threads. Even a synchronous, thus sequential, debugger can only handle user-level threads with major efficiency degradations. More details regarding efficiency can be found in Chapter 5 and Appendix C.

Another significant problem exists if only UNIX debugging primitives are used. Traditional debuggers under UNIX create a breakpoint by inserting a special trap instruction into the target code and saving the original instruction(s). When the target executes, the debugger is blocked until some event happens in the target. Eventually, when a trap is hit by the target, a trap signal is raised. Instead of delivering this signal to the target process, the UNIX kernel stops the target, wakes up the debugger and notifies it about the event. Afterwards, the debugger can examine the state of the target, in particular the program counter register, to determine which breakpoint was encountered.

An efficiency problem exists when the target continues execution, but the breakpoint remains in the code. To continue the target, the breakpoint code is replaced by the original code and a breakpoint is set immediately at the next instruction, again by replacing and saving the code. Then the target is continued and this temporary breakpoint is hit. Control goes back to the debugger, which again replaces the temporary breakpoint and inserts the original breakpoint. Now the target can continue normally.

This procedure is reasonable if encountering a breakpoint involves user interaction, as it does for a sequential target. However, it is very inefficient for a multi-threaded target, if multiple threads of control execute the same code image. In this scenario, a breakpoint

might inadvertently be triggered very often before an appropriate thread hits the breakpoint and the user is notified. Each time a breakpoint is triggered by an inappropriate thread in the traditional scenario, several kernel context switches and code replacements take place before execution can continue. This introduces an efficiency bottleneck in debugging multi-threaded shared-memory applications.

4.6 Summary

The inability to debug multi-threaded shared-memory applications and control every thread independently is the major restriction of all surveyed debuggers. This work closes this gap.

To solve all these problems, another more flexible form of interprocess communication must be used to notify the debugger about events in the target. This in turn requires local code to be added to the target that can handle the necessary protocol. Future operating systems may offer different mechanisms that make this addition obsolete, but for this project a hybrid approach was chosen that distributes part of the debugger into the application. This approach is presented in Chapter 5.

Chapter 5

Design Aspects

The design of the debugger should make the functionality, as postulated in Chapter 4, possible. Furthermore, there are the following additional design goals:

- portability for different architectures and source languages
- interoperability with other debugging tools
- additional debugging support to handle a large number of threads

This chapter describes the design decisions made to accomplish these goals.

5.1 Solutions

Three new approaches are chosen to achieve the desired goals and to fill the gaps in previous approaches:

1. distributing the debugger
2. support for a general application model
3. runtime system cooperation

Additionally, parts of GDB code are re-used to save implementation time and to achieve portability with respect to symbolic data access.

5.1.1 Distributing the Debugger

The traditional approach of debugging involves controlling and manipulating the target application through operations provided by the operating system, like the `ptrace` system-call for UNIX [32] or the functionality of the `/proc` filesystem [31]. Unfortunately, using only these operations imposes significant problems, as discussed in Chapter 4, since these operations are only suited for a sequential debugger. Specifically, a traditional sequential debugger pauses when the application is running and vice versa, since the UNIX system services do not support asynchronous communication between the debugger and target.

The solution to achieve efficient asynchrony between debugger and target, as well as to deal effectively with user-level threads and shared memory applications, is a hybrid approach, which splits the communication into two independent channels.

One channel is created by using the usual debugging primitives. It is mainly used to remotely manipulate the target, like changing code or data, but it is not used to receive events from the target application. It is important to mention that, in the case of `ptrace`, this channel is only established temporarily, whenever an operation is requested. This temporary connection is necessary, because as soon as the debugger takes control over a target process using the `PTRACE_ATTACH` request, this target process cannot continue execution normally. Therefore, after the operation is finished, control of the target process is released using `PTRACE_DETACH`. A similar method is unnecessary for the `/proc` filesystem, since an established connection via `/proc` does not prohibit the target process from continuing normal execution. However, it is not possible to asynchronously wait for events from the target.

The second channel is established by distributing parts of the debugger into the target, called the *local debugger*. The local debugger catches all debugging events from the target application, like hitting a breakpoint, and reports them to the global debugger. As well, the local debugger performs various operations on behalf of the global debugger, making them more efficient than using remote access methods. The relationship between the global debugger and the local debugger is illustrated in Figure 5.1.

The communication between the local debugger and the corresponding control instance in the global debugger is by message passing through a `socket`. This communication channel enables communication that does not block the whole debugger neither the whole target. Therefore, it fulfills this important requirement for independent target thread control. Since the local debugger is added to the target and does not reside in a separate process, it can properly deal with shared memory applications. Applications that are spread over multiple address spaces can contain an instance of the local debugger in every

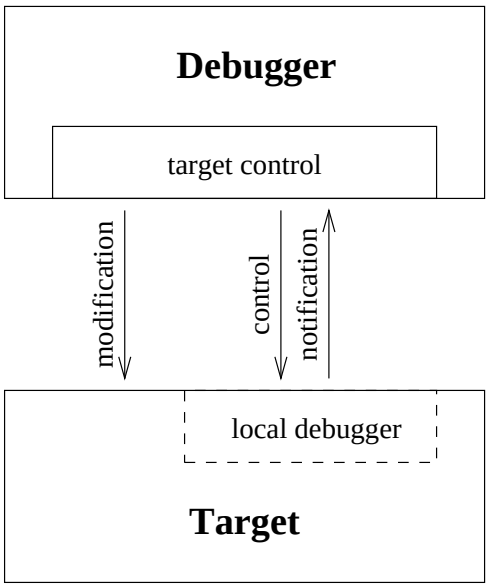


Figure 5.1: Distribution of Work between Debugger and Target

address space and the different event streams from the target threads can be multiplexed in the global debugger.

Using a local debugger allows many events that occur in the target to be handled in user-level code, so that the number of kernel context switches between debugger and target is decreased. This approach promises a significant increase in efficiency, which is important for certain kinds of debugging operations. For example, a single breakpoint often applies only to one or a small number of threads. However, in an application with many threads, a breakpoint may be inadvertently triggered by other threads thousands of times. The check to see if a breakpoint applies to the current thread can be done locally, and only if the breakpoint is applicable is the global debugger notified.

An important aspect of efficiency is whether user interaction takes place. Since users do not react in orders of microseconds, there is no need to have high efficiency when executing user commands. But, if code is executed without any user interaction, as it is the case in the described breakpoint scenario, the highest possible efficiency is deemed to be important. However, the highest efficiency can only be achieved when neither inter-process communication nor a context switch to the kernel is necessary.

Some restrictions arise when distributing parts of the debugger into the target application. It must be possible to preclude any safety leaks, introduced by having code and

data in the application's address space. The best solution is for the runtime system to provide a memory segment that can be used for the data of the local debugger and this data must not be overwritten by the target application. If the debugger's data cannot be protected, the data segment must at least be separated from the rest of the application's data, so that integrity checks can be performed on it. Then, additional redundancy can be introduced to recover from data losses.

A second restriction is the fact that on most platforms the hardware architecture or the operating system do not allow a user program to change its own code. Since breakpoints are mostly implemented by replacing a certain piece of code with a branch to the breakpoint handler, this precludes the local debugger from inserting or removing breakpoints. Therefore, the other communication channel, using the traditional operating system's primitives, is designated to perform these kinds of operations. A detailed and more concrete discussion of these issues can be found in Chapters 7 and 6.

At a first glance, the distribution of work might seem to introduce additional dependencies on the runtime system. But it increases portability, if many dependencies on the runtime system can be hidden by the local debugger and the communication between local and global debugger takes place on an abstract level. This approach might also seem to increase the probe effect, but it is overall more efficient, compared to the alternative of checking the applicability of a breakpoint in a different debugger process. Hence, the probe effect is not increased, but reduced.

To trigger a call to the local breakpoint handler, there are two possibilities, which were both considered and tested during the development of the debugger. The first alternative is to insert a trap instruction and handle it locally. This mechanism still involves the operating system kernel, but is approximately 36 times faster than checking the applicability of a breakpoint in the global debugger. A second alternative is to use a branch instruction to invoke the local breakpoint handler. This approach introduces some restrictions, but since it produced an additional speedup factor of 66 on top of the local trap handling, this approach is used in this project.

The speed tests for the two alternative mechanisms only simulate the case when a thread is continued immediately, which happens if the encountered breakpoint is not valid for this thread. Results of the speed tests can be seen in Table 5.1; details can be found in Appendix C. The speed tests were done on only one architecture (Solaris 2.3), but seem to have general applicability to different versions of UNIX. However, the design of the debugger ensures that this part can easily be changed to another approach, if for some architectures, the gains in speed do not justify the additional work that is necessary to cope with the increased complexity.

method	user time	system time	real time	factor (real)
local branch	15	4	15	1
local trap handler	245	756	999	66
traditional	637	3839	36511	2434

Table 5.1: Speed of Breakpoint Handling (in clock ticks)

A similar approach, used in a different context, is described by Kessler in [15]. Kessler also discusses the implications of implementing breakpoints in user-level code and presents speed tests that confirm the enormous speedup that can be achieved by local breakpoint handling. Differences among the breakpoint implementation presented in this work and Kessler's work are discussed in Section 6.1.

There is only one significant drawback for this approach: the maximum number of possible breakpoints per code image has to be specified at compile time of the application (the reasons are explained in Section 6.1). However, I do not believe this is a practical restriction.

Furthermore, the distribution of work is an essential design objective, otherwise it is impossible to realize the desired multi-threaded design in today's UNIX systems, and even in the future, it might never be possible for systems that create user-level threads.

5.1.2 General Application Model

As stated in Section 2.2, there is a large variety of application models for multi-threaded programs. All surveyed debuggers address only a subset of the possible programs. For example, GDB and Jacob's debugger are not able to deal with multiple address spaces, whereas LPdbx and NodePrism are not intended to debug shared-memory applications. The distribution of work, as it is introduced in the previous section, allows the following application view, which can handle most of the existing models, since it is not bound to operating system constraints.

The debugger only deals with user-level threads. If a kernel thread is not split up into several user-level threads, it is treated as executing only one user-level thread. User-level threads can be aggregated in a cluster, which in turn can use an arbitrary number of kernel threads to execute the user-level threads. Furthermore, it is guaranteed that all kernel threads, and hence all user-level threads in a cluster, operate in the same address

space with respect to data. Kernel threads either share a code image or have separated code images. In the case of μC++, each kernel thread is a UNIX process and has its own code image. If the notion of clusters does not exist, then every code image is treated as a cluster containing only one kernel thread. Finally, several clusters are aggregated into a *target entity*. It is guaranteed that all members of a target entity operate in the same data address space and that a target entity consists of only one executable file, and therefore all code images are the same. Also, every executable file contains an instance of the local debugger, which shares the data address space of the target entity. The debugger can handle an arbitrary number of target entities. In the following, some examples are given of how real world applications can be mapped into this model.

For a data parallel application on a multiprocessor machine consisting of a number of UNIX processes, every UNIX process is treated as a target entity that contains one cluster. The cluster contains one kernel thread executing one user-level thread. The same mapping can be used for a traditional networking application that consists of multiple UNIX processes on different machines.

An application that consists of three UNIX processes, which all contain multiple kernel threads given by a kernel thread library can be seen as an application consisting of three target entities, which contain one cluster each. Each cluster executes a number of user-level threads using the same number of kernel threads. All threads in one cluster share the same code image.

5.1.3 Runtime System Cooperation

To support user-level threads, a programming language or a thread package typically provides a runtime system that executes user-level threads by time slicing one or more kernel threads and performing context switches among the user-level threads. To effectively debug an application employing such user-level threads, some cooperation is needed from the runtime system. The most important aspects are the dynamic creation and destruction of the entities that are introduced by the programming language, like user-level threads and clusters. A cooperating runtime system must notify the debugger about such events during the application's execution, otherwise the debugger has to poll for them. A notification should also happen when the dynamic associations among kernel threads, user-level threads and clusters are changed, e.g., if a user-level thread is allowed to migrate from one cluster to another. Finally, there must be a mechanism to prevent a user-level thread from being scheduled for further execution when the thread is stopped at a breakpoint, as well as to resume execution when it is eventually continued by the global debugger.

The cooperation of the runtime system is necessary for an effective debugger, just as it is necessary to have operating system support for kernel threads from the operating system, even for a traditional, sequential debugger. The runtime system support can be defined as an abstract protocol, partly matching the protocol between local and global debugger. If there is a general scheme, i.e., which runtime events are reported to the debugger, then this increases the generality and portability of the debugger.

5.2 Portability

Portability across different operating systems and architectures is important, as well as portability for different source languages. Portability can be achieved by strict encapsulation of dependent code. Figure 5.2 illustrates the separation of the debugger into multiple modules to ensure certain portability aspects, which are discussed in this section.

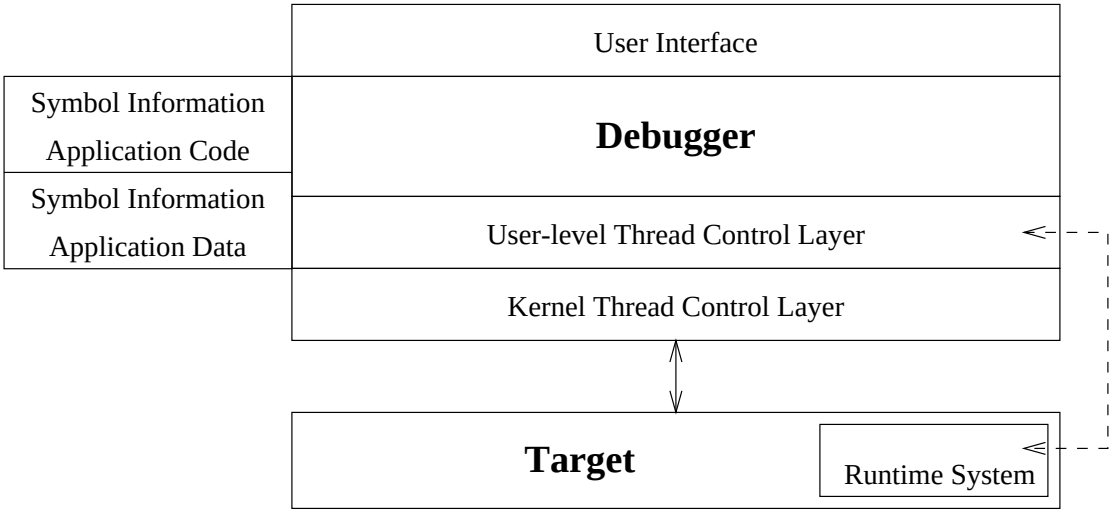


Figure 5.2: Portable Design

An important aspect of debugging is the availability of any debugging hardware support. Hardware support might be supplied for single-stepping, if a trace bit can be set in the processor status register, so that a trap occurs after the execution of every instruction. Another feature is monitoring memory access. In this case, a trap is raised on every access of one or more memory addresses.

Unfortunately, a debugger cannot rely on the existence of such features or even on a

common interface. Thus, the relevant code must be encapsulated so that it can be easily adapted to different architectures and still achieve optimal efficiency on each.

Furthermore, if little or no hardware support is given, these facilities must be emulated in software. Single-stepping can be achieved by setting a breakpoint after the current instruction. The breakpoints in turn are either implemented as a special instruction that raises a trap explicitly or as a call to a dedicated breakpoint routine. Monitoring memory access can be simulated in a way similar to single-stepping.

The operating system might also provide services to control the behaviour of an application, but again, the debugger can neither rely on their existence nor on a common functionality. However, in general, the operating system is the provider of communication services among kernel threads.

Clearly, code that controls kernel threads depends on the operating system, while the code that controls the user-level threads relies on the runtime system. Hence, the part of the debugger that communicates with the operating system, as well as the part communicating with the runtime system, have to be strictly separated from the rest of the debugger.

As well, the debugger works symbolically, therefore it must access the application's global symbol table and code debugging information, which is created during the compilation of the source program. The format of this information may depend on the source language as well as on the operating system. Thus, a separate module is designated for parsing the symbol information. To be even more flexible, a distinction between code symbols and data symbols is made, since they may require different treatment.

Finally, to let the debugger be flexible with regard to the desired user interface, the implementation of the debugger is separated from the implementation of the user interface, so that various styles of user interfaces can be built.

5.3 Interoperability

Event collection and graphical visualization of the behaviour of a concurrent program can help understanding the program as well as help the programmer to reason about faulty behaviour. Therefore it seems reasonable to integrate all these abilities into a debugging environment. To support this, the local debugger must provide additional functionality to report specified events or the results of statistical probes of the program's state to analysis or visualization tools.

However, code for analysis and visualization and code for interactive debugging must be able to interact, but be designed separately. Although this thesis deals mainly with interactive source-level debugging, these compatibility issues have to be kept in mind. Figure 5.3 shows the relationship between visualization and debugging.

