A Thread Debugger for Testing and Reproducing Concurrency Scenarios

by

Satyavathi Malladi

Bachelor of Engineering

Computer Science and Engineering

Osmania University, Hyderabad, India - 2000

Submitted to the Department of Electrical Engineering and Computer Science

and the Faculty of the Graduate School of the University of Kansas

in partial fulfillment of the requirements for the degree of

Master of Science

Thesis Committee:

____

Dr. Jerry James: Chairperson

____

Dr. Douglas Niehaus

____

Dr. Joseph Evans

Date Accepted: ____________________

Abstract

Definitive testing of concurrency scenarios has long been a challenging task for developers of concurrent software. Conventional debuggers cannot be used for debugging arbitrary multithreaded programs because the debugger is not aware of the presence of multiple stacks. The problem becomes more pronounced with user level thread libraries because they use their own scheduler and data structures. Context switching, event ordering and synchronization that are inherent in multithreaded programs add to the woes of the developer of concurrent software. The main challenge in debugging threads arises from the fact that thread interleaving can take place at arbitrary places and need not repeat across executions. When debugging a particular execution sequence that results in a crash, it is desirable to repeat the exact sequence to detect the cause of the crash. To provide such a feature it should be possible to record, analyze and play back a particular execution sequence. The BERT architecture is designed to build concurrent software that allows replay of concurrent scenarios. In this architecture the thread library is implemented as a user level thread library (Bthreads library) that allows us to force context switches and obtain the desired thread interleaving. In this thesis, extensions were made to the GNU debugger (GDB) to provide support to debug multithreaded programs written using the Bthreads library. A mechanism is provided to record the context-switching pattern and replay support is provided from within the debugger. Using all these capabilities the developer of concurrent software can have detailed control of the programming model and hence more definitive testing of the software under development is made possible.

Dedicated to my Parents

Acknowledgements

First and foremost, I would like to thank my advisor Dr. Jerry James for guiding and motivating me through out the duration of my thesis. Working for him was a really good experience. I would like to thank Dr. Douglas Niehaus for his timely advice and helpful suggestions. His ideas formed the basis for my thesis. I would like to thank Dr. Joseph Evans for serving on my committee and reviewing my thesis.

I would like to thank my colleague Sunil for his kind cooperation throughout the duration of my thesis. Working with him was a great experience. I would also like to express my sincere thanks to my colleague Raj.

I would like to extend my sincere gratitude to all my friends in KU for their support and encouragement during my stay in Lawrence.

Table of contents

1. INTRODUCTION.. 1

1.1 THESIS CONTRIBUTION 2
1.2 THESIS ORGANIZATION.. 3

2. BACKGROUND 4

2.1 USER LEVEL THREADS.. 4
2.2 KERNEL LEVEL THREADS 5
2.3 DIFFERENCES BETWEEN MULTITHREADED AND SINGLE THREADED PROGRAMS.... 6
2.4 LIMITATIONS WITH CONVENTIONAL DEBUGGERS 6
2.5 REQUIREMENTS OF A THREADS DEBUGGER 7

3. RELATED WORK.. 9

3.1 KDB 9
3.2 SMARTGDB 11
3.3 ML THREADS DEBUGGER 12
3.4 OMNISCIENT DEBUGGING (ODB).. 13
3.5 MACH DEBUGGER.. 14

4. BERT: A CONCURRENT SOFTWARE IMPLEMENTATION PATTERN 16

5. DESIGN 20

5.1 BTHREADS LIBRARY INTERNALS 20
5.1.1 Thread States.. 20
5.1.2 State and resource queues 21
5.1.3 Scheduling.. 23
5.2 PROCESS CONTROL.. 24
5.2.1 Operation of Ptrace 25
5.3 GDB INTERNALS.. 27
5.2.1 Supporting various targets 28
5.2.2 Gdb support for debugging threads.. 29
5.3 REPLAYING A SPECIFIC THREAD INTERLEAVING.. 32

6. IMPLEMENTATION 35

6.1 BTHREAD TARGET SPECIFIC OPERATIONS 35
6.1.1 Attach.. 35
6.1.2 Detach 35
6.1.3 Resume.. 36
6.1.4 Wait 37
6.1.5 Fetch Registers.. 38
6.1.5 Store Registers 39
6.1.6 Thread alive.. 39

6.1.7 Find new threads 40
6.1.8 Xfer memory 40
6.1.9 Mutex info and ConditionVariable info 40
6.2 IDENTIFYING THE BTHREAD TARGET 41
6.3 INITIALIZING THE BTHREAD DEBUGGING MODULE 41
6.4 INTERACTING WITH THE THREAD DEBUGGER INTERFACE (TDI) MODULE 41
6.5 REPLAYING SPECIFIC THREAD INTERLEAVING 42
6.6 USER INTERFACE EXTENSIONS 43
6.7 ATTACHING TCL SCRIPTS TO BREAKPOINTS 44
6.8 THREAD DEBUGGING USING THE ENHANCED DEBUGGER 44
6.8.1 Thread debugging commands 44
6.8.2 Setting breakpoints in multithreaded programs 48
6.8.3 Debugging signals 48
6.9 REPLAYING A RECORDED SCENARIO 49
6.10 OBTAINING A SPECIFIED THREAD INTERLEAVING 51
7. TESTING 53
7.1 CORRECTNESS TESTING OF THE DEBUGGER 53
7.2 TESTING THE REPLAY OF CONCURRENCY SCENARIOS 53
8. CONCLUSIONS AND FUTURE WORK 56
BIBLIOGRAPHY 58

Table of Figures

Figure 2.1 User level threads 4

Figure 3.1 Debugger Design of KDB 10

Figure 3.2 Flow of control in SmartGDB 12

Figure 4.1 Thread-to-BERT mapping 19

Figure 5.1 State Transition Diagram for BThreads Library 22

Figure 5.2 Interaction of gdb with the thread library 31

Figure 6.1 Fetch Registers operation 39

Chapter 1

Introduction

In current software development practice, threads have become an accepted model for expressing concurrency. Because of the differences between the programming models and debugging models in existing thread libraries, the gap between what programmers can write, and what they can test and debug is widening. The two models can match if the thread library can give sufficient control to the user. This includes the ability to force context switches and the ability to make scheduling decisions. Providing such capabilities in a kernel level thread library is difficult because the threads are under the control of the kernel scheduler. A user level thread library is ideal for incorporating such features because the scheduler is at the user level.