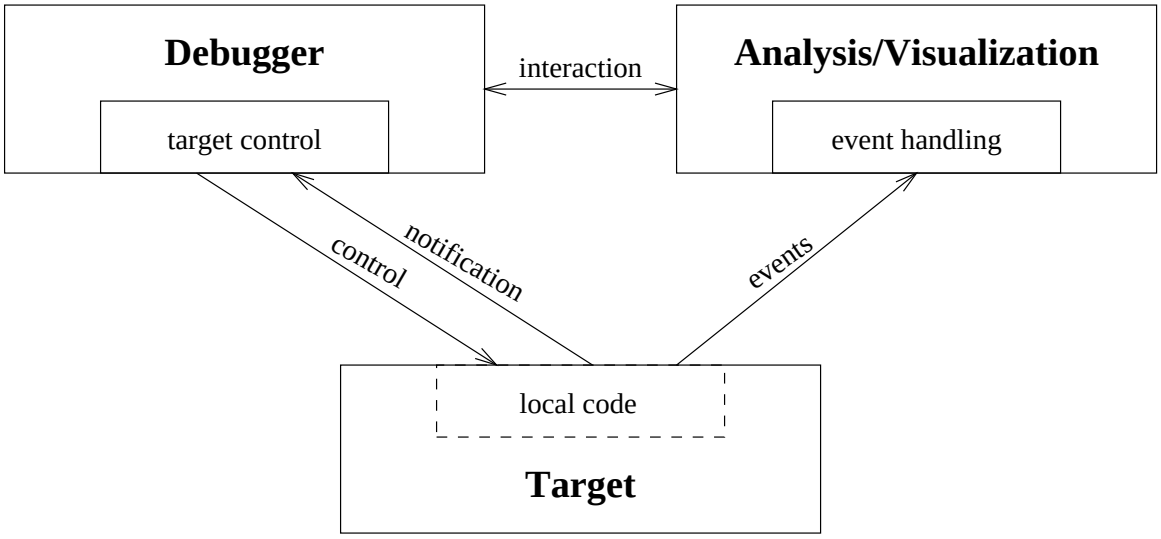


Figure 5.3: Interactive Debugging and Event Generation

5.4 Additional Support

Since the number of target threads can grow very large, it is very convenient to aggregate threads into a group and issue commands to a group of threads, as well as watch the behaviour of a group of threads. Two different kinds of grouping are identified. An *operational group* denotes aggregating multiple threads into a group for the purpose of directing operations to all threads in the group. A *behavioural group* is a group of threads where events in only a subset of these threads can have an impact on all threads in the group. For example, if one thread hits a breakpoint, then all threads of this group are stopped, regardless of where they are executing.

5.4.1 Operational Grouping

Threads can be arbitrarily grouped into an operational group, but certain commands can be applied only if they all share the same source code. For example, without sharing the

same source code, it would be meaningless to specify a breakpoint in the source code for all members of the group.

Threads can belong to more than one group and do not lose their ability to be started and stopped separately. Operational grouping is mainly a convenience facility to simplify the task of applying the same command to a number of threads.

5.4.2 Behavioural Grouping

Threads can be arbitrarily grouped into a behavioural group, but no thread can belong to more than one behavioural group of threads. It is possible to integrate groups into new higher-level groups or decompose a group into multiple groups. When a thread belongs to a behavioural group, it is no longer possible to start and stop this thread separately. Instead, the state of the thread is always determined by the state of the whole group.

In Section 3.2, event-based approaches for debugging multi-threaded applications are presented. One aspect of these approaches is to compare the expected and the actual behaviour of an application on a higher level, where multiple threads that perform a certain task are aggregated for debugging purposes. Behavioural grouping supports this approach, for example, all threads in a group are stopped, if one thread encounters a breakpoint.

No more attention is paid to behavioural grouping, since it is beyond the scope of this work, which only attempts to establish basic support to effectively control the behaviour of a multi-threaded application.

5.5 Using GDB Code

Access to code symbol information, as well as to data symbol information is encapsulated by two separate modules to increase the portability of the debugger. To achieve the highest level of portability and to avoid unnecessary implementation work, parts of the GDB debugger [26] were re-used.

The code that accesses the symbol information in the executable file must be split into two layers. The lower layer deals with the different file formats for executable files, like *a.out, elf*, etc. and enables access to the global symbol table. Part of the GDB source distribution is a library called *libbfd* [34], which is well-suited to provide this layer.

If an executable file is build using the debug option (usually the `-g` flag in the command line), it contains additional debug information. The upper layer deals with the format of this additional debug information itself. Unfortunately, there is a large variety of different formats and the format depends on the source language, the compiler and even the assembler [2] [36]. Therefore, this second layer provides transparent access to the debug information. Because GDB provides this access on a variety of different platforms and configurations, GDB code is used to access the debug information, instead of doing a new implementation.

Additionally, GDB code is used to look up the contents of variables and to interpret the data on the execution stack of a thread. The output is captured and presented in the debugger's user interface.

GDB is not run as a separate process. Furthermore, all parts of GDB code that can interfere with the main debugger by calling operating system functions to gain control over a target's UNIX process or to manipulate the target must not be called, so they are removed.

There are two drawbacks of reusing GDB modules. The design of GDB is tightly integrated, so it is non-trivial to use just the necessary parts of the code. This problem is solved by creating a library that contains the needed source code and excludes all potentially harmful code. The second problem is the size of this code and the necessity to configure and compile GDB before the multi-threaded debugger can be built.

However, because of the following two advantages, this approach seems to be a reasonable strategy. First, large parts of code are re-used, instead of doing a new implementation for symbolic data access. Second, it is possible to benefit from future GDB development, especially with respect to portability for different platforms. The idea for this solution is based on a library providing the whole GDB functionality that is described in a document distributed with the GDB sources [35]. Unfortunately this document is somewhat outdated and the library cannot be built. Hence, significant additional work was necessary to accomplish this task.

Chapter 6

Breakpoints in User Code

6.1 General Aspects

As stated in Section 5.1.1, it is desirable to execute the first step of breakpoint handling, i.e., deciding if the breakpoint is applicable, in user code. Figure 6.1 shows the first step of local breakpoint handling in C-style and assembler pseudo code.

The basic idea is that breakpoints are realized by inserting a call to a dedicated breakpoint routine. The original instructions at the breakpoint location are saved and if the breakpoint is not applicable for the current thread, they are executed at the temporary location. Using this scheme, there is no additional code replacement nor an address-space context-switch necessary, if a breakpoint is not applicable.

To keep the call to the breakpoint handler as simple as possible, the breakpoint number is not passed to the breakpoint handler. Instead, N dedicated breakpoint handler routines like *breakpoint_handler_N()* are created, one for each breakpoint. If a breakpoint is set, then a call to the appropriate breakpoint handler, depending on the breakpoint's number, is inserted in the application's code.

To access the information about the applicability of a breakpoint as fast as possible, the runtime's data structure for a user-level thread is augmented with one additional field that contains a bitmask for all possible breakpoints. If a breakpoint is applicable for the thread, a bit is turned on by the local debugger. In the pseudo code, this is denoted by the field *bp_check* in *UserLevelThread*.

To handle breakpoints, the user-level state of the thread must be saved, so that execution can eventually continue from the same state. This is mainly achieved by saving all

```
class UserLevelThread {
//  ...
    char        bp_check[maximum_bp_number / 8];      // assume a char takes 8 bit
//  ...
};

bool breakpointHandler( int no ) {
    sendMessageToGlobalDebugger( no );
    bool breakpoint_removed = receiveContinueMessageFromGlobalDebugger();
    return breakpoint_removed;                                                  10
}

void breakpoint_handler_N() {
    saveApplicationState();
    // this_thread is a pointer to the currently executing user-level thread
    if ( this_thread->bp_check[ N / 8 ] & (1 << (N % 8)) ) {
        if ( breakpointHandler( N ) ) {
            // Decrease the return address to execute the restored code at its original location.
            // SPARC specific: -8. RA denotes the register where the return address is stored. If
            // the breakpoint is still implemented for other threads, it is triggered again, but   20
            // since the bitmask is already changed, this just slightly delays execution.
            asm(" sub RA, -8, RA ");
            restoreApplicationState();
            return;
        }
    }
    restoreApplicationState();
    asm("
        ! reserve as many NOP instructions as needed to store the temporary code
        nop                                                                     30
        ...
        ! final return to jump back into the application
        jmp address_after_breakpoint_code
    ");
}
```

Figure 6.1: Breakpoint Handling in User Code

registers and loading them back immediately before execution continues. In Figure 6.1, this is shown by a call to *save_application_state()*. Depending on the architecture, some or all registers are automatically saved upon the branch to the breakpoint handler. Only

the rest of them are saved from within the breakpoint handler.

When a breakpoint is hit by any thread of the target application, the corresponding nth breakpoint handler is called. Then, the nth bit is checked in the bitfield of the current thread's data structure. If it is set, the breakpoint is valid for the current thread, the global debugger is notified by the call to `breakpointHandler`, and the target thread is blocked from further execution.

When a *continue* command is issued for this thread or the breakpoint is not valid in the first place, execution continues. If the breakpoint is removed in the meantime, the original instructions are already restored at the original location, but the address that is used for the `return` instruction does not match this value and has to be adjusted backwards. This adjustment can be seen in line 22 of the pseudo code. The application's state is restored and execution continues. If the breakpoint is not removed, the original instructions have to be executed within the local breakpoint handler.

Since an application can not easily change the size of its code segment in memory, placeholders for the original instructions must be added at compile time. Therefore, every breakpoint handler function reserves some assembler instructions to hold the original instructions, so that they can be executed at this temporary location. The fact that a fixed number of breakpoint handler functions are added to the target application is the reason why the maximum number of breakpoints is limited, but can be set arbitrarily when the application is compiled.

To execute the instructions at the temporary location, the state of the target is restored by a call to *restore_application_state()*, execution continues and a final jump transfers control back into application code at the address following the call to the breakpoint handler.

A similar approach for breakpoints handled in user code is presented and discussed by Kessler in [15]. Kessler starts from the same precondition that is true for breakpoints for a multi-thread application: Substantial gains in execution of the breakpoint code justify spending more time on implementing and removing a breakpoint. Both approaches are compared in the following two sections.

6.2 Restrictions on Breakpoints in User Code

This section decribes a number of restrictions when using local breakpoint handling that does not involve the kernel, most of which forbid certain kinds of code optimization.

Nevertheless, the advantages in speed justify the restrictions and the additional effort used to work around most of them.

6.2.1 Runtime System

Because breakpoint handling interacts with the part of the runtime system that handles the scheduling of the user-level threads, this part of the runtime system can not be debugged using these kind of breakpoints. This seems to be an acceptable restriction; a traditional sequential debugger can be used to debug this part, if necessary.

6.2.2 Leaf Procedures

Implementing breakpoints by a call to a local handler places certain restrictions on the target application. One aspect of Kessler's approach [15] is that it uses the `branch` instruction of the SPARC processor [12], which cannot span the whole address range of the processor. Kessler's approach circumvents this by ensuring through the linking and loading process that the breakpoint handler is reachable within the possible range. A similar mechanism was considered during this work, but because of the debugger's portability goal, this additional intrusion into linking and loading was unacceptable.

Usually, a microprocessor offers a mechanism to transfer control to almost any position in the addressable code memory. This mechanism is used for this work.

On the Alpha AXP microprocessor [24] this instruction is the `jmp` instruction and on the MIPS architecture [14] the `j` instruction. On both architectures such a jump can be used to branch to the local breakpoint handler, as well as to return back to the application at the end of the handler. Both architectures provide a slightly different version of a jump that stores the current program counter value (return address) in a dedicated register and is used for normal routine calls.

Unfortunately, the SPARC architecture only provides the second type of jump, the `call` instruction, which is used to realize subroutine calls. Therefore, the return address of the call instruction is always stored in a dedicated register and can be used by a normal subroutine to perform the return jump.

If a routine itself makes calls to other subroutines, it must store the contents of this dedicated register on the stack and pop it off the stack, before the return instruction is executed. Code for this is usually added by a compiler automatically for every routine in a program. One usual optimization option is to identify *leaf procedures*, i.e., routines that do

not perform calls to subroutines, and remove the code for saving and restoring the contents of the return address register from those routines. This optimization must not be used on the SPARC, otherwise a breakpoint in an optimized leaf procedure would overwrite the return address for the procedure, causing the program to fail. Another possibility would be to disallow setting breakpoints in optimized leaf procedures, but this is not considered in this work, since debugging happens on non-optimized code in most cases, anyway.

6.2.3 A Race Condition

On a RISC architecture, the target address for a control transfer instruction is either calculated implicitly (SPARC, MIPS) and the instruction after the call is executed in the *delay slot*, or the target address has to be calculated explicitly and given to the jump instruction in a register (Alpha AXP). In both cases, to insert a call to a local breakpoint handler, more than one instruction has to be changed. In case of delayed instructions, the instruction after the jump has to be set to `nop`, in the other case it is obvious that the target address calculation has to be implemented. This places slight restrictions on the granularity of breakpoints, for example, it is impossible to set two breakpoints at two consecutive addresses.

A more serious implication is a possible race condition. If any thread is executing in between these instructions or if, because of time-slicing, a thread is interrupted in between the instructions, when they are changed, the thread executes only a part of the changed instructions after it is eventually resumed. This is likely to fail. This race condition exists during implementation and removal of a breakpoint. Kessler proposes the use of the *annul* bit of the SPARC processor to reduce the size of the jump to the breakpoint handler to one instruction, which precludes this race condition, since one instruction is either completely executed or not executed when a context switch happens or a UNIX process is stopped by a debugger. If the annul bit is set in an unconditional branch instruction (`ba,a`), execution of the instruction in the delay slot is suppressed. However, this solution is not sufficient, because, as stated above, a branch cannot span the whole address range. Additionally, on processors like the Alpha AXP the target address must be calculated explicitly, which is not addressed by this approach, since it is only suited for architectures that execute delay instructions while calculating the target address of a branch. Furthermore, even if a RISC processor uses delayed branches, there is no guarantee that such an annulment exists. For example, the MIPS architecture, that has delayed branches, does not provide the possibility to annul instructions in the delay slot after an unconditional branch.

Therefore, another solution is used to handle this race condition: Every user-level

thread's program counter is checked before a breakpoint is inserted or removed. If the race condition is detected, the operation is refused and has to be tried again. The current implementation automatically retries the operation. The error-and-retry approach is satisfactory, since the probability of this race condition occurring is very small and it always exists only temporarily. A thread can be blocked through synchronization mechanisms of the runtime system and because of an error in the program it may never continue execution. However, in this case its execution location is within the runtime system and breakpoints are not allowed to be set there (see section 6.2.1). Because the biggest part of the check is done in the local debugger, it is not necessary to transfer the values of each thread's program counter between UNIX process boundaries, so it is efficient enough not to introduce a significant delay into this operation, which is in response to user interaction.

The check requires support from the runtime system. In detail, for every user-level thread that is not executed by a kernel thread the, program counter at which the last context switch occured must be saved. The algorithm for the whole breakpoint insert and removal procedure is explained in Section 8.1.

6.2.4 Re-using a Breakpoint Handler for a New Breakpoint

When a breakpoint is no longer valid for any thread, the code changes are undone and the local handler can be used for another breakpoint. This reuse is necessary, since the number of breakpoint handlers is finite, therefore new breakpoints must use the same handler routines as previous ones. A problem can occur under the following circumstances:

1. Thread A encounters breakpoint X, calls `breakpoint_handler_X` and is blocked.
2. Breakpoint X is removed for all threads.
3. Breakpoint X is re-used for a different breakpoint.
4. Thread A is continued.

When the thread is eventually continued, expecting that the original instructions can be executed at the temporary location, it executes the wrong instructions and the application fails.

The following mechanism, which is also described in the comment in line 18 of Figure 6.1, prevents this from happening. When a thread is continued, a flag is sent to the local debugger, indicating whether the last encountered breakpoint is reset for this thread.

If the flag is set, the thread continues execution not in the local breakpoint handler, but at the location where the breakpoint was set. If the breakpoint is completely removed, then the original code is already restored and execution continues normally. If not, control is transfered back to the local breakpoint handler, but since the breakpoint bitmask is already changed, the thread is not blocked and the original code is executed at the temporary location.

On the other hand, if a breakpoint handler is re-used for a new breakpoint, a race condition is possible for the temporary code, if a thread is executing the temporary code and the breakpoint was not valid for this thread in the first place. This race condition is similar to the race condition for the instructions at the breakpoint's address (see Section 6.2.3), but it is even less probable, since it is unlikely that a breakpoint is re-used fast enough to ever trigger it. However, this race condition is handled in the same way, i.e., a check is performed, mostly locally, if any thread is executing or blocked in the local breakpoint handler and the operation is refused and retried when the situation is detected.

6.3 Saving/Restoring the Local State

As described in Section 6.1, the local state of the application must be saved upon entry of the local breakpoint handler. On the SPARC, the normal register window is saved using the `save` instruction, which is automatically inserted by the compiler at the start of each function. Additionally registers must also be saved. This is done by the code shown in Figure 6.2.

```
or %g0,%g1,%l5          ! save global register
ta ST_GETCC             ! get condition codes
or %g0,%g1,%l6          ! save condition codes
rd %y,%l7               ! save y register
```

Figure 6.2: Saving the Application's State

The global register `%g1` can be used by an application to store temporary data. Therefore, it must be saved in the breakpoint handler. The trap instruction in line 2 gets the condition code bits from the processor status register and stores them into `%g1`, which is subsequently saved. Finally, the `%y` register, which is used temporarily for multiplication and division instructions, is saved. The local registers `%l5`–`%l7` are not used in the local

breakpoint handler, thus they can be used to hold the application's register values. If the breakpoint is applicable and the local breakpoint handler calls into the local debugger to notify the global debugger, these registers are saved using the normal SPARC window mechanism. When the thread is continued, the reverse instructions are executed, before control transfers back into the application or before the temporary code is executed. These instructions can be seen in Figure 6.3.

```
wr %g0,%l7,%y                    ! restore y register
or %g0,%l6,%g1                   ! set condition codes
ta ST SETCC                      ! reinstall condition codes
or %g0,%l5,%g1                   ! restore global register
```

Figure 6.3: Restoring the Application's State

Chapter 7

The Debugger's Design

In this chapter, the following notions from μC++ are used to distinguish abstract data types with respect to concurrency.

A *class* denotes an abstract data type that has no concurrency properties.

A *monitor* is a class where the invocation of public member routines is guaranteed to be mutually exclusive. Additionally, a monitor provides mechanisms to schedule threads externally by controlling calls to mutex member routines or internally by condition variables.

A *task* is considered to be an active monitor, i.e., each task object has its own thread of control associated with it. Since a task also has the mutual exclusion properties of a monitor, member routines of a task can only be called when the task's thread of control accepts these member routines or blocks internally.

7.1 Static Design

The debugger is separated into different modules according to the outline show in Figure 5.2. Each module consists of a set of classes. An object-oriented design ensures the re-usability of the abstract parts of the debugger.

7.1.1 Symbol Access Modules

The encapsulation of the address space property is done with respect to data sections and code sections as stated in Section 5.2. In the current implementation, both classes

use the functionality of the `AccessGDB` monitor that hides the details of using GDB code. `AccessGDB` serializes access to GDB code, since that code is not thread-safe.

CodeSymbols

An instance of class `CodeSymbols` represents a section of code in the target application that is built from one executable file. This class provides methods to look up the memory addresses of locations in the source code, specified either by a function name or by a source file and a line number. It returns an object of class `CodeAddress`. The class `CodeAddress` must be adapted to each target system to properly represent a code address. For example, if an application is spread over multiple code images, it must contain information to identify the code image. In the case of μC++, a `CodeAddress` object is just a plain pointer value, since all code images are built from the same executable file.

DataSymbols

An instance of class `DataSymbols` has the corresponding functionality as `CodeSymbols`, but delivers the memory addresses of data symbols in the target application. It also has methods to look up the contents of objects in the application, as well as getting the stack backtrace of a user-level thread. The latter methods rely on GDB code, but since GDB cannot distinguish between different user-level threads, additional information has to be supplied, which is used to let the stack of the chosen user-level thread appear to be the only stack in the application, so that the correct information is returned.

7.1.2 Kernel Thread Control

The following classes are designed to provide an abstract interface for the low-level debug operations to enhance the portability of the debugger, as outlined in Section 5.2.

KernelThread

An object of type `KernelThread` represents a kernel thread in the target application and has the same lifetime. This class offers methods to control a kernel thread, like `stop`, `continue`, `kill`. Additional methods are available to look up or change a kernel thread's registers, `readRegisters` and `writeRegisters`. The methods `stop` and `continue` keep track of whether a kernel thread is already stopped, so that repeated attempts are not

made to stop a kernel thread. Nested calls to `stop` and `continue` are allowed, but the number of `continue` requests must match the number of `stop` requests, before a kernel thread is actually continued.

An object of `KernelThread` can be created in two variants. A flag, given upon creation, indicates whether the kernel thread has its own private code image. If yes, it also provides a set of methods to inspect and change code and data private to a kernel thread. If not, i.e., if the kernel thread shares the code image with other kernel threads, the following methods have no effect. These methods are `readCode`, `writeCode`, `readData` and `writeData`. In detail, these operations invoke the corresponding `ptrace` or `/proc` system calls, depending on which service is available on the particular architecture. If the `/proc` filesystem is available, it should be the first choice, since it is the newer mechanism and considered to be more efficient [16].

The ability to create a restricted object of type `KernelThread` is used in the current implementation to deal with the μC++ uni-processor mode (see Appendix A), where the runtime system creates and reports multiple data structures for kernel threads, but only the first one is associated with an actual UNIX process. It might also be used for systems that create multiple kernel threads that use the same code image.

Additionally, there is a method `migrate` that performs all necessary updates if a kernel thread migrates from one cluster to another.

Cluster

When multiple kernel threads that are aggregated in a cluster (see Section 2.2) use private copies of the code image, each operation that applies to code images, like altering code, must be executed on each kernel thread's private code image. Additionally, access to multiple kernel threads in a cluster must happen consistently, so that all threads appear to execute a single code image. Therefore, operations on a `KernelThread` object are not invoked directly, but through corresponding methods of the appropriate `Cluster` object. These methods are `writeCode`, `readCode`, `writeData` and `readData`.

When data is looked up or modified, this has to be done for only one kernel thread, since according to the application model from Section 2.2 all kernel threads in a cluster share a data address space. For looking up code it is also sufficient to perform this operation on only one kernel thread, since all kernel threads belonging to the same target entity have the same code images (see Section 5.1.2). However, modification of code has to be done for all code images to keep them consistent.

Before any code or data is looked up or changed, all kernel threads belonging to a cluster are stopped to achieve consistency. This is automatically done when one of the methods is called. However when a client of a `Cluster` performs multiple calls, it increases efficiency significantly if the kernel threads are stopped and restarted only once. Therefore, the public methods `stopKernelThreads` and `contKernelThreads` are available. Nested calls to `stopKernelThreads` and `contKernelThreads` are allowed, because the `KernelThread` class keeps track of the number of stop and continue requests. This allows for a high-level function to explicitely stop and restart the kernel threads in a cluster for efficiency reasons, even if subsequently invoked low-level functions call these methods again.

If a breakpoint is requested for multiple user-level threads in the same cluster, it must be implemented only once. The methods `setBreakpoint` and `resetBreakpoint` keep track of this. The details of implementing and removing a breakpoint are described in Section 8.1.2.

The abstract data type `Cluster` is a major synchronization unit. To ensure that multiple accesses to kernel threads do not interfere with each other, `Cluster` is implemented as a monitor. A cluster can be acquired and released explicitly, using the methods `Acquire` and `Release`. This allows multiple operations to be executed atomically, which is especially important if a client makes use of the possibility to stop and restart all kernel threads once for multiple method calls. Calls to `Acquire` and `Release` can be nested, but their number must match in order to release the `Cluster`.

On the other hand, this mechanism implies that multiple method calls are necessary, after a thread has acquired the lock. Hence, `Cluster` is an *owner monitor*, i.e., the thread that acquires the monitor is allowed to call in when the monitor, in principle, is closed. This mechanism is also useful, because there are situations where a function is called from within `Cluster` and subsequently calls back into `Cluster`.

Breakpoint_$architecture

The class `Breakpoint_$architecture` (where `$architecture` is replaced with the appropriate CPU name) is inherited from the `Breakpoint` class (see below) and hides the breakpoint related details of the current machine architecture from the rest of the application. It generates the correct breakpoint instructions, as described in Section 8.1.1. It also looks up and stores the original code that is overwritten by the breakpoint code. The results can be accessed in a generic way through the interface of the `Breakpoint` superclass.

7.1.3 User-level Thread Control

In this part of the debugger, the main abstraction is a user-level thread. User-level threads can be manipulated and controlled in the same way as UNIX processes in a traditional debugging environment.

ULThread

`ULThread` is a task, instances of which represent a single user-level thread in the target application. It provides the functionality to control the behaviour of the corresponding target thread and encapsulates the details that depend on the runtime system. Basic methods are `step` and `next` for single stepping, `stop` and `cont` to change the state of the target thread, `setBreakpoint` and `resetBreakpoint` to set and remove breakpoints for the corresponding target thread, as well as `printBacktrace` and `requestSymbolInformation` to look up data.

Additionally, there are methods that are called by the `TargetEntity` (see Section 7.1.4) task to deliver events from the local debugger. These methods are `breakpointHit`, called when the target thread encounters a valid breakpoint and `migrate`, which is called when the target thread migrates from one cluster to another.

All operations are executed with the thread of control of `ULThread`; this allows the control and manipulation of different target threads independently. The general mechanisms for methods of `ULThread` are presented in Section 7.2 and algorithms for complex operations are discussed in Chapter 8.

An object of `ULThread` stores all data that is necessary to access and control the corresponding thread in the application. This includes a pointer to the appropriate `Cluster` object and the current register contents. This information is updated by messages from the local debugger when migration is reported or a breakpoint is encountered. Alternatively, a snapshot can be taken from a running thread. Taking a snapshot of a running task allows the lookup methods `printBacktrace` and `requestSymbolInformation` to be invoked, even when the corresponding target thread is not stopped by the debugger. While the use of this information is rather limited if the thread is executing, it is deemed to be helpful, if a thread is blocked due to synchronization. Especially in a deadlock situation, this information might help to gain insight about the reasons for the deadlock.

ThreadGroup

An object of task `ThreadGroup` is created whenever the user aggregates target threads into a group. It provides a subset of methods from `ULThread` that are applicable to a group of threads. Requests to a `ThreadGroup` are dispatched to the appropriate `ULThread` objects using the thread of control of `ThreadGroup`. The class `ThreadGroup` implements an operational grouping facility as described in Section 5.4.1.

AbstractThread

`AbstractThread` is an abstract base class for both `ULThread` and `ThreadGroup`. It declares the common parts of the interface and implements common functionality.

Breakpoint

This class is an abstract base class that provides an interface to access all information needed to implement and remove a breakpoint in the application. For every actual breakpoint that is set in the target application, an object of the appropriate subclass of `Breakpoint` is created. If the same breakpoint is set for multiple user-level threads, only one object is created. To achieve this, `Breakpoint` objects are created and destroyed by a dedicated server object. This functionality is available in the `TargetBreakpoints` monitor (see below).

7.1.4 Main Debugger

The main debugger uses the abstract interface given by the user-level thread control classes. It administers the relationship between entities in the target application and the corresponding objects in the debugger. It also provides the main user interface.

DebuggerMain

There is only one object of task `DebuggerMain`, which is the main object of the debugger. It creates a socket server object that accepts connections from local debugger instances. Whenever a local debugger instance registers at the global debugger, a corresponding `TargetEntity` object is created. A main user interface is provided that lists all user-level threads that currently exist in the target application. `DebuggerMain` provides the thread

of control to serve requests from this user interface. From the list of target threads, single threads or groups of threads can be selected and inspected separately.

TargetEntity

An object of task `TargetEntity` exists for every local debugger instance that registers with the global debugger. It creates the objects that communicate with the local debugger, `SocketReceiver`, `EventBuffer`, `EventCourier` for receiving data and `SocketSender` for sending data. A `TargetEntity` object is an administrator with its own thread of control. It dispatches all incoming requests to the appropriate objects in the debugger. It contains a monitor of type `TargetBreakpoints`, which is a server object that manages the creation and destruction of `Breakpoint` objects. Every object of type `TargetEntity` creates one `CodeSymbols` and one `DataSymbols` object corresponding to the local debugger's code image and data address space.

7.1.5 Communication Classes

SocketSender

The monitor `SocketSender` is used to communicate with the local debugger. All objects call the method `sendToLocalDebugger` to deliver their messages. If multiple messages have to be sent and received without interference from other threads, this is supported by the concept of an *atomic operation*. An atomic operation is invoked by calling the `startAtomicOperation` method before the operation is started. `finishAtomicOperation` is called at the end. An atomic operation guarantees that no other thread of control can send requests in the meantime.

Nested calls of `startAtomicOperation` are allowed, an internal counter keeps track of the number. It is decreased on every call of `finishAtomicOperation` and has to reach 0 to eventually finish the atomic operation.

SocketReceiver

`SocketReceiver` is a task that reads incoming events from the socket. It stores the events in a monitor object of type `EventBuffer`. A dedicated courier task, `EventCourier` takes the events and delivers them to `TargetEntity`. The only exception is the confirmation from the local debugger for an atomic operation, which is not enqueued in `EventBuffer`, but delivered directly to the `SocketSender` object.

7.1.6 User Interface

The user interface is built by various objects, each of which represents a window in the user interface. The three most important ones are `MainInterface`, the main user interface that cooperates with `DebuggerMain`; `ThreadInterface`, of which an object is created for every object of type `ULThread`; and `GroupInterface` that belongs to `ThreadGroup`. Each interface object represents a window on the user's terminal and delivers the user's requests from this window to the appropriate debugger objects.

7.1.7 Local Debugger

The concept of having a local debugger instance added to the target application is introduced in Section 5.1.1. This section presents its realization.

The names of the classes follow the μC++ naming convention (prefix "u" followed by a capital letter) to avoid name conflicts, since their code is linked with the application.

uLocalDebugger

One object of the class `uLocalDebugger` is created in each part of the target application that is stored in a separate executable file. The following methods are called by the runtime system: `createCluster`, `destroyCluster` for creation and destruction of clusters, `createKernelThread`, `destroyKernelThread` for creation and destruction of kernel threads, `createULThread`, `destroyULThread` for creation and destruction of user-level threads, as well as `migrateKernelThread` and `migrateULThread` to notify the debugger about migration of threads to different clusters. The corresponding messages are sent to the global debugger and, if necessary, the calling thread is blocked until the global debugger sends a response.

Another method, `breakpointHandler` notifies the global debugger about encountering a breakpoint. It is called by the appropriate method of `uLocalDebuggerHandler`, if a breakpoint is applicable for a user-level thread. The thread is stopped until the user issues a continue command.

uLocalDebuggerTask

An object of task `uLocalDebuggerTask` belongs to a `uLocalDebugger` object, has its own thread of control, and receives and dispatches the messages that are sent from the global

debugger. When a request for an atomic operation is sent by the global debugger, the task sends a confirmation and enters a special state accepting the messages belonging to the atomic operation and creating responses. During an atomic operation, `uLocalDebugger` is locked, so that no events for the global debugger are produced in the meantime.

uLocalDebuggerHandler

The class `uLocalDebuggerHandler` provides N methods named `bp_handler_`n, corresponding to the `breakpoint_handler_N()` function that is shown in Figure 6.1 in Chapter 6, for N possible breakpoints, where the applicability of a breakpoint to a user-level thread is checked. Calls to these methods are inserted in the application, if a breakpoint is set. Space is reserved, by some `NOP` instructions, so that the original code replaced by the call to the breakpoint handler can be stored and subsequently executed in these functions. The source code for this class is created by a shell script, so that it can be adapted to any maximum number of breakpoints.

7.2 Dynamic Design of the Main Debugger

In this section, some central mechanisms are described to demonstrate how certain actions take place in the global debugger. For all figures that are shown in this section and the following sections, double frames indicate that a thread of control is associated with an object; a dashed outer frame denotes that the object is a monitor.

7.2.1 Changing Code in the Target Application

To implement and remove breakpoints in the target application, code has to be changed using methods of the `KernelThread` class. If a breakpoint is set for a user-level thread, the code has to be changed for all code images belonging to the corresponding cluster, since the user-level thread can be executed by any kernel thread in the cluster. In μC++, every kernel thread is created by a UNIX process, and hence has a private code image, but all code images the same. Also, code changes have to happen consistently and atomically. Therefore, code change requests are executed through the `Cluster` object that represents the cluster in the target application.

Figure 7.1 shows two `ULThread` objects that request a code change. Only one object is allowed to call into the monitor `Cluster`. Inside of `Cluster` is a check whether this

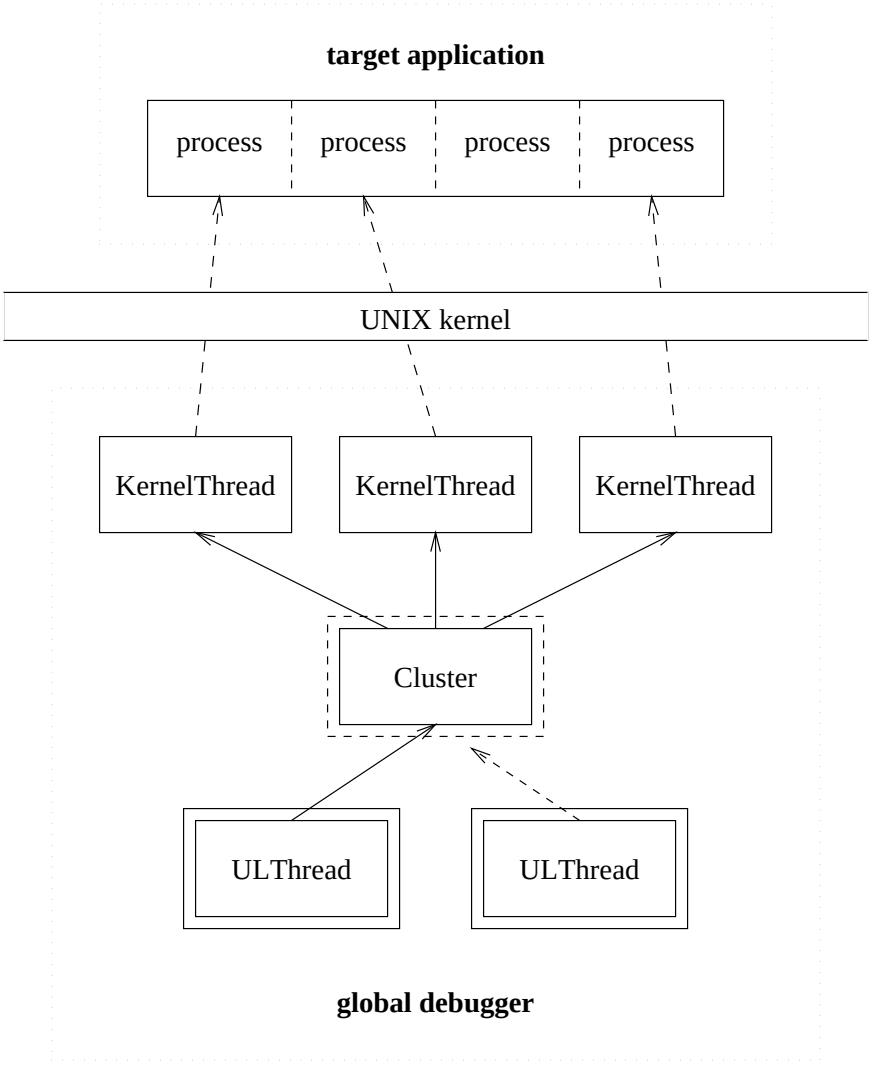


Figure 7.1: Changing Code in the Target Application

breakpoint is already implemented, i.e., set for another user-level thread on this cluster. If necessary, the code is changed for all kernel threads belonging to that cluster, before the other `ULThread` is allowed to set a breakpoint. As discussed earlier, the actual change is done using operating system services that are encapsulated by the `KernelThread` class.