When threads run, the execution control interleaves between the threads. The order of interleaving need not be the same in different executions. When debugging multithreaded applications it may be necessary to replay a particular execution sequence to exactly locate the source of a problem. To achieve this, it is necessary to have the capability to record concurrent events taking place in the applications and replay them.

The BERT architecture is designed to build concurrent software that allows replay of concurrency scenarios. In this architecture the thread library (Bthreads [9]) is implemented at the user level. In this library, many user level threads are mapped to one kernel level process due to which only one thread actually runs at a time. Hence, it is possible to have fine-grained control over the order in which threads execute. This thread library allows the programmer to force a context switch to the desired thread and hence it is possible to obtain the desired thread interleaving.

BERT is based on the Reactor [10] pattern. Software is structured as a set of non-blocking handlers, all controlled by the Reactor. The purpose of the Reactor is to examine a set of inputs, choose one that can be processed without blocking, and call the associated handler. The Reactor is thus a pattern for structuring event-driven software.

1.1 Thesis Contribution

We have developed a debugger that allows us to debug multithreaded programs written using the Bthreads library. Capabilities were provided in the Bthreads library that allow us to context switch to any thread that is in the ready queue. Apart from these capabilities, user interface extensions were provided to help the user in visualizing the information necessary for debugging. The capability to record, analyze and replay a particular execution sequence has been provided for programs written using the Bthreads library. A method for obtaining a specified thread interleaving is

also illustrated. Using all these capabilities it is possible for developers of concurrent system software to do more definitive testing of the software under development.

1.2 Thesis Organization

Chapter 2 gives a brief introduction to threads and the requirements of a thread debugger. Chapter 3 discusses related work in the field of debuggers. Chapter 4 discusses BERT software implementation pattern. Chapter 5 discusses the Bthreads library internals and the design decisions made for the thread debugger. Chapter 6 discusses implementation details. Conclusions and scope for future work are presented in chapter 7.

Chapter 2

Background

Concurrency is used in programming to improve performance and better utilize computing resources. Many paradigms and tools have been proposed and implemented to facilitate use of concurrency in programming. Threads are one such mechanism to express and exploit concurrency. Threads come in two main flavors: user level threads and kernel level threads [1].

2.1 User level threads

User level threads are created and maintained at the user level. A user level scheduler is responsible for scheduling these threads. The threads are all mapped to a single process within the kernel.

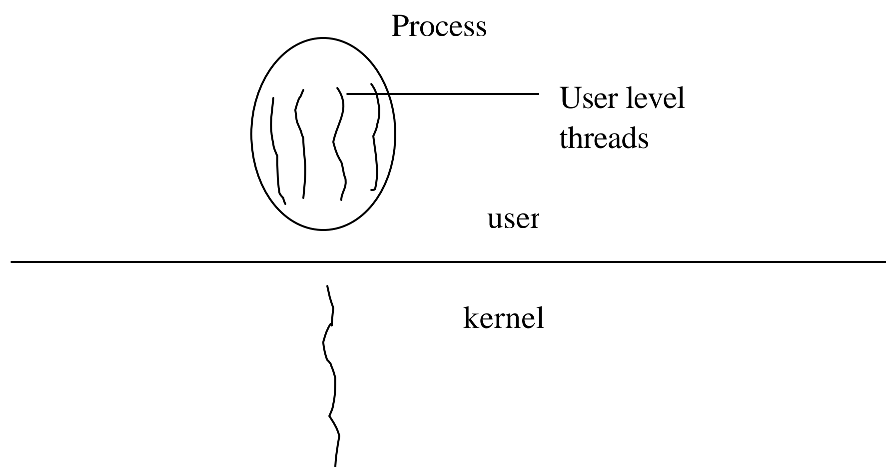

Figure 2.1 User level threads

A user level thread library has several advantages. One main advantage is that it is cheap to create and destroy threads. Since all the thread administration is kept in the user address space, the price of creating a thread is primarily determined by the cost of allocating memory to set up a thread stack. Analogously, destroying a thread mainly involves freeing memory for the stack, which is no longer in use. Both operations are cheap compared to the cost of making system calls.

A second advantage of user-level threads is that switching thread contexts can often be done in just a few instructions. Basically, only the values of the CPU registers need to be stored and subsequently reloaded with the previously stored values of the thread to which it is being switched. There is no need to change memory maps, flush the TLB, perform CPU accounting and so on.

A major drawback of user level threads is that invocation of a blocking system call will block the entire process to which the thread belongs, and thus also all the other threads of that process. Another drawback is that user level threads cannot take advantage of multiple CPUs.

2.2 Kernel Level Threads

Kernel level threads carry out all thread operations in the operating system kernel. In an application using kernel level threads, when one thread blocks on I/O, other threads can execute without blocking the entire process. The disadvantage of using

kernel level threads, however is that, every thread operation has to be carried out by the kernel, requiring a system call. Switching threads may be nearly as expensive as switching process contexts.

2.3 Differences between multithreaded and single threaded programs

Multithreaded programs differ in several ways from single threaded programs [2].

1. Execution control interleaves between the threads in a multithreaded program. In a single threaded program no such interleaving takes place.

2.Threads may voluntarily suspend and resume execution, or they might do it because of events like signals.

3.The order of interleaving between the threads is partially determined by synchronization between the threads.

2.4 Limitations with conventional debuggers

There are several problems with current debuggers that pose hurdles in effective debugging of multithreaded programs [2].

1. The debugger does not have sufficient information to display the state of threads and other synchronization primitives. This information is very important for debugging multithreaded programs. The problem is more acute with user level thread

libraries because the thread library has its own scheduler and data structures to manipulate threads.

2. It might be necessary to forcibly suspend the execution of a thread and resume some other thread to investigate the behavior of the program under different circumstances. This might not be possible with many thread libraries.

3. There is no information about threads waiting on synchronization objects such as mutexes and condition variables.

Because of these differences, conventional debuggers used for debugging single threaded programs cannot be effective in debugging a multithreaded program. This factor plays a major role in the design of a thread debugger.

2.5 Requirements of a threads debugger

To be able to debug a multithreaded application effectively, the debugger must be familiar with the application and how each of the threads synchronize with each other. For a debugger to know about multiple stacks, it must understand the runtime structure of the concurrent system such as finding all the threads and the points their executions have reached. Hence, the debugger should be kept informed about the dynamic structure of the application. Once this information is obtained, it can be displayed in various ways to provide the user with a visualization of the concurrent control flow. Such visualization when provided would greatly help the user in tracking algorithmic or other errors which cause the program to behave erratically.