In Section 6.2.3, a race condition is discussed that can occur during setting a breakpoint in the target application. Section 8.1.2 presents the mechanism used to solve this problem. Unfortunately, it turns out that this solution does not fit very well in the existing design, since its implementation is dependent on the runtime system. To maintain portability, the

function that performs this check should belong to `ULThread`. On the other hand, direct access to the kernel thread objects is needed, and from the dynamic point of view, the function is only invoked if a `Cluster` object really changes code to implement a breakpoint. Therefore, an internal method is added to `Cluster` that implements the test for this race condition.

7.2.2 Setting/Resetting a Breakpoint

Figure 7.2 shows the insertion of a breakpoint in the target application. If a breakpoint is removed, similar steps are applied in reverse order. During setting of a breakpoint, the code in the application is only changed if necessary, i.e., if this breakpoint is not already implemented for another user-level thread. A bitmask is used in the local debugger to check for the applicability of a breakpoint to a user-level thread (see Section 6.1). When a breakpoint is set, it is updated only in the global debugger. However, before the thread is continued, the updated bitmask is transferred and installed by the local debugger.

In detail, these are the steps to set a breakpoint:

1. The user request for setting of a breakpoint is delivered to the `ULThread` object.
2. The corresponding breakpoint is requested at the appropriate `TargetBreakpoints` manager.
3. If no such `Breakpoint` object for this address exists, `TargetBreakpoints` creates one.
4. The number of the breakpoint is delivered back to `ULThread`.
5. `ULThread` implements the breakpoint by a call to the `Cluster` object corresponding to the cluster in which the user-level thread is currently executing.
6. The code and the address of the breakpoint is looked up.
7. If the breakpoint is not already implemented for this cluster, it is implemented now, i.e., the code is changed in the target.
8. The implementation takes place using operating system primitives.

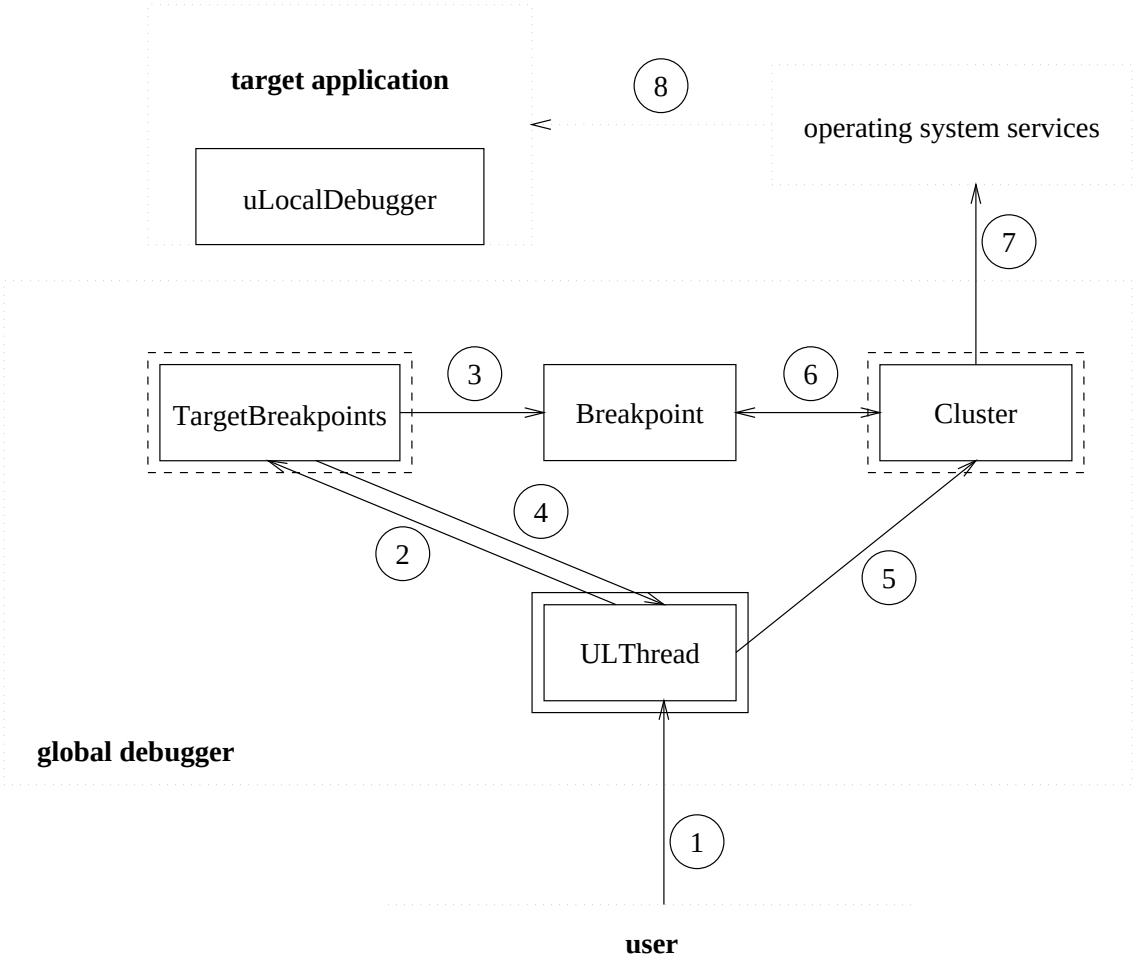


Figure 7.2: Setting a Breakpoint

7.2.3 Encountering a Breakpoint/Continuing the Target

Figure 7.3 shows the steps that are executed when a user-level thread in the application encounters an applicable breakpoint, i.e., a breakpoint that was set for this thread. The thread calls into the local debugger, which sends a notification to the global debugger and blocks the thread until the local debugger receives a continue request. Steps 9–14 show the steps that are performed to continue a user-level thread in the target application. The continue request is received by the `uLocalDebuggerTask` object in the local debugger, which subsequently schedules the continued thread for normal execution.

1. The thread calls the breakpoint handler in the local debugger.

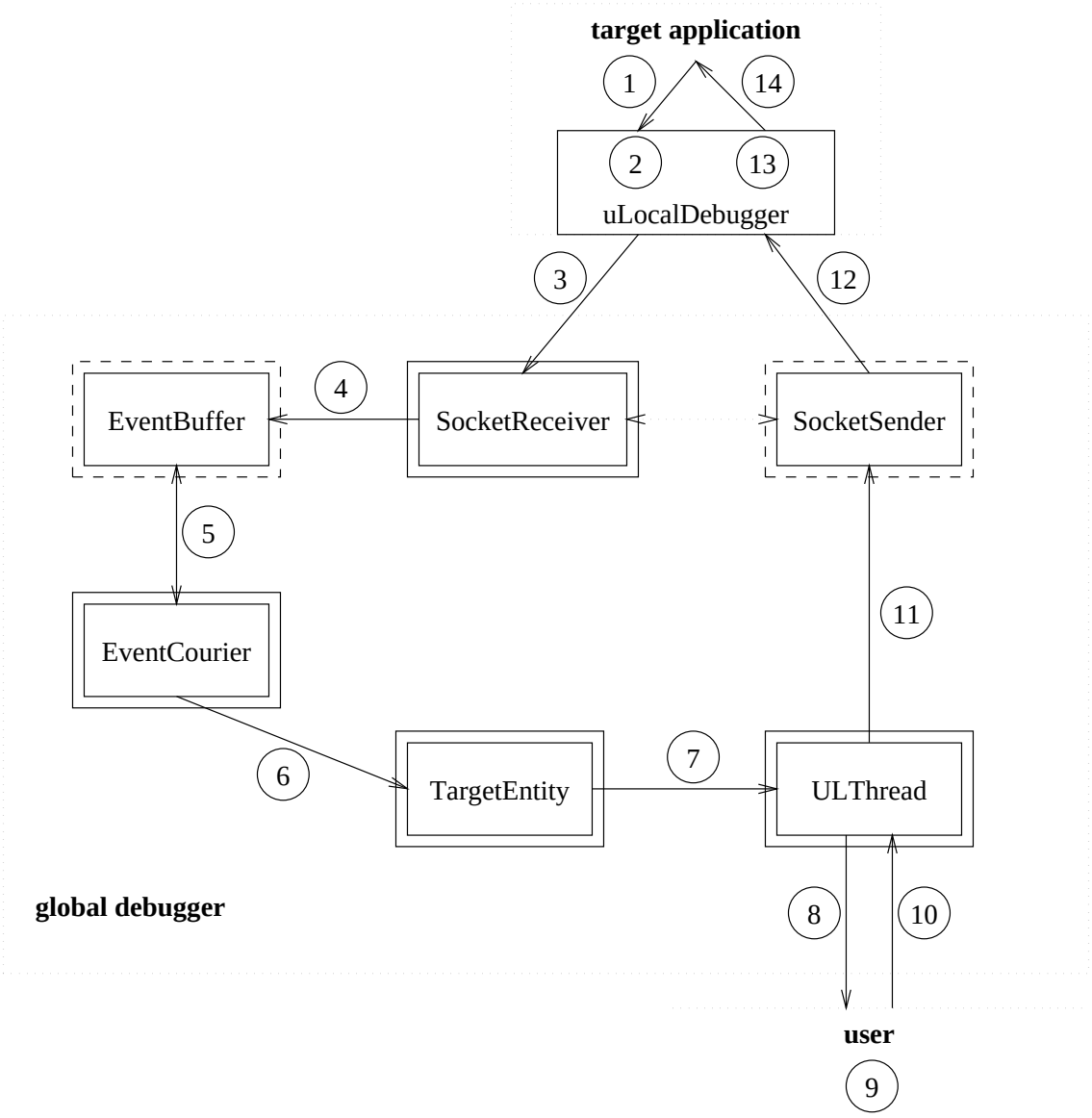


Figure 7.3: Encountering a Breakpoint

2. The breakpoint handler determines whether the breakpoint is applicable to this thread.

3. The local debugger sends a notification about hitting the breakpoint. This notification message also contains the current register set, including the current address of the stack pointer.

4. The appropriate `SocketReceiver` task reads the message from the socket and enqueues it into `EventBuffer`.
5. The `EventCourier` takes the event from `EventBuffer`.
6. The event is delivered to `TargetEntity`.
7. `TargetEntity` notifes `ULThread`.
8. The state of the user interface is changed to indicate that this task encountered a breakpoint. Additional information is provided to the user about the location of the breakpoint at the source code level.
9. The user continues the target thread.
10. The request to continue the thread is delivered from the user interface.
11. A message is constructed and sent to the local debugger using the `SocketSender` object. Among other information, this message contains the information about which breakpoints are applicable for this thread in a bitmask.
12. The corresponding message is transmitted to the local debugger.
13. The local debugger applies the new bitmask to the thread and resumes its execution.
14. The thread continues execution.

7.2.4 Deadlock Prevention

Events that are sent from the local debugger, as well as events that are created due to user interaction, are not synchronized. Since all tasks in the global debugger are mutex objects, i.e., member functions can only be called when a task explicitly accepts them, this can easily lead to a deadlock if two tasks try to invoke methods of each other. It can be seen from Figure 7.3 that usually there is no synchronization needed between sending to and receiving messages from the local debugger. This, together with queueing events in a buffer, prevents deadlocks and allows certain operations, like `printBacktrace` on an `ULThread` object, to be invoked safely even when the corresponding target thread is executing.

The only exception is during an atomic operation, when the `SocketReceiver` directly invokes a method of `SocketSender` to indicate that the local debugger has confirmed

an atomic operation, so messages for the atomic operation can now be sent to the local debugger. However, this is also deadlock-safe, because once an atomic operation is started, `SocketSender` only accepts this confirmation method before accepting or sending any other messages. The dotted line in Figure 7.3 shows this synchronization.

7.2.5 Migration of Kernel and User-Level Threads

Kernel threads as well as user-level threads are allowed to migrate from one cluster to another. If this happens, the target application synchronizes with the global debugger. The thread of control that executes the migration in the target is blocked until the global debugger has performed necessary updates. Besides updating of internal data structures, the debugger has to ensure that all breakpoint implementations are changed appropriately.

If a kernel thread migrates from one cluster to another, all breakpoints that are set for user-level threads on the old cluster are removed from the kernel thread and the breakpoints that are set for user-level threads on the new cluster are implemented.

If a user-level thread migrates from one cluster to another, its breakpoints are reset at the old cluster and set at the new cluster. Setting and resetting takes place using the usual methods that are offered by the `Cluster` monitor, which ensures that code changes are done if and only if they are appropriate.

7.3 Interaction with the X Window System

Preliminary work was necessary to create a modified X Window System package for μC++. An overview of this package is given in Appendix B. A detailed description can be found in [5].

Since `ULThread`, `ThreadGroup` and `DebuggerMain` are task objects, i.e., they each have their own thread of control, it would be appropriate to let them independently communicate with the X server. Unfortunately, the only mechanism offered by the X library to achieve this includes the creation of a separate socket connection to the X server for each object. Due to the large number of `ThreadInterface` objects that may be created during a debugging session, this can easily become inefficient and resource-wasting. Therefore, only one socket connection is created to the X server and a dedicated object with its own thread of control, provided by a modified X Window System package for μC++ is used to receive events from the X server. This object is of type `uXEventLoopTask`.

The X Toolkit Intrinsics library deals with user requests by registering callbacks for the different kinds of events that occur due to user interaction. To catch user generated events (buttons clicks, mouse movement, etc.), methods of the interface objects are registered as callbacks. Incoming events from the X server are dispatched by the `uXEventLoopTask` object to the appropriate user interface object using the X Toolkit Intrinsics callback mechanism. Then, the events are directed to the different tasks, which subsequently execute the requests using their own thread of control.

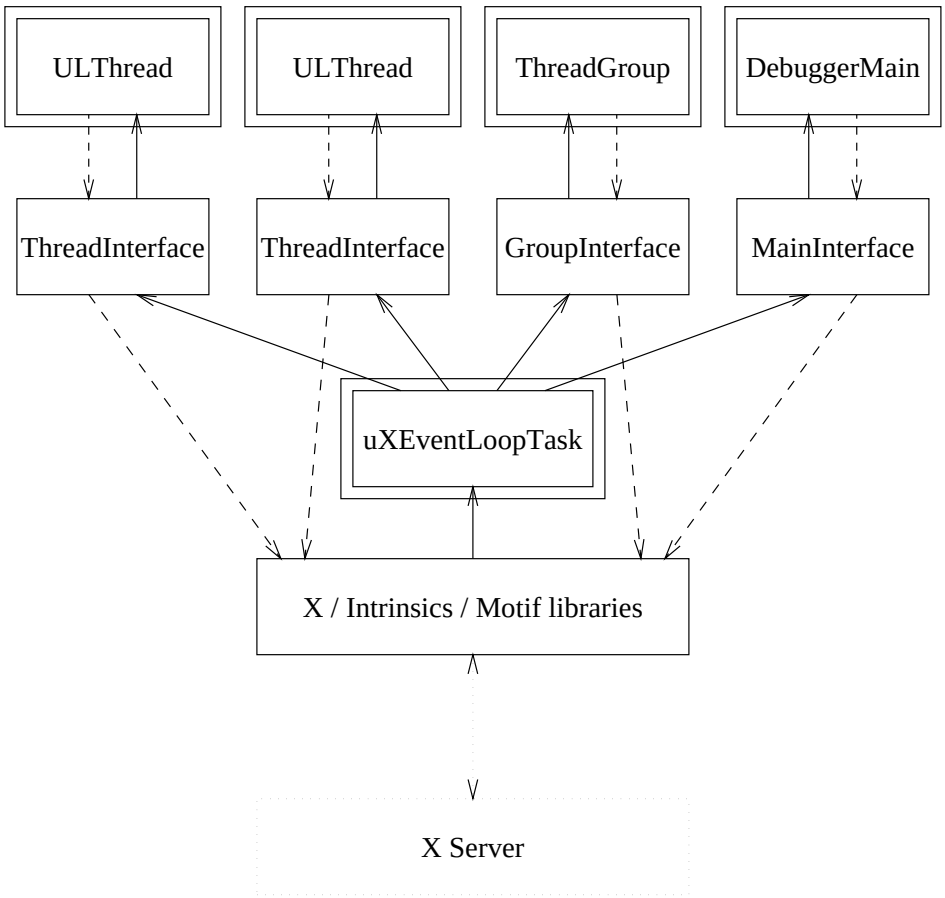


Figure 7.4: Interaction with X Window System

Figure 7.4 shows the interaction of the `uXEventLoopTask` with the corresponding objects in the debugger. The interface objects are created by the corresponding debugger tasks. They have methods that are registered as callbacks and those methods in turn invoke the appropriate methods of the debugger tasks. In the figure, this is shown by solid lines.

Whenever the debugger sends requests to the X server, for example when windows are created or changed, the appropriate functions of the X libraries are called through the interface objects and executed using the thread of control of the caller. This is shown in Figure 7.4 by dashed lines.

7.4 Interaction with GDB Code

The cooperation with GDB code works as follows. A library, *libgdb*, is constructed that contains the used GDB code. Access to this library is covered by the `AccessGDB` monitor, mentioned in Section 7.1.1. GDB's design provides the possibility to register a data structure, `target_ops`, containing callbacks for a variety of actions, for example reading a target's registers. Usually, this mechanism is used within GDB to dynamically switch between different types of target programs, for example an executable file and a core file. The same mechanism is used to let *libgdb* cooperate with the rest of the debugger. Callbacks are registered for reading registers, reading code and data, and for printing output.

To identify a specific thread, a request to `AccessGDB` is accompanied by the register set for the thread and pointer to the `Cluster` object, corresponding to the cluster the thread is currently executing in. Then, the callback functions use this data to serve requests from *libgdb*. Figure 7.5 shows the interaction.

1. A request is made to `AccessGDB`. The register set and the pointer to the `Cluster` is stored in `AccessGDB`.
2. The request is given to `libgdb`.
3. A callback is invoked, for example to read data from the target application.
4. The callback function uses the previously stored information to direct the request to the appropriate `Cluster` object and provides the requested data.
5. The `Cluster` object in turn uses the discussed mechanisms to look up the data in the target application.
6. Results are delivered back to `AccessGDB`.
7. `AccessGDB` transforms the results from GDB specific data structures into general debugger data structures and delivers them to the requesting object.

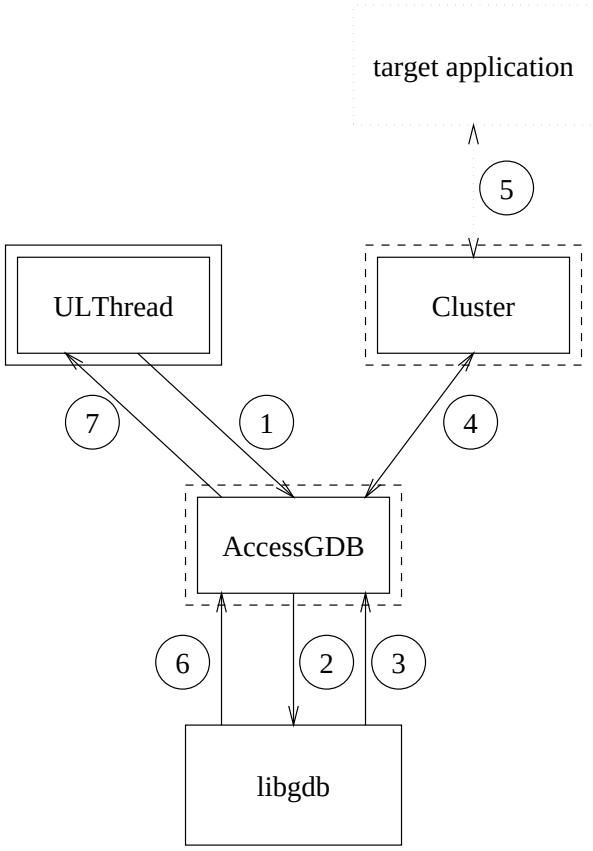


Figure 7.5: Access to GDB Code

GDB code reconstructs a stack image from the given registers using the data lookup through the `Cluster` object. Using this image, it can print a stack backtrace or look up the contents of local variables, again by using the data lookup callback.

Chapter 8

Algorithms

In Chapter 6, the concept of having breakpoint handlers in the application's code is presented and restrictions are discussed. This chapter presents the concrete algorithms implemented in the prototype to deal with this complexity.

8.1 Setting/Resetting a Breakpoint

Setting a breakpoint for a user-level thread is split up in two parts. First, a call to the local breakpoint handler is inserted in the application and the original instructions are transferred into this handler. This step is done by the global debugger, using the `ptrace` or `/proc` interface and happens once per breakpoint per cluster, even if a breakpoint is specified for multiple threads. Second, the thread's breakpoint bitmask is changed to reflect that the breakpoint is valid for the thread. Usually, this changes happens in the corresponding `ULThread` object in the global debugger and the bitmask is transferred to the local debugger when the continue request for this thread is sent. The local debugger installs the bitmask in the runtime system's data structure for the thread. One exception exists and is explained in Section 8.2. Inserting of a breakpoint in the code image is split into two steps, creation of the temporary instructions and the code change. Both are discussed in the following two sections.

Similar actions are performed when a breakpoint is removed. The bit in the task's bitmask is turned off and the code is changed back when the breakpoint is no longer valid for any thread belonging to a cluster. The bitmask is transferred when the thread is eventually continued.

8.1.1 Creating the Temporary Instructions

This section describes how the temporary instructions are generated for the SPARC processor [12]. This mechanism is implemented in the class `Breakpoint_sparc` as described in Section 7.1.2.

General Case

When an object of type `Breakpoint_sparc` is created, it is given the address of the breakpoint, the address of the breakpoint handler, as well as the cluster in which it is created. The cluster is used to look up the original instructions at the breakpoint location. Then they are checked against a table of instructions that need special treatment. In principle, the instruction at the breakpoint address is replaced with a `call` instruction that points to the appropriate breakpoint handler and the next instruction is replaced with a `nop` instruction.

A problem can occur if a `branch` points to the address right after the breakpoint address. If the branch is taken while the breakpoint is implemented, it does not transfer to the breakpoint handler, but executes the `nop` instruction instead of the original instruction. To prevent this from happening, the code of the current function is checked for branches to this address, before the temporary instructions are generated in the local debugger. This test is reasonably efficient, since the code can be looked up in the target's executable file, and even for a large function, the test is a simple a loop containing a compare statement and this check happens in response to user interaction. If a branch is detected, the address of the breakpoint is adjusted by +1. This is sufficient, because usually a compiler does not produce assembler code, with a `branch` instruction spanning multiple functions.

At some instructions, a breakpoint cannot be set. Instead, it is adjusted to a location either at the next or the previous address. These cases are listed in Table 8.1. While adjusting the breakpoint address is slightly inaccurate from the assembler point of view, experience shows that it does not cause major problems in real applications, since none of the listed cases usually exists on statement boundaries of a high-level programming language.

Figure 8.1 shows the code in the breakpoint handler in the general case. The `restore` instruction restores the local state of the application, which lets the return address appear in register `o7`, instead of the usual `i7`. The original instructions are executed and since the return address is in register `o7`, a `retl` instruction is used to jump back into the application.

at `save`	adjustment by +1
before `ret`	adjustment by +1
after `ret`	adjustment by −1
at `restore`	adjustment by −1
before `call`	adjustment by +1
after `call`	adjustment by −1
before `branch`	adjustment by +1
after `branch`	adjustment by −1

Table 8.1: Adjustments for Breakpoints on SPARC

```
restore                                  ! redo save instruction state
original instruction 1
original instruction 2
retl                                     ! jumps back into application
nop
```

Figure 8.1: Temporary Code (general case)

`call` Instruction

If a breakpoint is set at a `call` instruction, the argument of the `call` has to be adjusted when the instruction is moved to the temporary location, since the target address for the `call` is calculated relative to the program counter register. The temporary code is shown in Figure 8.2.

```
call target address
restore                                  ! delay slot, executed before call
retl                                     ! jumps back into application
nop
```

Figure 8.2: Temporary Code (`call` instruction)

The `restore` instruction is placed in the delay slot of the `call`, otherwise the `call` itself would overwrite the register `o7` that is used by the `retl` instruction to jump back

into the application.

The instruction in the delay slot of a call is **not** replaced by a `nop`, but remains at its location, so that it is executed before the breakpoint call takes place. This guarantees the correct relative execution order of the original instructions.

`branch` Instruction

If a breakpoint is set at a `branch` instruction, the argument has to be adjusted, since, as with a `call`, the target address is calculated relative to the program counter register. The temporary code is shown in Figure 8.3.

```
restore                                  ! redo save instruction
branch if condition to line 6
original instruction 2
retl                                     ! jumps back into application after branch
nop
call target address
nop
```

Figure 8.3: Temporary Code (`branch` instruction)

The same type of branch (conditional, annulment) as in the application is implemented in the breakpoint handler. If the branch is taken, the `call` in line 6 transfers control to the original target address of the branch. If the branch is not taken, the `retl` instruction in line 4 jumps back in the application after the branch. The instruction in the delay slot of the branch is treated the same way with regards to annulment as it would be treated at the original location.

8.1.2 Implementing/Removing a Breakpoint

The mechanism described in this section detects the race condition discussed in Section 6.2.3 and is applied when a breakpoint is implemented as well as when a breakpoint is removed.

Two steps are necessary to implement or remove a breakpoint. A check must be made if any user-level thread is executing the code that is to be changed before the code

```
SocketSender* socket_sender;

bool Cluster::implementBreakpoint( int bp_number ) {
    bool result;
    Acquire();                                          // nobody shall use the cluster
    socket_sender->startAtomicOperation();
    stopKernelThreads();
    // checkBreakpoint has to restart the single kernel thread in uni-processor to perform
    // the local check, but preemption is turned off in the target during an atomic operation.
    if ( result = cluster->checkBreakpoint( bp_number, cluster ) {                    10
        ReallyImplementBreakpoint( bp_number );
    } // if
    contKernelThreads();
    socket_sender->finishAtomicOperation();
    Release();                                          // release the cluster
    return result;
} // implementBreakpoint
```

Figure 8.4: Implementing a Breakpoint

change itself happens. Both operations have to be executed atomically, i.e., it must be ensured that none of the affected threads makes any execution progress between check and implementation. Another source of complexity is the fact that μC++ programs can be run in a single UNIX process, simulating the behaviour of the multi-processor environment (see Appendix A). Pseudo code for implementing a breakpoint can be seen in Figure 8.4.

After the thread of control (which belongs to the corresponding `ULThread` object) that executes the breakpoint insertion in the debugger has acquired access to the `Cluster` object, a message is sent to the local debugger, indicating that an *atomic operation* is started. The local debugger turns off preemption for its UNIX process and is subsequently the only executing thread in this process. Then it sends a confirmation back to the global debugger that the atomic operation can start.

The function `checkBreakpoint` performs the check for the race condition in two steps. First, all user-level threads that are currently executed by kernel threads for this cluster are looked up and their execution location is determined using the services of the `KernelThread` class. The location is checked against the address where the breakpoint is implemented. If this check is successful, a message is sent to the local debugger that contains the range of code to be changed. The local debugger uses a list in the runtime system to check for all user-level threads that are not currently executed by kernel threads for

the point where the last context switch occurred. This address is saved on every context switch in a dedicated field of the data structure for the thread. Then, the local debugger sends a response which indicates whether the check was successful. Finally, the atomic operation is finished by notifying the local debugger, which in turn restarts preemptive scheduling for its UNIX process. The result of the check is returned to the caller, and if necessary, the caller retries the operation after a random delay.

8.2 Stopping a Thread

Two phases are necessary to stop a user-level thread. First, the current location is looked up to find out where the thread is executing. Second, a breakpoint is inserted at the next valid location and the bitmask of the thread is updated remotely, using the services of the `KernelThread` class (inserting a breakpoint is discussed in Section 8.1, restrictions in Section 6.2.1). The remote update of the bitmask is an exception, since usually the bitmask is transferred when a stopped thread is continued. Figure 8.5 shows both phases in pseudo code.

Both steps have to occur atomically, i.e., the thread must not make any execution progress during the operations. Therefore, inserting a breakpoint to stop a thread is an atomic operation. After the atomic operation is started and all affected kernel threads (i.e., those belonging to the cluster) are stopped, the current location of the user-level thread is looked up. If the thread is executing in the runtime system, the next appropriate location is determined by walking through the calling stack and finding the closest location where a breakpoint can safely be inserted.

Then in the second step, the breakpoint is implemented and the breakpoint mask is updated remotely, since there is no continue request, which could be used to transfer the mask. Finally, the atomic operation is finished, and the application continues. As soon as the user-level thread is scheduled for execution, it eventually hits the breakpoint and is stopped.

The stop operation already starts and finishes an atomic operation, so the atomic operation within `implementBreakpoint` must not have any effect. This is the reason, why atomic operations can be nested (see Section 7.1.5).

```
Cluster* cluster;
SocketSender* socket_sender;
TargetBreakpoints* target_breakpoints;

StackBacktrace ULThread::getStackOfRunning() {
    RegisterSet regSet = getRegisterSetOfULThread();          // eventually uses KernelThread
    return createStack( regSet );                             // uses GDB code to build stack
} // getStackOfRunning

int ULThread::createStopBreakpoint() {                                              10
    StackBacktrace stack = getStackOfRunning();
    for ( int i = 0;; i += 1 ) {
        // indicate that no stopping is possible (which should not happen)
        if ( i >= stack->getNoOfFrames() ) return -1;
        // check, if the address belongs to the runtime system
        if ( checkBreakpointAddress( stack->getFrame(i).getPC() ) ) {
            return target_breakpoints->createBreakpoint( stack->getPC() );
        } // if
    } // for
} // createStopBreakpoint                                                          20

bool ULThread::stopULThread() {
    bool result;
    cluster->Acquire();                                       // nobody else shall access the cluster
    socket_sender->startAtomicOperation();                    // waits for confirmation from local debugger
    cluster->stopKernelThreads();
    bp_no = createStopBreakpoint();                           // determines the next address in application code
    if ( bp_no < 0 ) {
        return false;
    } // if                                                                        30
    if ( result = implementBreakpoint( bp_no ) ) {
        refreshBreakpointMask();                              // remote refresh using services of Cluster class
    } // if
    cluster->contKernelThreads();
    socket_sender->finishAtomicOperation();                   // done with this...
    cluster->Release();                                       // release cluster
    return result;
} // stopULThread
```

Figure 8.5: Stopping a Thread

8.3 Single Stepping

Single stepping on a per user-level thread basis is achieved by inserting temporary breakpoints in the target application, continuing the target and removing the breakpoints afterwards. For a single step, multiple breakpoints may actually be inserted and all are removed afterwards. There are two operations for single stepping: *next* does not suspend execution in subroutines, whereas *step* does.

In principle, single stepping operates only at the source-line level. At first, the assembler code for the source line is looked up. A breakpoint is set at the beginning of the next line of source code. Then, a check is done for three exceptional cases and additional breakpoints may be implemented whenever such a case exists. The exceptional cases are:

- If a `call` instruction is detected in the line and the operation is *next*, a breakpoint is set at the target address of the call.
- If a `return` instruction is detected in the line, a breakpoint is set in the caller's function after the call.
- If a `branch` instruction is detected and the target address is beyond the scope of the current line, a breakpoint is set at the target address.

After the breakpoints are inserted, the thread is continued automatically until it hits a breakpoint. As with a normal breakpoint, the thread is stopped in the local debugger and the global debugger is notified. Subsequently, all temporary breakpoints are removed.

During a *next* operation, it is possible that a function is called and one of the normal breakpoints is hit in the function. In this case, the temporary breakpoints are not removed automatically, since the user may want to continue immediately and finish the *next* operation. Therefore, the breakpoints for the single step remain in the target until one of them is hit. On the other hand, if the user, instead of continuing at the normal breakpoint, issues another single step command, the breakpoints set for the previous single step are removed, before the new temporary breakpoints are inserted.

Another possible conflict occurs when any of the temporary breakpoints is generated for an address where a normal breakpoint is already inserted for the same user-level thread. In this case, the address is ignored during insertion and removal of the temporary breakpoints, i.e., the normal breakpoint remains set after the single step operation is done.

8.4 Target Abort

A variety of fatal errors usually causes an application to abort, for example a segmentation violation. A traditional debugger that synchonously waits for events from the target, using the UNIX primitives, is automatically notified about such an error. Subsequently, the user can examine the target's state to reason about the cause of the fatal error.

This mechanism does not work for the presented design, because the debugger never uses the UNIX primitives to wait for events from the target. Instead, when a fatal signal is delivered to one of the target's UNIX processes, the runtime system calls into the local debugger and the global debugger is notified. Subsequently, the user can examine the state of any thread that currently exists in the application and afterwards the user confirms the abort. This releases the target, so it can shut down and dump its core image.

Chapter 9

User's Guide

This chapter describes *kdb*, the prototype implementation that can be used to debug μC++ [6] applications. μC++ applications can be compiled either in a uni- or a multi-processor mode, which are both supported by the debugger. The user interface is based on the *X Window System* and the *Motif* widget set. This chapter assumes an understanding of the runtime entities of μC++ (see Appendix A for a brief description).

All windows in the user interface can be resized arbitrarily from the window manager's border and the size of their window components is adapted automatically. Additionally, the main window and the task window are divided into several panes, whose relative height can be changed by using Motif's *PanedWindow* mechanism.

9.1 Starting a Debug Session

The debugger operates as a server at which the target application registers as a client. When a μC++ application is started, the μC++ runtime system starts the local debugger, which is linked to the application when the compilation flag `-debug` is specified.

The local debugger checks for two environment variables, which denote the IP address of the global debugger's socket. If present, the local debugger connects to the global debugger, which in turn takes control of the target application. A UNIX shell script can be used to start debugger and target. It is invoked by calling:

`kdb` *targetname [target arguments]*

The debugger's main window (see Figure 9.1) appears and the target application is started and immediately stopped at the beginning of `uMain::main`.