When threads run, the execution control interleaves between the threads. The order of interleaving need not be the same in different executions. When debugging multithreaded applications it may be necessary to replay a particular execution sequence of the threads to exactly locate the source of a problem. To achieve this it is necessary to have the capability to record concurrent events taking place in the applications and replay them. The programmer might sometimes want to test the concurrent software for a thread interleaving. To achieve this it is necessary to have scheduling control, wherein threads can be forcibly suspended and resumed from within the debugger.

A debugger should provide a set of standard capabilities when working with a thread package. It should be able to identify all the threads and display their state. It should be possible to view the stack trace of all the threads and set thread specific breakpoints. It should also be possible to make state enquiries of synchronization objects like mutexes and condition variables.

Chapter 3

Related work

Several thread debuggers were developed for debugging various types of user level and kernel level thread packages. This chapter discusses the architecture of several such debuggers. Various issues related to thread debugger design are also discussed here.

3.1 KDB

KDB (Kalli's debugger) [6] is a concurrent debugger built over GDB. It runs on Unix based symmetric shared-memory multiprocessors with the goal of achieving independent control of user-level threads. KDB supports the μC++ execution environment, which shares all data, and has multiple kernel and user-level threads.

Instead of using vendor specific kernel threads, μC++ obtains kernel threads from UNIX processes. A kernel thread is created by forking a UNIX process and *mmapping* all data to be shared with the parent process. One limitation of *mmap* is that the code image cannot be shared, which means that there are multiple code

images, one for each kernel thread. Since there are multiple code images, the debugger should take care to *set* and *reset* breakpoints in each code image.

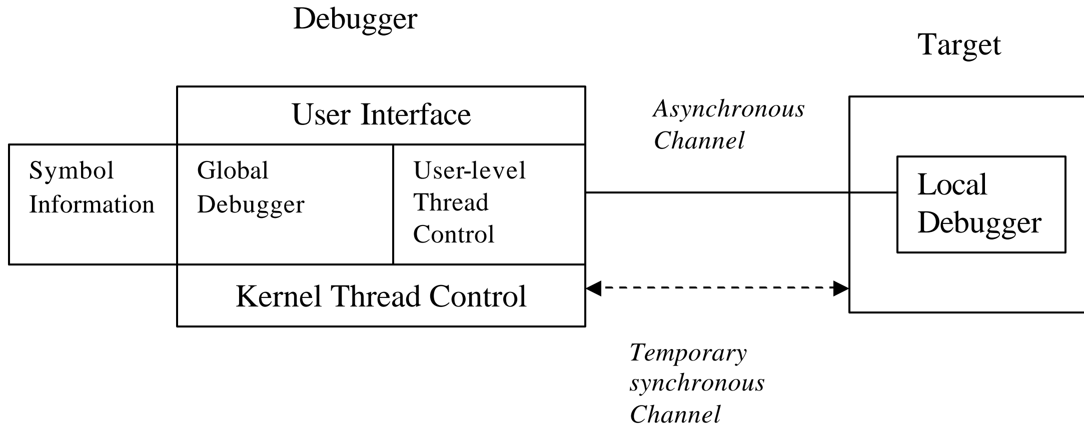

Figure 3.1 Debugger Design of KDB [6]

To achieve asynchronous execution of an application and the debugger, a part of the debugger, called the local debugger is distributed into the target application. Communication with the application process is done by the global debugger using two different channels. The first channel is synchronous because it is implemented by Unix debugging primitives; this channel is temporary, lasting only as long as necessary to modify an application's code, e.g. to set and reset breakpoints. The second channel is a synchronous channel. It operates via a socket between the global and local debugger, and communicates events generated by the user interacting with the global debugger. The internal as well as external interactions of the global debugger are asynchronous because the global debugger is itself a multithreaded application, written using μC++.

3.2 SmartGDB

SmartGDB [13] is a debugger modified from GDB. The mechanism exploited by SmartGDB is the execution of Tcl procedures at breakpoints. Commands were provided that let the user place a breakpoint or a watch point in the code and associate a Tcl procedure to be executed when the program reaches that particular breakpoint.

The flow of control in SmartGDB is shown in the form of a state diagram in figure-3.2. SmartGDB's programmability and flexibility is supported by the Tcl procedures that can be executed at breakpoints or watch points. By proper and intelligent use of these breakpoint handlers, it is possible to construct high-level abstractions that can greatly aid in the debugging process. Thus by simple use of scripts it is possible to debug multithreaded applications, perform state exploration, and expose errors due to concurrency [7].

Debugging with SmartGDB involves the following steps:

- Identify breakpoints for state exploration.
- Prepare scripts to test program states at particular breakpoints.
- Install breakpoints and the associated handlers.
- Run the program.

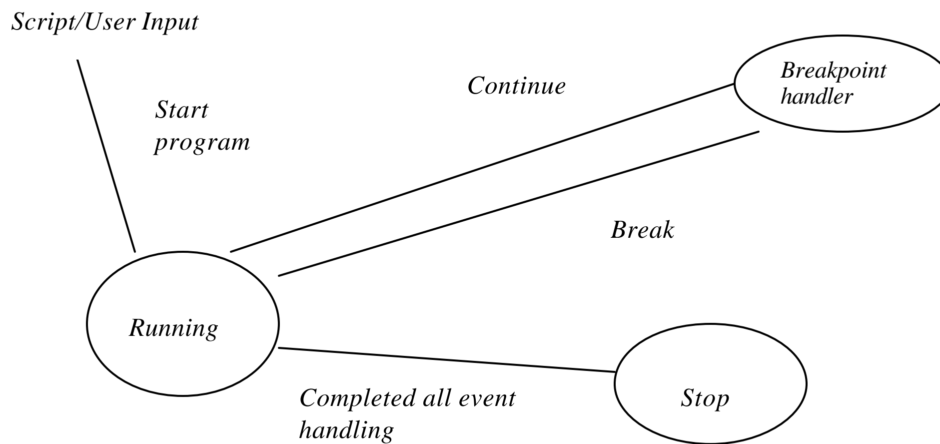

Figure 3.2 Flow of control in SmartGDB [7]

3.3 ML Threads Debugger

This is an interactive debugger for the Standard ML language with thread extensions [5]. Automatic instrumentation of the program's source code is used as the debugging approach by this debugger. When code is compiled under debugger control, special hooks are inserted at frequent intervals. When the program runs, the debugger can use these hooks to gain control, answer user queries and support breakpoints. To debug ML threads, a special debugger version of the thread library containing instrumented versions of thread routines is used.