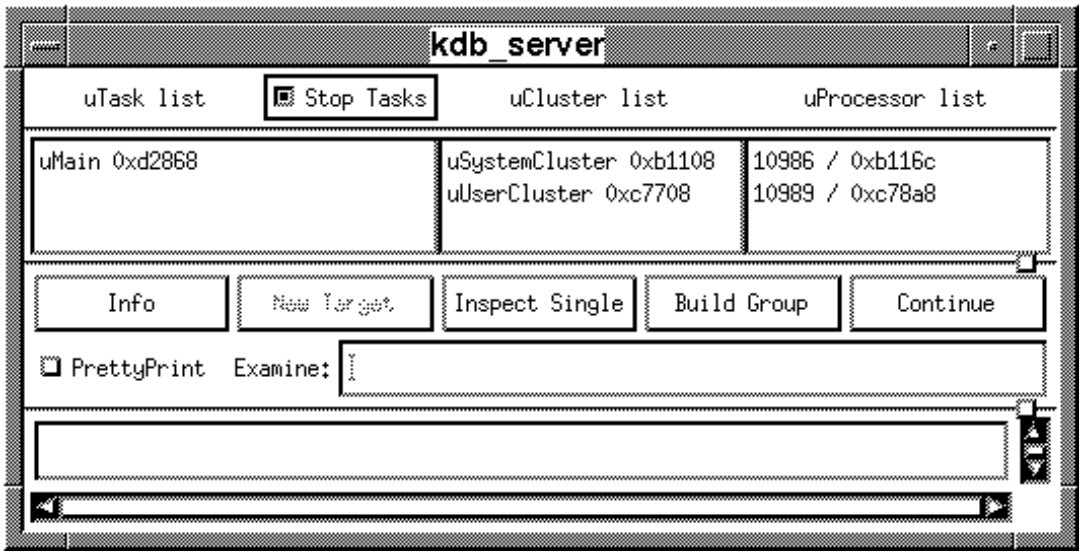


Figure 9.1: Main Window after Startup of Debugger and Target

9.2 Ending a Debug Session

To terminate the debugger, the main window can be closed at any time by using the appropriate mechanisms of the window manager to close the main window. If the target is still executing at this time, the debugger releases control and the target continues normally.

If the target finishes execution, the debugger does not have to be restarted to proceed with a new debugging session of the same or a new target program. When the target is gone, the debugger's main window enters the state shown in Figure 9.2. The only sensitive button is *New Target*, which can be clicked on to prepare the debugger for a new target program. Subsequently, a new target can be invoked using another shell script:

`kdb_target` *targetname [target arguments]*

9.3 Target Abort

If a fatal error occurs in the target application, for example a segmentation violation, a window is popped up to indicate this to the user. Aftwards, the state of the target can be examined using the normal mechanisms. When the *OK* button is clicked in the popup window, the target finishes and dumps a core file.

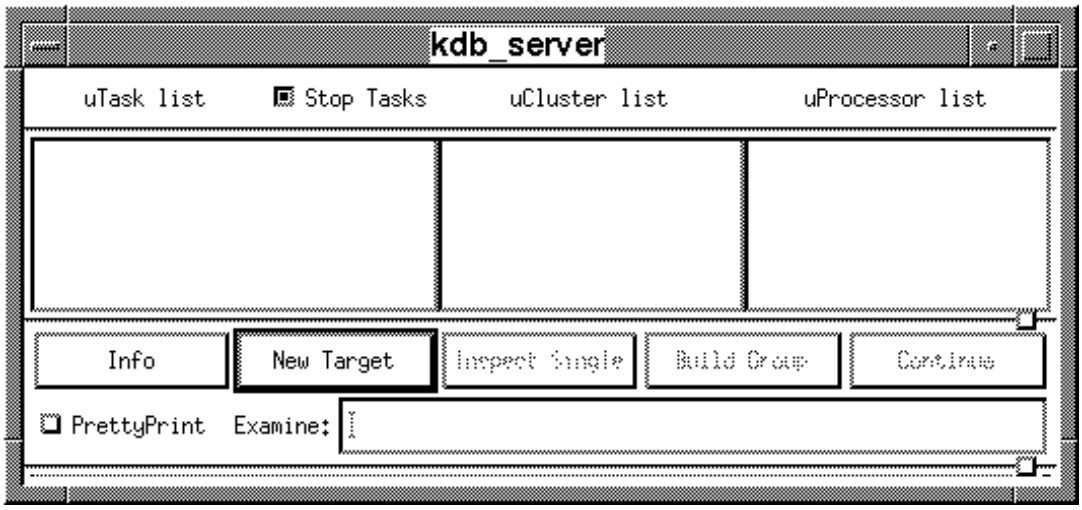


Figure 9.2: Main Window after End of Target

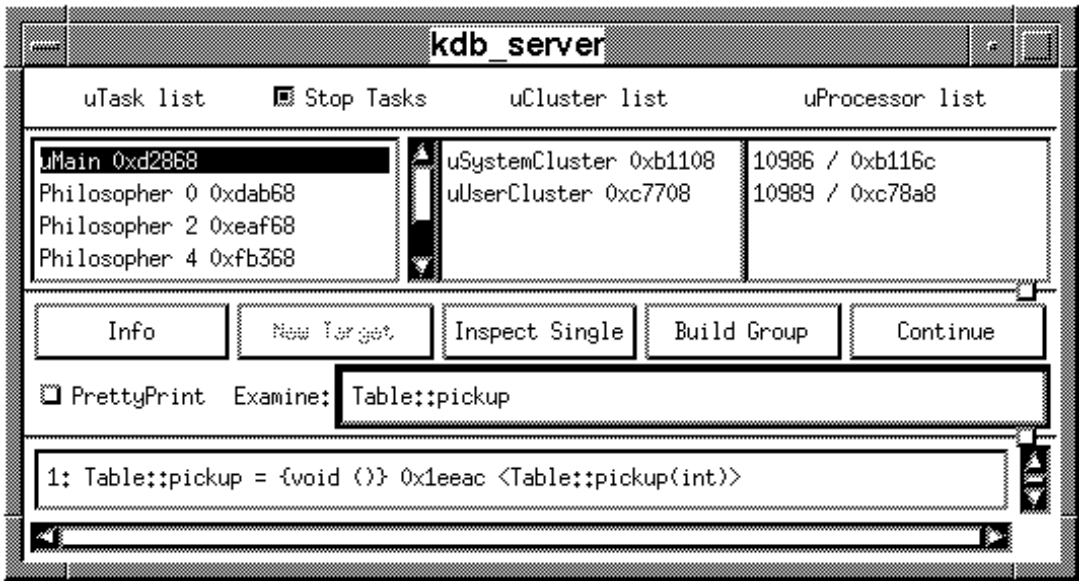


Figure 9.3: Main Window showing Symbol Lookup

9.4 Main Window

This section discusses the debugger's functionality and how to access it through the main window (see Figure 9.3).

The main window has 3 panes (number 1–3 from top to bottom). Their relative height can be adjusted by using the adjustment box on the horizontal separator lines between the panes.

- Pane 1 contains the list of tasks, clusters and processors that currently exist in the application.
- Pane 2 contains a row of debugger control buttons, a text field where symbol names can be typed in and a toggle button to structure the output.
- Pane 3 contains a text field that is used to show output.

The *Info* button in Pane 2 pops up a short copyright note (and may provide help information in the future).

uTask list

Tasks are identified by their name and memory address in the target application. While μC++ allows the name of a task to change dynamically, the debugger only uses the name given when a task is created. Normally, a newly created task is stopped at the beginning of its `main` member routine and remains stopped until it is continued by the user. In certain situations, the user might not be interested in debugging new tasks. Therefore, if the toggle button *Stop Task* is turned off, subsequently created tasks continue execution immediately, after they are registered at the global debugger.

One or multiple tasks can be selected from the task list for further examination by clicking on the task's entry. Clicking on the *Inspect Single* button causes a dedicated task window to pop up for every selected task. Double-clicking on a single entry of the task list does the same as selecting a task and clicking *Inspect Single*. Clicking on *Build Group* forms a group of all selected tasks and pops up one window, where commands can be issued on all tasks that belong to the group. Finally, the *Continue* button is a convenience function, that continues execution of all selected tasks. If some of the tasks are already running, the continue request is ignored for these tasks.

uCluster list

There is one operation available for clusters. By double-clicking on an entry in the cluster list, a window (see Figure 9.4) pops up to control whether the global debugger is notified

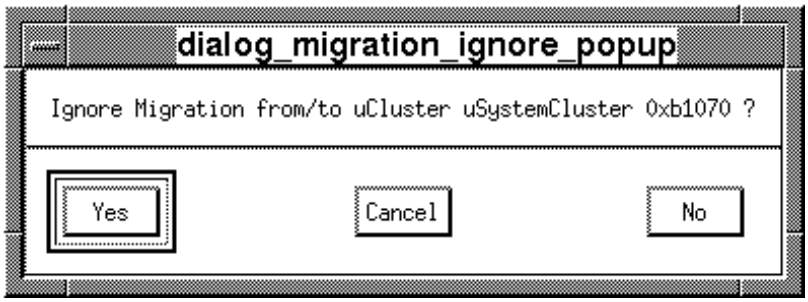


Figure 9.4: Dialog about Further Migration (uCluster)

when tasks migrate to and from this cluster. Selecting *Yes* means the global debugger ignores further task migration and *No* means the global debugger is notified about further task migration.

As discussed earlier, every migration of a task involves overhead, since it has to be reported to the global debugger and code in the target's UNIX processes may have to be changed. Hence, with a good understanding of the application and the underlying UNIX and μC++ principles, there can be situations where ignoring of migration improves performance. However, this mechanism should be used only by experienced users to prevent problems, since it is not coordinated with the rest of the debugger's operations. The default settings are carefully chosen to provide the best efficiency in most situations.

uProcessor list

The only operation available for processors is a similar mechanism to ignore migration of processors among clusters by the global debugger. Again, this is an insecure optimization mechanism that should just be used by experienced μC++ software developers.

Global Symbol Lookup

In the text field in Pane 2, right beside the label *Examine*, global symbol names can be specified, and on pressing `Return` in the text field the symbol's contents are displayed in the text field in Pane 3 (see Figure 9.3). If *Pretty Print* is turned on, the contents of complex data types are shown in a structured way, which usually results in longer output.

9.5 Task Window

This section discusses the debugger's functionality for examining and controlling a specific task through a dedicated task window (see Figure 9.5).

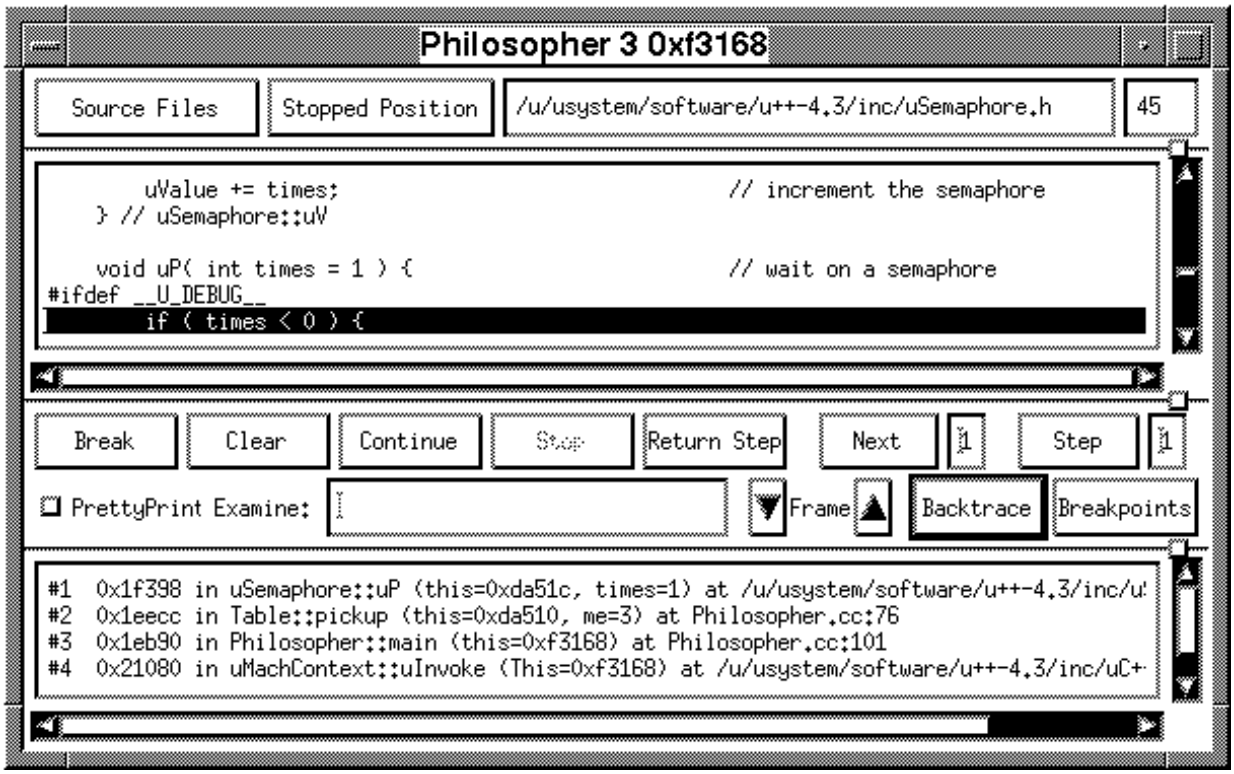


Figure 9.5: Task Window showing Backtrace

A task window has 4 panes (number 1–4 from top to bottom) Their relative height can be adjusted by using the adjustment box on the horizontal separator lines between the panes.

- Pane 1 contains two buttons to select which source code is currently shown in Pane 2, as well as the filename and the line number of the currently highlighted line in Pane 2.
- Pane 2 contains a text field that shows a source code file.
- Pane 3 contains various control buttons and input fields.
- Pane 4 contains a text field that is used to show output.

Every task that is selected for examination is controlled independently by a dedicated window. When a task window pops up, it shows the last know position of the corresponding task by highlighting that line in Pane 2. The corresponding source file name and line number appear in Pane 1. If the task is executing when the task window is created, a snapshot of the current execution stack is taken and used to determine the position of the highlighted line. The last known position of a running task can be refreshed by selecting the task again from the main window or by double-clicking on the *Backtrace* button. Clicking the button *Stopped Position* always switches Pane 2 back to the last known execution position. The highlighting bar in Pane 2 can be moved using the cursor keys or by clicking on a line of source code.

The *Backtrace* button produces a backtrace of the calling stack for the task and shows it in Pane 4. This can be seen in Figure 9.5. The arrow buttons to the left and right of the label *Frame* in Pane 3 can be used to step up and down the calling stack, which also causes the text field in Pane 2 to switch to the corresponding source file and line number. All symbol names are resolved relative to the selected frame when looking up the contents of variables.

Symbol Lookup

To look up symbol information, the same mechanism is used as in the main window. Beside the *Examine* label in Pane 3, symbol names can be typed in, and on pressing `Return` the appropriate contents are displayed in Pane 4. Names are looked up relative to the current frame, e.g., in Figure 9.6, the name `this` refers to the object on which the class method `pickup` is invoked. There is also a *Pretty Print* option that formats the output in a structured way.

Control Buttons

The control buttons in Pane 3 (see Figure 9.5 or Figure 9.6) are used to control execution of a task. Execution of other tasks in the application is not affected, other than through synchronization that is implemented in the target application. To set a breakpoint, position the highlighting bar at the desired line of source code and click on *Break*. A breakpoint is removed by clicking on *Clear* when the breakpoint's location is highlighted. Clicking the *Continue* button resumes execution of the task.

Clicking *Next* executes a line of source code. If this line contains function calls, the functions are completely executed. This is different from *Step*, which also executes a line

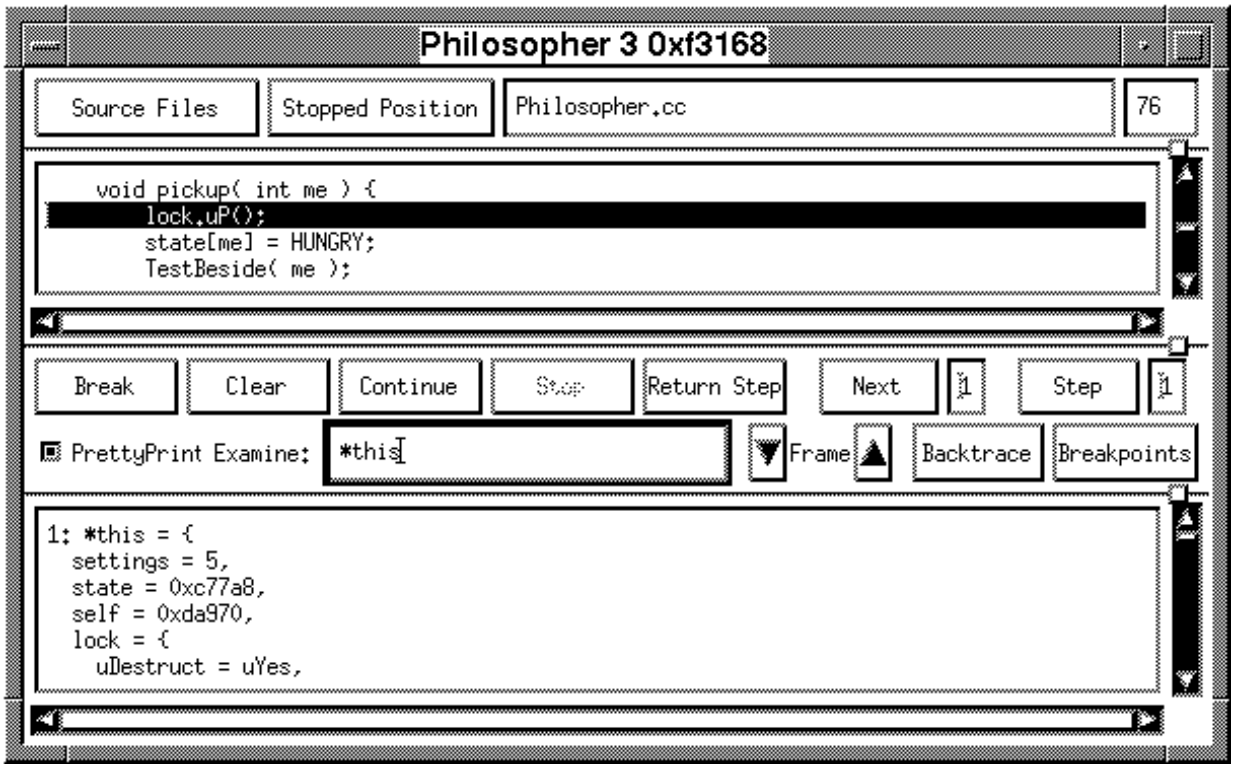


Figure 9.6: Task Window showing Symbol Lookup

of source code, but steps into each function call and stops execution at the beginning of the function. For both *Next* and *Step* button, a number can be specified in the corresponding text field right beside each button to perform multiple operations.