Each thread primitive in this instrumented library has a hook point. Breakpoints can either be specified at the source level or as a value of a Software Instruction Counter (SIC), which is the number of hook points encountered so far by the program.

Each time a hook point is encountered, the SIC is incremented. The SIC is used to implement the feature of reverse execution. The programmer can jump back and forth in time and replay his program. Some support is provided to log thread interaction and provide for replay that may be useful for reproducing errors due to concurrent interaction of threads. The use of logging techniques to capture non-determinism and simulation based approaches were both considered in this debugger.

3.4 Omniscient Debugging (ODB)

ODB [19] is a debugging tool that allows developers to step backwards through the execution of a program to determine programming errors. This tool was written using Java ™ technology. ODB records each change of state in the target application, thus permitting the developers to navigate backwards in time. Knowledge of all previous values for the application's objects and variables simplifies the task of debugging programs significantly.

ODB inserts code into a program's class files. As the program is run, this code collects information on each method call and variable assignment. It assigns time stamps to each event and stores them in a log file. After the program executes, it

displays them in a graphical user interface (GUI). The developer can then review the behavior of the objects, variables, and method calls from the GUI to debug the program.

With Omniscient Debugging, the developer can look at the history of an entire program run and select the pertinent time stamps. ODB displays the trace history of every method call. The developer can use this display to step though the history of the program, selecting the most interesting method invocation for further examination.

In addition, a TTY Output pane displays the lines printed by the program, a Code pane shows the line of code that produced the selected time stamp, a Stack pane displays the current stack, and a Threads pane shows the current thread. The developer can select lines from any of these panes to *revert* the display to the state of the program when that line executed.

3.5 Mach Debugger

This debugger was a modification to GDB for debugging multithreaded applications under the Mach operating system [11]. The Unix *ptrace* system call was modified to deal with multiple threads. The reading and writing of specific memory locations in the application is done by using Mach's kernel *read/write* operations, which are used to directly access memory of another task. Also Mach places all stacks of threads in

kernel memory, so all stacks are accessible, from which the register values can be accessed.

The calls *thread_suspend* and *thread_resume* are used for execution control, to control application threads. Single stepping is done by accessing process control registers of the desired thread, setting the trace bit and performing *task_resume*.

To overcome the limitations imposed by Unix signals, Mach provides a message-based exception handling facility. This exception handling facility is useful in recording all exceptions that have happened to the application since it was last modified. The debugger uses this exception handling facility to find out the global state of the application, such as concurrent exceptions due to multiple breakpoints.

To allow independent examination and modification of thread states, extensions were made to GDB's user interface. New commands were added to identify threads and to select a thread to manipulate. The user can manipulate one thread at a time like single stepping a particular thread. The Mach debugger provides basic state exploration commands with no support for a flexible interface or constructs to debug concurrency related errors.

Chapter 4

BERT: A concurrent software implementation pattern

One of the major challenges in developing concurrent software is that executions of the software are generally irreproducible. This lack of reproducibility stems from an inability to control event delivery at the finest level of detail, or at the most fundamental level of system implementation; e.g., the timing of thread context switches, message delays, and signal delivery. The inability to control all aspects of system behavior affecting the behavior of the program hampers both software testing and debugging. Testing is hampered because specific concurrency scenarios cannot be reproduced on demand. Debugging is troublesome when an incorrect synchronization structure leads to intermittent incorrect behavior. The average time between manifestations of the incorrect behavior can be very large or entirely dependent on uncontrollable decisions made by the system, and on external events whose timing is arbitrary, making the debugging process very difficult.

High fidelity experimental monitoring, simulation, and steering are beneficial, and often necessary, to achieve an environment with an adequately detailed level of information and control for reproducibility. Monitoring provides relevant information about the system. Simulation provides the ability to work with larger systems than are available in practice, and provides control over events that are not controllable in the larger systems than are available in practice, and provides control over events that are not controllable in the real system. Steering enables the playing out of specific scenarios, for both debugging and testing purposes.

BERT is an implementation pattern for reflecting in software architecture semantic constraints, or specified semantics, behaviors and scenarios, of interactions of objects in a concurrent application. It provides the system architect with the means of constructing controlled experiments. It represents a fundamental advance in our ability to build, test, and debug concurrent software systems.

BERT is based on the Reactor pattern. Software is structured as a set of non-blocking handlers, all controlled by the Reactor. The purpose of the Reactor is to examine a set of inputs, choose one that can be processed without blocking, and call the associated handler. The non-blocking nature of the handlers is very important. To avoid blocking the entire process, a handler may be forced to register another input source and associated handler with the Reactor. The Reactor is thus a pattern for structuring event-driven software.

As a software implementation pattern, BERT is very flexible. It can be implemented by hand in each application, submerged in a middleware product offering a higher-level set of services, or built into an object oriented compiler. Furthermore, BERT is sufficient, in the sense that we can represent everything that matters about a concurrency scenario. Using it, we can record sufficient information so that accurate playback is possible.

Multithreaded software, whether using kernel-level or user-level threads, suffers from lack of repeatability. This can be alleviated for both development and maintenance purposes by mapping threading primitives onto the BERT pattern. The result is a user-level thread package that has control of the usually non-deterministic elements of the threading model. The mapping is accomplished by keeping a list of CPU contexts, and linking with the BERT library before linking with standard system libraries. The BERT library provides the standard API for system calls that can block, but takes extra actions to determine when those calls can be made without blocking the entire process. In short, we ensure that all activities affecting concurrency, including blocking system calls, pass through BERT and are thus subject to various forms of manipulation. The degree to which this approach can succeed will vary with the support provided by the underlying operating system.

BERT has a common handler for events affecting concurrency, such as a timer event or a potentially blocking system call. That handler invokes the thread scheduler,

which chooses a thread context to run from the ready list, as shown in Figure 5.1. This architecture provides an opportunity for experimenting with scheduling policies. Since the scheduler is in user space, we can allow the user to tune the scheduler to the application by setting parameters of interest. If necessary, we can even allow the user to specify the scheduling function, in order to consider application semantics when making scheduling decisions. On single-processor machines, we expect to achieve very good performance in this manner, due to using a low overhead user thread package that does not block individual threads on normally blocking system calls, and which makes scheduling decisions based on the application state.

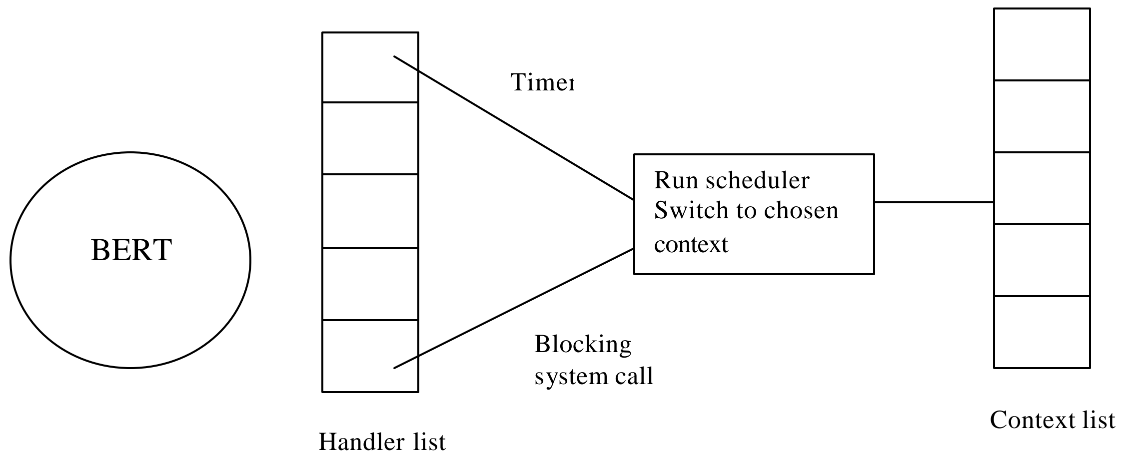

Figure 4.1 Thread-to-BERT mapping

Chapter 5

Design

In this chapter, we first present the internals of the BThreads library [9]. Then the internals of the GDB thread support module and the mechanism of replaying particular execution sequences are presented.

5.1 Bthreads library internals

5.1.1 Thread States

The Bthreads library is a user level thread library. So, the states of all the threads are maintained in user level. The possible thread states for Bthreads Library are:

Ready:

A thread in this state is ready to be scheduled.

Blocked:

The state associated with a thread blocked on a synchronization device like a mutex, condition variable or waitlock, or which is waiting on termination of another thread.

Running:

The state associated with the currently running thread.

Sigwait:

The state associated with a thread blocked in *sigwait*. A thread calls the *sigwait* function to synchronously wait for asynchronous signals and is blocked until one of the signals in the set is delivered.

Waiting:

The state associated with a thread waiting for completion of an I/O request.

Killed:

The state associated with a terminated thread.

5.1.2 State and resource queues

The following queues are present in Bthreads:

Ready Queue:

This queue holds threads in the ready state.

Termination Queue:

This queue holds terminated threads in the detached state. A thread can be in the detached state or the joinable state. If it is in the detached state, it runs independently of any other thread. If it is in the joinable state, another thread can wait on its termination. A detached thread memory resources, such as the thread stack, are deallocated when it terminates.

Sigwait Queue:

This queue holds threads blocked in sigwait.

The state transition diagram for BThreads is shown in figure 5.1.

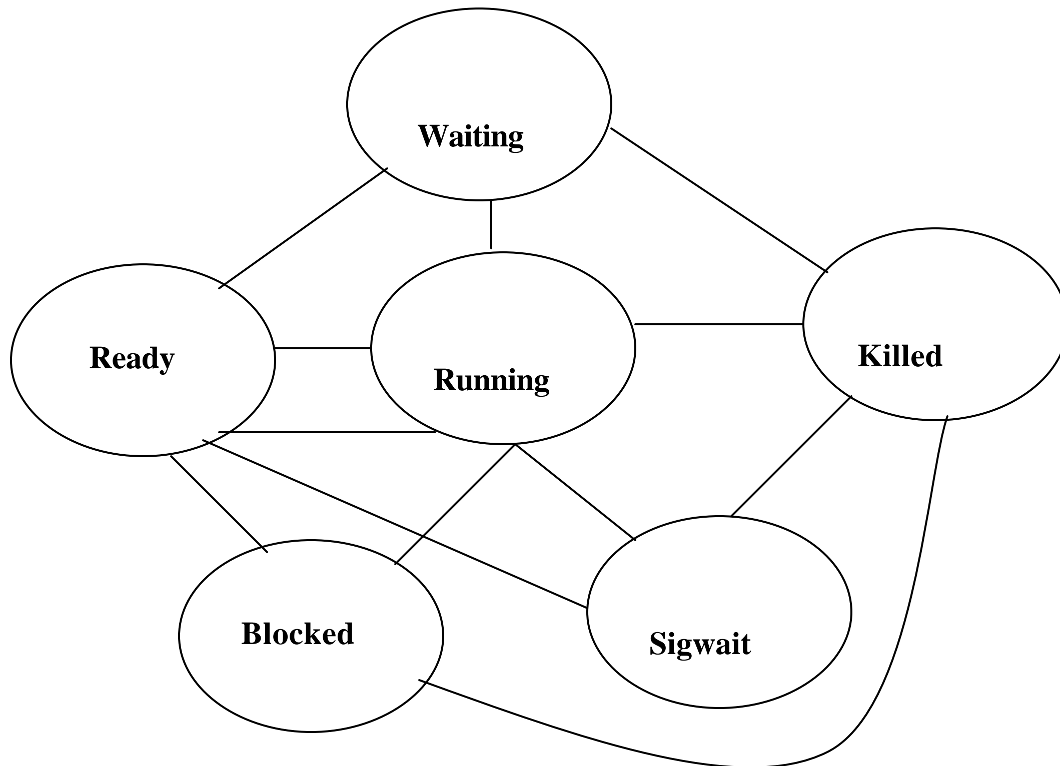

Figure 5.1 State Transition Diagram for BThreads Library [9]

As shown in figure 5.1, when a thread is created it is in the ready state. It is scheduled some time later and enters the running state. When a thread is running, it can make the transition to:

- The waiting state by making I/O calls.
- The blocked state due to synchronization calls.
- The sigwait state by calling sigwait.
- The killed state by calling thread exit function or due to cancellation.
- The ready state as it ran till completion of its quantum.

Round Robin (RR) scheduling policy is used by the Bthreads library. RR Scheduling is a preemptive scheduling policy and the quantum is the maximum amount of time a thread can run before another thread is scheduled.