Clicking on *Return Step* causes the task to continue execution until the end of the current function is reached. Execution stops after the caller's function call.

Source File Selection

To examine any of the source files associated with this application and its runtime libraries, click on *Source Files* and a window showing a list of all source files for the current application pops up. In this window, a source file can be selected, which is subsequently examined in the task window. To select a source file, either double-click on its name or click on its name and the *OK* button.

Breakpoint Selection

The *Breakpoints* button pops up a window (see Figure 9.7) that contains a list of all breakpoints that are currently set for this task. A breakpoint is selected by double-clicking on its list entry or typing the breakpoint number in the selection window and the source file containing it appears in Pane 2, positioned at the breakpoint's location. Additionally, breakpoints can be deleted by clicking on a breakpoint's entry in the list and the *Clear* button of the breakpoint window.

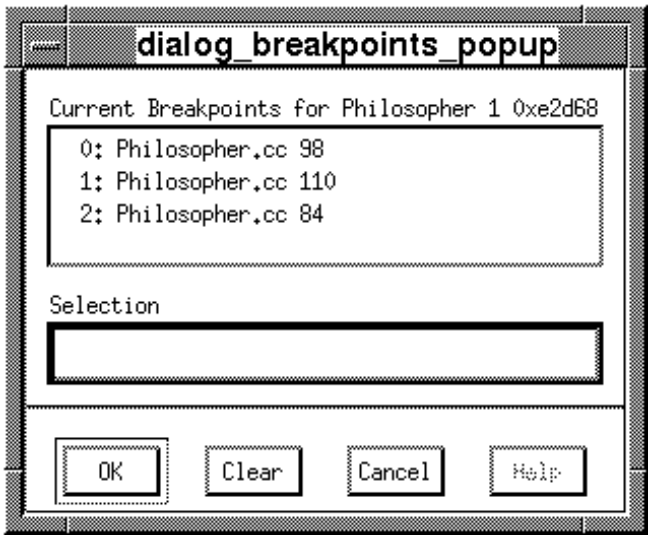


Figure 9.7: Breakpoint List

Examining a Running Task

When a task is running, the task window switches over to the state shown in Figure 9.8. At this point, all control buttons become inactive, whereas the *Stop* button becomes active. Clicking *Stop* stops the task at the next possible location. However, if a task is currently blocked in the application (for example, if a task is waiting on the entry queue of a mutex object), the stop request does not take effect until the task is eventually made active again.

Additionally, it is possible to monitor the current execution of a running task by clicking on the *Backtrace* button. This gives a snapshot of the current execution stack (see Pane 4 in Figure 9.8) and is especially helpful if an application runs into a deadlock.

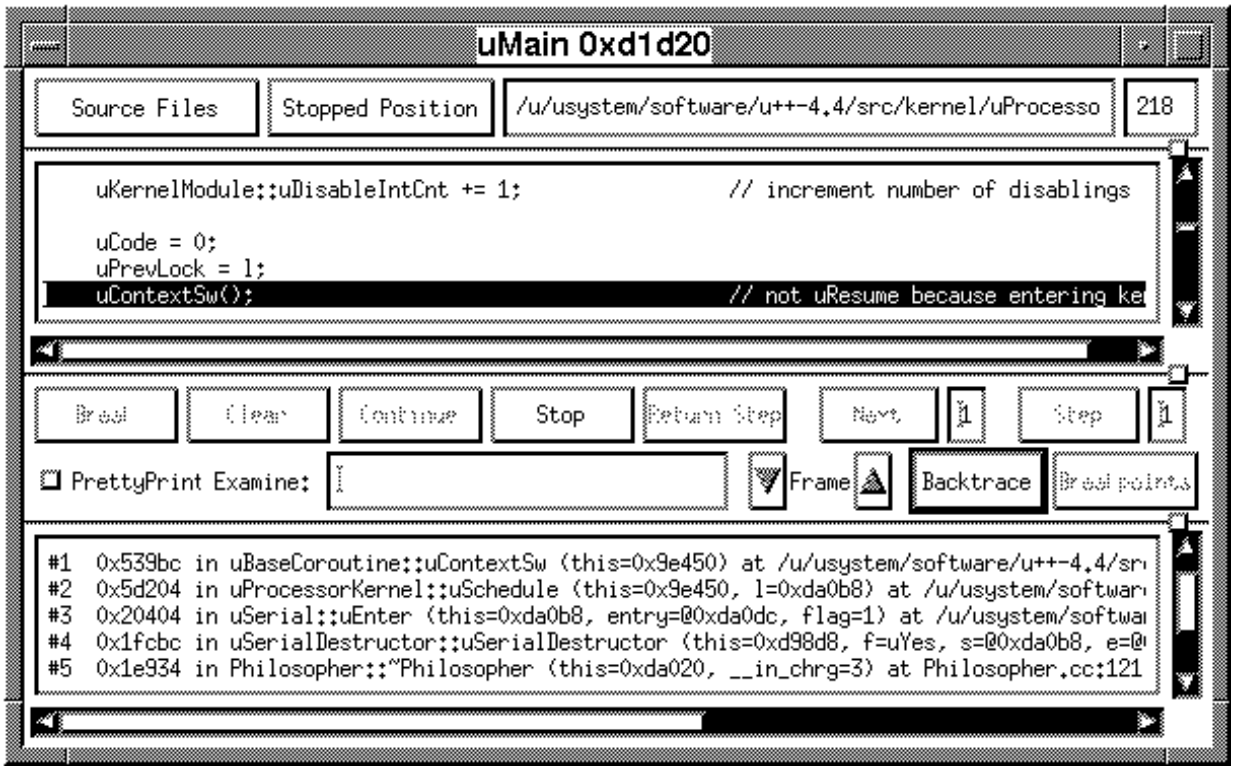


Figure 9.8: Task Window showing Running Task

Initially, the frame arrow buttons and the symbol lookup field is inactive, because looking up variables of a runnning task might display stale information and cause confusion. On double-clicking the *Backtrace* button these buttons are made active and a new snapshot is taken from the task's execution stack. This snapshot is used for subsequent examination of variables. It is possible to walk through this execution stack snapshot using the frame arrow buttons. Variables of a running task can be looked up relative to the selected frame in the same way as for a stopped task. Again, this facility is deemed helpful in case of a deadlock, when the tasks are blocked, so the stack frame and variables are not changing.

It is also possible to view different source files when a task is running, but as soon as the task encounters a breakpoint, Pane 2 shows the current stopped location.

9.6 Group Window

This section describes the functionality of the debugger that is available for a group of tasks through the group window (see Figure 9.9).

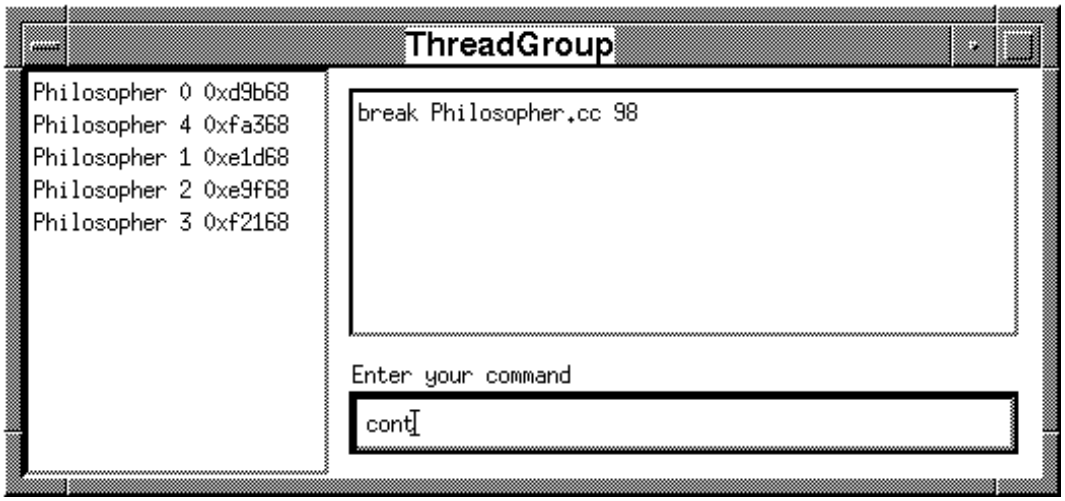


Figure 9.9: Group Window

For the user's convenience, tasks can be grouped together and operations can be issued on a group of tasks. To form a group, multiple tasks are selected from the task list in the main window and a click on the button *Build Group* (see Figure 9.1 or Figure 9.3) pops up the interface for a group of tasks. In this window, the following commands can be entered and the corresponding operations are performed on all members of the group:

- Set a breakpoint:

  `break` *sourcefilename linenumber*
  `break` *functionname*

- Remove a breakpoint:

  `clear` *sourcefilename linenumber*
  `clear` *functionname*

- Stop execution:

  `stop`

- Continue execution:

  `cont`

- Perform a number of single steps with automatic execution of subroutine calls:

  `next` *[number]*

- Perform a number of single steps without automatic execution of subroutine calls:

  `step` *[number]*

In general, if a command is not applicable to one of the tasks, e.g. `stop` for an already stopped task, the command is silently ignored. Grouping tasks together does not affect the ability to control every task separately and reactions to the commands issued in the group windows, such as the encountering of a breakpoint, become visible in each task's window.

Chapter 10

Conclusions and Future Work

The goal of this thesis was to identify the key aspects for the design of a multi-threaded interactive source-level debugger. It should be flexible with regard to different paradigms for multi-threaded programming and communication, and provide independent control over each thread in the target application.

10.1 Summary

It turns out that debugging multi-threaded applications is considerably different from debugging sequential programs. The different forms of parallelism and the limited debugging support given by the UNIX operating system place significant restrictions towards controlling multiple threads that share a single code image.

The presented design overcomes these restrictions with partly distributing the debugger's work into the target application and using fast user-level breakpoint handling. Thus, the debugger fulfills the aspired goals. It is a multi-thread application and allows the user to control each thread independently of other threads.

A prototype is implemented for two UNIX versions, SunOS 4.1 and Solaris 2.3. This prototype shows that another goal for the debugger, portability, is also achieved, since both versions of UNIX differ significantly in two areas that are important for debugging: the format of executable files and the operating system access to another UNIX process. The different formats of executable files is handled by the GDB library that was built. The different access methods to control another UNIX process are handled by having two implementations of class `KernelThread`, which results in approximately 200 lines of code that differ between both versions.

The prototype was used by students for course work, who found it a useful tool to debug their assignment projects. It was also used during development of the debugger itself. It will be used for future course work and will be made publicly available for debugging μC++ applications. The prototype shows itself to be efficient enough to be used in a multi-user academic environment.

10.2 Future Work

Future work should be directed towards 3 areas: portability, functionality and interoperability.

The design of the debugger supports portability for different architectures, but an implementation only exists for two UNIX versions that run on SPARC architectures. Additional ports should be done to different processor types to show the general applicability of the concept of user-level breakpoint handling. As well, the debugger could be extended to support multiple user-level packages and/or to support heterogenous applications to demonstrate this aspect of portability.

Functionality can be extended to fully support multiple address spaces and to debug programs that are spread over different machines. To support this, a debugging stub module would run on each machine, which performs a small number of requests, like code changes, on behalf of the global debugger. A protocol for communication between global debugger and debugging stub module has to be designed. However, most of the debugger's activities are already set up to support remote debugging, since everything that is handled by the local debugger is reported to the global debugger using a `socket` and this mechanism can easily be extended to communicate across machine boundaries.

The prototype implementation was only intended to be a proof of concept that the proposed design for a debugger is realistic and useful. Since it is already reasonable stable and efficient, which was an unexpected bonus, the decision was made to release it for public availability. The user interface should be revised considering feedback from future users. As well, users may request additional commands, different data presentation, etc.

Probably the most challenging future work is the task of integrating high-level analysis approaches, such as the ones surveyed in Chapter 3, with this source-level debugger. This can generate a comfortable debugging environment for the development of multi-threaded applications and increase the overall efficiency and applicability of parallel computing.

Appendix A

Overview of μC++

The μC++[6] project introduces concurrency into the object-oriented language C++[27]. To achieve this goal, a set of programming language abstractions are added to C++ that provide concurrency and synchronization. The synchronization features of μC++ are not discussed here, since they are not important from the standpoint of building the debugger.

μC++ consists of a translator and a runtime library. The translator transforms the additional language constructs into C++ code. After the source code is compiled, the runtime library is linked to the application and provides user-level scheduling for multiple threads of control. The runtime system also provides mechanisms to perform lightweight-blocking I/O operations.

A thread of control can be created by declaring a class-like data type, called `uTask`. Each task data type must contain a member routine `main`, which is invoked concurrently to when an object is created. There is a dedicated task type `uMain` of which one object is created initially. This replaces the usual `main` routine of a C++ program. The application programmer implements the `main` routine of `uMain`, which is invoked during startup and subsequently additional tasks can be created using the described mechanism.

Applications are not restricted to one UNIX process. To make use of multiple CPUs in a shared-memory computer, several processes can be created by creating objects of type `uProcessor` during runtime to achieve parallelism. Tasks and processes are related through objects of type `uCluster`. Each `uCluster` contains a number of tasks and processes, and tasks are automatically scheduled to be executed by the processes associated with that cluster. The scheduling mechanism guarantees load balancing among UNIX processes for each cluster, but there is no load balancing among clusters. Both tasks and processes can migrate from one cluster to another.

The memory model is a single address space that is shared among all tasks in the application. In the multiprocessor case, the address space of all UNIX processes is mapped into this single data address space.

μC++ provides a uni-processor mode, in which the environment of the multi-processor mode is simulated, but the application is restricted to one UNIX process and concurrency is achieved only by user-level context switching. This mode can be used to reduce resource usage on a multi-user system.

There is no inherent restriction to the number of tasks, processes or clusters that exist in an application besides the resource limits of the operating system and machine.

Appendix B

X Window System for μC++

Version 11, release 6 of the X Window System [11] [20] is the first release that is intended to support client applications that work in a multi-threaded environment. While the client libraries are configured to work with the thread libraries of multiple vendors, some additional work was necessary to make them compatible with μC++. A short overview is given here, suitable for a reader who is already familiar with multi-threading in X11/R6. A detailed description on how to create concurrent X applications can be found in [11] and [20], which are both distributed with the X source distribution. Information special to X and μC++ can be found in [5].

As a first step, wrapper functions were written, so that the X libraries, which are programmed in C, can internally use the `uLock` and the `uCondition` classes of μC++. All locks in X are owner locks, i.e., they can be re-acquired by the thread of control that currently holds the lock, but they have to be released exactly as often as they were acquired.

At the *Xlib* level, locking is performed automatically for every connection to the X server, i.e., for every *display* that is opened. When using the *Intrinsics* library, several *application contexts* can be created, each of which has own display connections, and locking occurs on an application context. Additionally, a global lock for the whole application can be used by widget developers, if global data has to be protected.

Since the X libraries use data that is private to each UNIX process, like file descriptors, etc., a dedicated `uIOCluster` (see [6] for a description) is created in an X application and a class `uXwrapper` is provided, which migrates the executing task to this cluster when an object is created, and migrates it back when the object is destroyed. This class can be used for automatic migration in every function that calls X library functions, like in Figure B.1.

```
#include <uXlib.h>

Display *dpy;

void createInterface() {
    uXwrapper dummy;                // automatic migration

    dpy = XOpenDisplay( NULL );

    // ...                                                          10

};                                  // automatic migration back on destruction of dummy
```

Figure B.1: Migration for X Library Calls

In a μC++ application, timer interrupts are used to realize preemptive scheduling. Certain UNIX system calls return a failure value and set the error number when a timer interrupt occurs while the system call is executed. This is partly handled in the Xlib, except for initialization of the socket connection with the X server. To prevent obscure error messages, preemption of μC++ is turned off, whenever a connection is established. Again, this mechanism is planted into the `Xlib` using wrappers from C++ to C.