The Reactor [10] is called whenever the scheduler is invoked. It checks for availability of pending I/O requests and inserts threads that are waiting for I/O into the ready queue if I/O is available. When a thread is blocked on a synchronization variable, its state is changed to ready by the thread that signals the synchronization variable. A thread enters the sigwait state when it calls *sigwait*. It makes the transition to the ready state when one of the signals that it is waiting for is ready to be delivered. Cancellation is a mechanism by which a thread can terminate execution of another thread. If it is of asynchronous type, the thread receiving the cancellation request terminates immediately. If it is of deferred type, thread receiving cancellation request exits only when it reaches cancellation points. As shown it is possible to make the transition from any state, except ready, to the killed state if cancellation is enabled and is of asynchronous type. The transition to the killed state is also possible if the thread exit function is called or due to deferred cancellation.

5.1.3 Scheduling

In the Bthreads library, scheduling is done in a Round Robin (RR) fashion. RR Scheduling is inherently preemptive. Preemption can be realized by setting up a timer that generates a scheduler timer signal at periodic intervals. The RR Quantum is the

time interval between scheduler timing signals. A FIFO (First In First Out) scheduling policy that is inherently non-preemptive can be realized by turning off generation of the scheduler timer signal. When the scheduler timer signal is turned off, scheduling can take place under control of the debugger. This can be realized by providing the debugger with an interface to switch to an arbitrary thread in the ready state. It is then possible to interleave instructions from different threads in an arbitrary way and thus detect potential synchronization deadlocks. In the design of the Bthreads, library provision was made for priority-based scheduling by including priority information in the Thread Control Block (TCB). The TCB is analogous to the Process Control Block (PCB). The TCB has all the necessary information about an active thread for managing all library operations.

5.2 Process Control

Process control is the ability to inspect a running process and alter its execution [3]. Process control is a key ability needed in debuggers because debuggers need to have access to the address space of the process to perform some basic actions such as inserting breakpoints.

A virtual file system is used to implement process control in many Unix based systems. Depending on the system the virtual file system may represent each running process by either a single file or a collection of files. The debugging process ascertains the state and modifies the execution of the process being debugged. The

debugging process is termed the *tracing* process and the debugged process is termed the *inferior* process. The tracing process achieves the process of tracing by accessing the file representing the inferior process with standard file system operations: reads, writes, lseeks, polls, and ioctls. When writes are performed on these virtual file system files, directives are passed to the kernel indicating what action is to be applied to the inferior process. Some file systems may allow a tracing process to write directly to the inferior process's address space. The current state of the inferior process can be read from these files. A poll on a virtual file for a process returns when the process stops due to a controlling action by the tracing process, serving as notification of an event.

On some systems like Linux, the system call *ptrace* is used to implement the process control mechanism. A tracing process calls *ptrace* with arguments identifying the inferior process and specifying the controlling action to be applied. The kernel inserts the tracing process as the parent when it uses the *ptrace* system call. The tracing process calls the *wait* system call to receive the stopping event information. Upon the receipt of a signal, the *wait* system call returns, and based on the reason for which *wait* returned, the appropriate controlling action can be taken.

5.2.1 Operation of Ptrace

The *ptrace* [15] system call provides a means by which a parent process may observe and control the execution of another process, and examine and change its core image

and registers. Trace can be initiated in the parent by calling fork and having the resulting child do a PTRACE_TRACEME, followed by an *exec*. PTRACE_ATTACH can alternatively be used by the parent to commence trace of an existing process. While being traced, the child will stop each time a signal is delivered, even if the signal is being ignored. The parent will be notified at its next *wait* and may inspect and modify the child process while it is stopped. The parent then causes the child to continue, optionally ignoring the delivered signal or even delivering a different signal instead. When the parent is finished tracing, it can terminate the child with PTRACE_KILL or cause it to continue executing in a normal, untraced mode via PTRACE_DETACH.

```
long int ptrace(enum __ptrace_request request, pid_t pid, void * addr, void * data)
```

The value of request determines the action to be performed. Some of the main request operations are explained below:

PTRACE_TRACEME

Indicates that this process is to be traced by its parent. Any signal (except SIGKILL) delivered to this process will cause it to stop and its parent to be notified via *wait*. Also, all subsequent calls to *exec* by this process will cause a SIGTRAP to be sent to it, giving the parent a chance to gain control before the new program begins execution.

PTRACE_GETREGS, PTRACE_GETFPREGS

Copies the child's general purpose or floating point registers, respectively to location *data* in the parent.

PTRACE_SETREGS, PTRACE_SETFPREGS

Copies the child's general purpose or floating point registers, respectively, from location *data* in the parent.

PTRACE_CONT

Restarts the stopped child process. If data is non-zero or SIGSTOP, it is interpreted as a signal to be delivered to the child; otherwise no signal is delivered.

PTRACE_SYSCALL, PTRACE_SINGLESTEP

Restarts the stopped child as for PTRACE_CONT, but arranges for the child to be stopped at the next entry to or exit from a system call, or after execution of a single instruction, respectively.

PTRACE_KILL

Sends the child a SIGKILL to terminate it.

PTRACE_ATTACH

Attaches to the process specified in pid, making it a traced child of the current process; the behavior of the child is as if it had done a PTRACE_TRACEME. The current process actually becomes the parent of the child process for most purposes, but a *getpid* by the child will still return the pid of the original parent.

PTRACE_DETACH

Restarts the stopped child as for PTRACE_ATTACH, and the effects of PTRACE_ME.

5.3 GDB internals

GDB is a breakpoint debugger. Debugging is done by inserting breakpoints at various places and then examining the process state. GDB attaches to a process by means of

the system call *ptrace*. Then it executes a *wait* on the inferior process. *Wait* returns whenever any signal has to be delivered to the inferior process or when the inferior exits. GDB looks at the reason for which the inferior stopped and takes the appropriate action.

How a breakpoint is inserted:

A trap instruction is written at the place where the breakpoint should be inserted. When the trap instruction is executed, a SIGTRAP is generated by the kernel that is first delivered to the debugger. On receiving a SIGTRAP, the debugger knows that the breakpoint is hit and takes the necessary action. To continue the process, GDB replaces the trap instruction with its original content; reduces PC by 1 and then continues execution.

5.2.1 Supporting various targets

GDB needs to be able to support various targets. To facilitate this, each action that needs to be performed on the debugged process is represented by a field of the following *target_ops* structure:

```
struct target_ops
{
char *to_shortname; /* Name this target type */
char *to_longname; /* Name for printing */
char *to_doc; /* Documentation. Does not include trailing
newline, and starts with a one-line description  */
void (*to_open) PARAMS ((char *, int));
void (*to_close) PARAMS ((int));
void (*to_attach) PARAMS ((char *, int));
void (*to_detach) PARAMS ((char *, int));
```

```
void (*to_resume) PARAMS ((int, int, enum target_signal));
int (*to_wait) PARAMS ((int, struct target_waitstatus *));
………/* Several other such functions */
……
};
```

This structure contains pointers to target specific functions. Each target supported by GDB has its own relevant implementation of the functions. Not all functions are implemented for all targets.

A *target* is an interface between the debugger and a particular kind of file or process. Targets can be *stacked* in *strata,* so that more than one target can potentially respond to a request. In particular, memory accesses will walk down the stack of targets until they find a target that is interested in handling that particular address. *Strata* are artificial boundaries on the stack, within which particular kinds of targets live. Strata exist so that people don't get confused by pushing e.g. a process target and then a file target, and wondering why they can't see the current values of variables any more (the file target is handling them and they never get to the process target). So, when you push a file target, it goes into the file stratum, which is always below the process stratum.

5.2.2 Gdb support for debugging threads

Gdb provides a generic framework for debugging threads of any thread library. This framework has all the required data structures needed to maintain information about

threads in gdb. The main data structure is *thread_info.* This contains all the information generic to any thread package that is needed by the debugger. It has a field *private_thread_info* that contains information specific to a particular thread package. If any operation in the framework is target specific, then it is implemented as a target specific operation in the support module provided for each thread package.

In the gdb framework, there are certain operations, such as fetching registers, transferring memory etc, which are target specific. These operations are implemented separately for each target. At run time the target is identified and the operation mapped to the appropriate target specific operation. This design of gdb makes the addition of new targets easier; since one only needs to identify the target specific operations and implement them.

Gdb provides commands for obtaining various kinds of debugging information about the threads. These commands are built over the generic framework for debugging threads. Hence, separate implementations need not be provided for each thread package. If there are any target specific operations in implementing these commands then they are dynamically mapped to the appropriate operation for the thread package being debugged. In this thesis, the support provided for Bthreads is restricted to the x86 platforms.

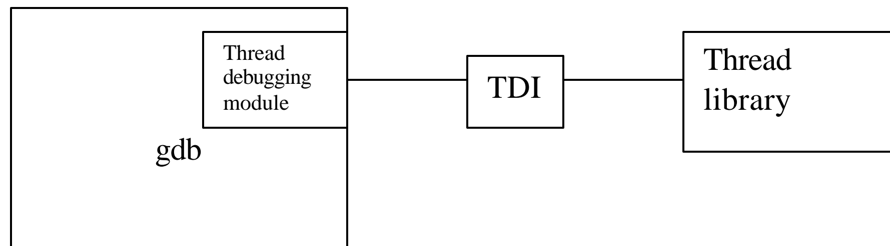

Figure 5.2 Interaction of gdb with the thread library

The debugger should be aware of the internal data structures and various events taking place within the thread library. The Thread Debugger Interface (TDI) provides a means by which the debugger can obtain the needed information from the thread library. The TDI has routines that read values from the necessary data structures in the process address space and return them to gdb. The key data structures that are accessed from the thread library data structure include:

thread_ handles : An array that has information about each thread in the thread library.

__thread_handles_num : The number of existing threads.

__linuxthreads_thread_threads_max: Maximum number of threads

__linuxthreads_thread_sizeof_descr : Size of the thread descriptor structure.

__linuxthreads_create_event: The event of a new thread being created.

__linuxthreads_death_event: The event of the death of a thread.

currentthread : The currently executing thread.

Mutexhead: The head of the mutex queue.

Conditionhead: The head of the condition queue.

The debugger maintains information about the threads in its own data structures. When implementing routines that are specific to the thread library, the debugger uses the TDI routines when accessing information from the thread library.

5.3 Replaying a specific thread interleaving

The execution of multithreaded programs is inherently non-deterministic. As a result, thread executions cannot be repeated. When thread programs behave aberrantly, then replaying a particular execution sequence would be extremely helpful in debugging. But GDB does not provide the facility to specify a particular thread interleaving sequence.

The ability to record a specific thread execution has been implemented in the Bthreads library [9]. In this thesis, a method to replay an execution sequence from the debugger is illustrated. Also a facility has been provided wherein a user can test a hypothesis using a specific thread interleaving.

To replay a particular thread interleaving sequence the events that need to be recorded are those that cause the threads to switch context. The following are the reasons for a context switch to take place:

1.A thread tries to acquire a mutex that is held by some other thread. Then the thread is put in the wait queue associated with the mutex and the thread state becomes blocked.

2.A thread is waiting on a condition variable that is has to be signaled by some other thread. Then the thread is put in the wait queue associated with the condition variable and the thread state becomes blocked.

3.A thread becomes blocked when performing I/O.

4.The scheduling timer expires and generates the signal SIGPROF.

5.A thread terminates because of the library call *thread_exit*.

6.A thread goes to the blocked state because of the library call *thread_yield*.

7.A thread goes to blocked state because of the library call *thread_join*.

In all the above cases except 4, the scheduling action takes place at fixed points in the program deterministically, also it repeats at the same place during replay. The only case that is non-deterministic and does not repeat at the same place in different executions is 4. This event is caused by expiration of the scheduling timer. Due to the granularity constraints on the clock, this signal need not be generated at exactly the same place in the execution of the program every time. Therefore, the exact PC location at which this signal was generated needs to be recorded. During replay when this PC is reached, the signal can be delivered by the debugger to the process so that the context switch takes place at exactly the same place.

Signals can also be delivered to a process asynchronously from an outside source. Hence, the PC location at which any signal was delivered externally to the multithreaded program should also be recorded. During replay all such signals can be delivered to the process externally by the debugger.

PC value alone is not sufficient when replaying events. This is because a particular PC value can be reached more than once in a program space (because of loops, jumps etc). Hence, it is also necessary to record the number of times a particular PC was reached. To capture the information about how many times a particular PC was reached, the concept of basic block can be used. A basic block is a software instruction. (A software instruction is a collection of hardware instructions that are always executed together). When a program is compiled, compiler divides the program into basic blocks. Compilers provide features that allow us to record the sequence of basic blocks executed. By counting the number of times the basic block in which the PC is present is entered, it is possible to determine the number of times a PC is reached. The *gcc* compiler [14] can be made to emit code that records the basic block that is entered in a global variable in the address space of the process being debugged. Thus we can know how many times the basic block in which the PC is located was entered by the time the signal was delivered. By knowing this count, we can know exactly when to deliver the signal during the replay.