Unfortunately, the Motif widget library is not thread-safe. Therefore, every call that accesses a Motif widget has to to acquire mutual exclusion across the entire X, Intrinsics and Motif libraries, hence the internal locking mechanisms of the Intrinsics library are largely obsolete. This problem can be seen in the function `changeValue` in Figure B.2. Furthermore, when a callback function is called, the lock for the application context and the global lock are already acquired. If a callback contains a call to a mutex member of a task, this can lead to deadlock situations if the task also tries to acquire the X locks to perform any changes in the interface. To handle this situation, another wrapper class, `uXmCBwrapper` is provided to release the owner locks and re-acquire them when the callback routine completes. This problem can be seen in the function `anyCallback` in Figure B.2. One should be aware that between creation and destruction of a callback wrapper no lock is held and therefore no call can safely be made that accesses Motif data.

```
#include <uC++.h>
#include <uXlib.h>
#include <X11/Intrinsic.h>

void anyCallback();
void changeValue( int );

uTask ULThread {

void main() {                                                          10
    // ...
    changeValue( 15 );
    // ...
    }

public:
    void valueNotification( int );

};
                                                                       20
ULThread* thread;
XtAppContext app;

void anyCallback() {
    int value;
    XtVaGetValue( my_widget, XmNvalue, &value, NULL );
    uXmCBwrapper dummy;    // releases the process lock and the application context lock
    thread->valueNotification( value );
} // dummy's destructor automically reacquires application context lock and process lock
                                                                       30
void changeValue( int x ) {
    XtAppLock( app );
    XtProcessLock();
    XtVaSetValues( my_widget, XmNvalue, x, NULL );
    XtProcessUnlock();
    XtAppUnlock( app );
}
```

Figure B.2: Callback Wrapper for Deadlock Prevention

Appendix C

Speed Tests

The following tests are just rudimentary simulations for the real scenario of a thread checking if a breakpoint applies to it. None of the tests actually check for applicability of a breakpoint, because once the breakpoint is identified, this would be similar in all methods. In a real debugger, the first two tested methods would have to perform an additional test to find out which breakpoint was triggered, but since the results of the tests are impressive enough, this was not implemented. This search is not necessary for fast breakpoints, since a dedicated handler exists for every breakpoint number, therefore, when it is executed, it is implicit which breakpoint is triggered.

The focus for the tests is the time needed to resume execution of a target thread, after the applicability test fails. The measured time is the real time that is used by the target application.

Every program was run 10 times, the number of loops was set to 100,000. The average results are shown in Table 5.1.

C.1 Traditional Breakpoints

C.1.1 Target

After the breakpoint is implemented in `function`, it is triggered n times as specified in the command line.

The following is the source code for the simulated target program:

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <sys/times.h>
#include <limits.h>

void function() {
    asm("nop");
    asm("nop");
}

int main( int argc, char** argv ) {
    int i, loops = 1;
    printf("PID : %d\n",getpid());
    printf("function's nop : %d\n",(long*)function + 1);
    getchar();
    if (argc > 1) {
        loops = atoi( argv[1] );
    }
    clock_t start_time, end_time;
    struct tms buffer;
    start_time = times( &buffer );
    for ( i = 0; i < loops; i += 1 ) {
        function();
    }
    end_time = times( &buffer );
    printf("real time (ticks) : %d\n",end_time - start_time);
    printf("system time (ticks) : %d\n",buffer.tms_stime);
    printf("user time (ticks) : %d\n",buffer.tms_utime);
}
```

C.1.2 Control Program

A breakpoint is implemented using the `/proc` filesystem. Afterwards, the target is continued and the control program waits for events from the target. When the breakpoint is hit, its code is changed back and another breakpoint is implemented at the next assembler instruction. When this breakpoint is hit, it is removed and the original breakpoint is implemented again. As stated previously, the control program would normally have to perform a search through all breakpoint locations to find out which breakpoint was triggered.

The following is the source code for the simulated control program:

```
#include <sys/types.h>
#include <sys/procfs.h>
#include <fcntl.h>
#include <unistd.h>
#include <signal.h>
#include <stdlib.h>
#include <stdio.h>

unsigned long   old_value, address, breakpoint_value = 0x91d02001;
int             pid, fd;                                                    10
sigset_t        sigs;
prrun           cont_pars;
prgregset_t     regs;

void set_break() {
    lseek( fd, address, SEEK_SET );
    read( fd, &old_value, 4 );                  /* reading old value */
    lseek( fd, address, SEEK_SET );
    write( fd, &breakpoint_value, 4 );          /* setting new value */
    ioctl( fd, PIOCRUN, &cont_pars );           /* continue without signal */   20
}

void wait_break() {
    ioctl( fd, PIOCWSTOP, 0 );                  /* wait for stop */
    ioctl( fd, PIOCGREG, &regs );               /* read registers */
    /* usually: figure out which breakpoint was hit */
    lseek( fd, address, SEEK_SET );
    write( fd, &old_value, 4 );                 /* write original value */
}
                                                                            30
int main( int argc, char **argv ) {
    if ( argc < 3 ) {
        fprintf( stderr, "usage : %s pid address\n", argv[0] );
        exit( -1 );
    }

    cont_pars.pr_flags = PRCSIG;                /* used in set_break */
    address = atoi( argv[2] );                  /* used everywhere */

    char filename[128];                                                     40
    sprintf( filename, "/proc/%d", atoi( argv[1] ) );
    fd = open( filename,O_RDWR );               /* attach to process */
```

```
    ioctl( fd, PIOCSTOP, 0 );                    /* stop process */

    printf("go ahead\n");

    premptyset( &sigs );
    praddset( &sigs, SIGTRAP );
    ioctl( fd, PIOCSTRACE, &sigs );          /* register for SIGTRAP */

    while(1) {
        set_break(); wait_break(); address += 4;
        set_break(); wait_break(); address -= 4;
    }
}
```

C.2 Local Trap Handling

C.2.1 Target

The function `sig_handler` simulates a local breakpoint handler. In order to be complete, the replaced instruction would be implemented at a temporary location and the `PC` register would be changed to point at this temporary location. The same search for the number of the triggered breakpoint would also apply for this case, as for traditional breakpoint handling.

The following is the source code for the simulated target program:

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <sys/times.h>
#include <limits.h>
#include <ucontext.h>
#include <signal.h>

void function() {
    asm("nop");
    asm("nop");
}

void sig_handler(int signo, siginfo_t *sfp, void *cxt) {
    ((ucontext_t*)cxt)->uc_mcontext.gregs[REG_PC] += 4;
```

```
}

int main( int argc, char** argv ) {
    int i, loops = 1;
    struct sigaction act;                                                    20
    printf("PID : %d\n",getpid());
    printf("function's nop : %d\n",(long*)function + 1);
    act.sa_flags = 0;
    (void (*)(int signo, siginfo_t *sfp, void *cxt))act.sa_sigaction = sig_handler;
    sigemptyset(&act.sa_mask);
    sigaction( SIGTRAP, &act, NULL );
    getchar();
    if (argc > 1) {
        loops = atoi( argv[1] );
    }                                                                        30
    clock_t start_time, end_time;
    struct tms buffer;
    start_time = times( &buffer );
    for ( i = 0; i < loops; i += 1 ) {
        function();
    }
    end_time = times( &buffer );
    printf("real time (ticks) : %d\n",end_time - start_time);
    printf("system time (ticks) : %d\n",buffer.tms_stime);
    printf("user time (ticks) : %d\n",buffer.tms_utime);                     40
}
```

C.2.2 Control Program

The breakpoint is implemented only once and never removed. Afterwards, the control program finishes, since it is not needed any more for test purposes.

The following is the source code for the simulated control program:

```
#include <sys/types.h>
#include <sys/procfs.h>
#include <fcntl.h>
#include <unistd.h>
#include <signal.h>
#include <stdlib.h>
#include <stdio.h>

unsigned long   old_value, address, breakpoint_value = 0x91d02001;
```

```
int             pid, fd;                                                    10
sigset_t        sigs;
prrun           cont_pars;
prgregset_t     regs;
int             flags = PR_RLC;

int main( int argc, char **argv ) {
    if ( argc < 3 ) {
        fprintf( stderr, "usage : %s pid address\n", argv[0] );
        exit( -1 );
    }                                                                       20

    cont_pars.pr_flags = PRCSIG;                /* used in set_break */
    address = atoi( argv[2] );                  /* used everywhere */

    char filename[128];
    sprintf( filename, "/proc/%d", atoi( argv[1] ) );
    fd = open( filename,O_RDWR );               /* attach to process */

    ioctl( fd, PIOCSET, &flags);
    ioctl( fd, PIOCSTOP, 0 );                   /* stop process */          30

    lseek( fd, address, SEEK_SET );
    read( fd, &old_value, 4 );                  /* reading old value */
    lseek( fd, address, SEEK_SET );
    write( fd, &breakpoint_value, 4 );          /* setting new value */
    ioctl( fd, PIOCRUN, &cont_pars );           /* continue without signal */
}
```

C.3 Fast Breakpoints

C.3.1 Target

The breakpoint handler (named `bp_handler`) just simulates saving and restoring of the state of the application. The necessary `save` and `restore` instructions are automatically inserted by the compiler.

The following is the source code for the simulated target program:

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
```

```
#include <sys/times.h>
#include <limits.h>

void function() {
    asm("nop");
    asm("nop");
}

void bp_handler() {
    asm("rd %y,%l5");
    asm("mov %g1,%l6");
    asm("ta 32");
    asm("mov %g1,%l7");
    asm("wr %g0,%l5,%y");
    asm("mov %l7,%g1");
    asm("ta 33");
    asm("mov %l6,%g1");
}

int main( int argc, char** argv ) {
    int i, loops = 1;
    printf("PID : %d\n",getpid());
    printf("function's nop : %d\n",(long*)function + 1);
    printf("bp_handler is at : %d\n",bp_handler);
    getchar();
    if (argc > 1) {
        loops = atoi( argv[1] );
    }
    clock_t start_time, end_time;
    struct tms buffer;
    start_time = times( &buffer );
    for ( i = 0; i < loops; i += 1 ) {
        function();
    }
    end_time = times( &buffer );
    printf("real time (ticks) : %d\n",end_time - start_time);
    printf("system time (ticks) : %d\n",buffer.tms_stime);
    printf("user time (ticks) : %d\n",buffer.tms_utime);
}
```

C.3.2 Control Program

The control program implements the breakpoints and finishes execution, since it is not needed afterwards.

The following is the source code for the simulated control program:

```
#include <sys/types.h>
#include <sys/procfs.h>
#include <fcntl.h>
#include <unistd.h>
#include <signal.h>
#include <stdlib.h>
#include <stdio.h>

unsigned long   handler_address, address, old_value[2], breakpoint_value[2];
int             pid, fd;                                                        10
sigset_t        sigs;
prrun           cont_pars;
prgregset_t     regs;
int             flags = PR_RLC;

int main( int argc, char **argv ) {
    if ( argc < 4 ) {
        fprintf( stderr, "usage : %s pid address handler_address\n", argv[0] );
        exit( -1 );
    }                                                                           20

    cont_pars.pr_flags = PRCSIG;                    /* used in set_break */
    address = atoi( argv[2] );                      /* used everywhere */
    handler_address = atoi(argv[3]);
    breakpoint_value[0] = (1 << 30) | ((handler_address - address) / 4);
    breakpoint_value[1] = (1 << 24);

    char filename[128];
    sprintf( filename, "/proc/%d", atoi( argv[1] ) );
    fd = open( filename,O_RDWR );                   /* attach to process */     30

    ioctl( fd, PIOCSET, &flags);
    ioctl( fd, PIOCSTOP, 0 );                       /* stop process */

    lseek( fd, address, SEEK_SET );
    read( fd, old_value, 8 );                       /* reading old value */
    lseek( fd, address, SEEK_SET );
```

```
    write( fd, breakpoint_value, 8 );          /* setting new value */
    ioctl( fd, PIOCRUN, &cont_pars );          /* continue without signal */
}                                                                          40
```

Bibliography

[1] K. Araki, Z. Furukawa, and J. Cheng. A General Framework for Debugging. *IEEE Software*, 3:14–20, May 1991.

[2] AT&T. *System V Application Binary Interface*. Prentice-Hall, Inc., Englewood Cliffs, New Jersey 07632, 1990.

[3] Peter C. Bates. Debugging Heterogeneous Distributed Systems Using Event-Based Models of Behavior. *ACM Transactions on Computer Systems*, 13(1):1–31, February 1995.

[4] David L. Black. Scheduling Support for Concurrency and Parallelism in the Mach Operating System. Technical Report CMU-CS-90-125, Carnegie Mellon University, 1990.

[5] Peter A. Buhr and Martin Karsten. μC++ Monitoring, Visualization and Debugging Annotated Reference Manual, Version 1.0. Technical report, Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, N2L 3G1, September 1995.

[6] Peter A. Buhr and Richard A. Stroobosscher. μC++ Annotated Reference Manual, Version 4.3. Technical report, Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, N2L 3G1, February 1995.

[7] Deborah Caswell and David L. Black. Implementing a Mach Debugger for Multithreaded Applications. Technical Report CMU-CS-89-154, Carnegie Mellon University, 1989.

[8] Perry A. Emrath and David A. Padua. Automatic Detection of Nondeterminacy in Parallel Programs. In *Proceedings of the ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging, published in ACM SIGPLAN Notices*, volume 24, pages 89–99, Januar 1989.

[9] C. J. Fidge. Partial Orders for Parallel Debugging. In *Proceedings of the ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging, published in ACM SIGPLAN Notices*, volume 24, pages 183–194, January 1989.

[10] J. Gait. A Probe Effect in Concurrent Programs. *Software Practice and Experience*, 16(3):225–233, March 1986.

[11] James Gettys and Robert W. Scheifler. Xlib - C Language Interface. electronic document.

[12] SPARC International. *The SPARC Architecture Manual.* Prentice-Hall, Inc., Englewood Cliffs, New Jersey 07632, 1992.

[13] Rory Alan Jacobs. A Debugger for Multi-Threaded Applications. Master's thesis, The University of Waterloo, 1994.

[14] Gerry Kane and Joe Heinrich. *MIPS RISC Architecture.* Prentice-Hall, Inc., Englewood Cliffs, New Jersey 07632, 1992.

[15] Peter B. Kessler. Fast Breakpoints: Design and Implementation. In *Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation, published in ACM SIGPLAN Notices*, volume 25, pages 78–84, June 1990.

[16] T.J. Killian. Processes as Files. In *Proceedings of the USENIX Conference*, pages 203–207, Summer 1984.

[17] David Kranz, Kirk Johnson, Anant Agarwal, John Kubiatowicz, and Beng-Hong Lim. Integrating Message-Passing and Shared-Memory : Early Experience. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, published in ACM SIGPLAN Notices*, volume 28, pages 54–63, July 1993.

[18] Thomas Kunz. Process Clustering for Distributed Debugging. In *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging, published in ACM SIGPLAN Notices*, volume 28, December 1993.

[19] Leslie Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, July 1978.

[20] Joel McCormack, Paul Asente, and Ralph R. Swick. X Toolkit Intrinsics - C Language X Interface. electronic document.

[21] Charles E. McDowell and David P. Helmbold. Debugging Concurrent Programs. *ACM Computing Surveys*, 21(4):593–622, December 1989.

[22] Emmi Schatz and Barbara G. Ryder. Directed Tracing to Detect Race Conditions. In *Proceedings of the ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging, published in ACM SIGPLAN Notices*, pages 260–262, Santa Cruz, California, May 1991. [Extended abstract].

[23] Steve Sistare, Don Allen, Rich Bowker, Karen Jourdenais, Josh Simons, and Rich Title. A Scalabe Debugger for Massively Parallel Message-Passing Programs. *IEEE Parallel & Distributed Technology*, 1(2):50–56, Summer 1994.

[24] Richard L. Sites. *Alpha Architecture Reference Manual.* Digital Press, 1992.

[25] Pierre E. Sorel, Mariano Fernandez, and Sumit Gosh. A Dynamic Debugger for Asynchronous Distributed Algorithms. *IEEE Software*, 11(1):69–76, January 1994.

[26] Richard M. Stallman and Roland H. Pesch. *Debugging with GDB.* Free Software Foundation, 675 Massachusetts Avenue, Cambridge, MA 02139 USA, 1995.

[27] Bjarne Stroustrup. *The C++ Programming Language.* Addison Wesley, second edition, 1991.

[28] James Alexander Summers. Precedence-Preserving Abstraction for Distributed Debugging. Master's thesis, The University of Waterloo, 1992.

[29] Sun. ioctl(2). manual page.

[30] Sun. mmap(2). manual page.

[31] Sun. proc(4). manual page.

[32] Sun. ptrace(2). manual page.

[33] Sun. wait(2). manual page.

[34] Cygnus Support. Libbfd, the Binary File Descriptor Library. electronic document.

[35] Cygnus Support. Libgdb. electronic document.

[36] Cygnus Support. The "stabs" debug format. electronic document.

[37] David Taylor. A Prototype Debugger for Hermes. In *Proceedings of the 1992 CAS Conference*, volume 1, pages 29–42, Toronto, Ont., Canada, November 1992. IBM Canada Ltd. Laboratory, Centre for Advanced Studies.

[38] United States Department of Defense. *The Programming Language Ada: Reference Manual*, ANSI/MIL-STD-1815A-1983 edition, February 1983. Published by Springer-Verlag.

[39] Mark Weiser. Program Slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352–357, July 1984.

[40] Ronald Scotte Zinn. Efficient Event Generation for Race Detection. Master's thesis, The University of Waterloo, 1993.