Chapter 6

Implementation

6.1 Bthreads target specific operations

The following are the target specific operations that are implemented to provide debugging support for Bthreads in GDB.

6.1.1 Attach

Attach to an existing process, then initialize for debugging it and wait for the trace trap that results fom attaching. The process stratum level attach operation can be called here directly because there is only one process that is present inside the kernel. The process stratum operation attach uses the *ptrace* system call with PTRACE_ATTACH as the requested operation.

6.1.2 Detach

Takes a program previously attached to and detaches it. The process stratum level detach is called here directly. The process stratum detach uses the *ptrace* system call with PTRACE_DETACH as the requested operation.

6.1.3 Resume

Resume the execution of a process. The process can be resumed or continued in single step mode.

Whenever a signal is delivered to the inferior process, the *wait* system call in the tracing process (gdb) returns. When continuing the process the signal has to be delivered to the process. (gdb can decide not to deliver this signal if the user specifies so). Hence, the resume operation should allow us to continue a process with a signal.

When a breakpoint is hit, program execution is stopped and control is given to the user. When the user issues the step command (to single step) or continue command the execution of the process should proceed. The target specific resume is called to resume the process.

In the case of Bthreads, when a breakpoint is hit in a thread, then it is always the currently running thread that stops because all the user level threads are mapped to a single process inside the kernel. Hence, to continue a thread stopped at a breakpoint, it is sufficient to simply continue the process that is currently executing. The *ptrace* system call is used for doing this. The requested action is PTRACE_CONT to continue the process, or PTRACE_SINGLESTEP to single step the process.

6.1.4 Wait

Wait for any threads to stop. This operation is called in gdb after resuming a process. After resuming a process, gdb waits for any threads to stop and then based on the reason for which the process has stopped, it takes the appropriate action. In addition, any target specific action that needs to be taken when a thread stops can be done here. The *wait* system call is used to wait for some signal to be delivered to the process. To find out the thread id of the thread that was executing when the signal was delivered, the *find_active_thread* routine is called, which obtains the currently running thread through the TDI.

There are several events in the thread library that the debugger would be interested in being notified about:

1.Thread creation event: Gdb has to notify the user whenever a new thread is created.

2.Thread death event: Gdb has to notify the user whenever a thread exits.

To make these events (taking place inside the process) known to gdb, breakpoints are set in the thread library in the thread creation and thread death routines. The addresses at which the breakpoints are inserted are stored inside gdb by the thread-debugging module. The signal that is raised when a breakpoint is hit is SIGTRAP. So, when the program stops due to SIGTRAP, we check to see whether it stopped at any of the events in which we are interested (by comparing the address at which the breakpoint was hit with the address stored in the gdb thread debugging module).

In case the event is thread death, then the user is notified that a thread exited and in case the thread is not already in the thread list in gdb, then we report that a spurious thread death event has taken place.

6.1.5 Fetch Registers

The Bthreads library is a user level thread library. The registers for the threads are stored in the user level. When context switch takes place to a particular thread, the registers for the current thread are loaded on to the machine and the registers of the previously running threads are stored in the user level. Hence, while performing a fetch registers operation, if the thread for which registers are being fetched is the current thread, then the operation should be performed on the currently running process. Otherwise, the registers should be obtained (through the thread debug interface) from the thread library.

To obtain the registers from the thread library the TDI routine to fetch registers is used. To fetch registers for the currently running thread, the *ptrace* system call is used with PTRACE_GETREGS as the requested operation for obtaining general-purpose registers and PTRACE_GETFPREGS as the requested operation for obtaining floating point registers.

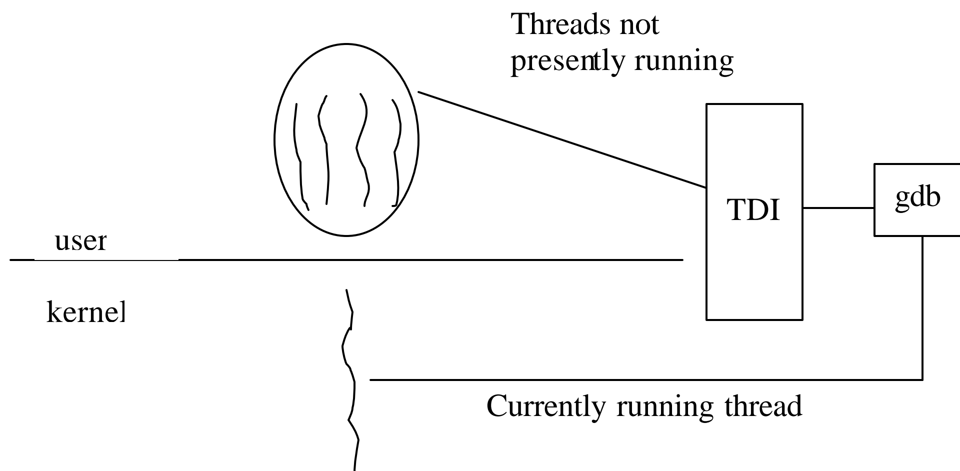

Figure 6.1 Fetch Registers operation

The format in which registers are stored within the thread library is different from that expected by gdb. Hence, mapping to the gdb format must be done before supplying the registers to gdb.

6.1.5 Store Registers

When storing registers for the currently running thread the operation should be performed on the currently running process. Otherwise, the operation should be performed through the TDI.

6.1.6 Thread alive

This operation is used to determine whether a particular thread is alive. The state of the thread is queried from the TDI to determine whether the thread is alive or not.

6.1.7 Find new threads

The TDI is used to query the existing threads. If any thread is not already in the gdb thread list, then it is added.

6.1.8 Xfer memory

This operation is used to transfer data from the process address space to the gdb address space. All the threads share the same address space. They have separate stacks and register sets. While transferring data from the process address space, if the current thread needs to be accessed, then the registers are obtained from the currently running process, otherwise the registers are accessed through the TDI (for a thread which is not currently running).

6.1.9 Mutex info and ConditionVariable info

The mutexes and condition variables created by the user are stored in linked lists in the thread library. The head of these lists are stored in the variables *mutexhead* and *conditionhead* respectively. The *mutexhead* and *conditionhead* variables are read from the process address space and then all the information contained in the linked lists is read and displayed to the user.

6.2 Identifying the Bthread target

When gdb loads the symbols of an executable, then based on the shared objects that are linked while creating the executable, the target thread library is identified and the appropriate target specific operations are loaded into gdb.

6.3 Initializing the Bthread debugging module

All the Bthread target specific operations are initialized. These operations are then pushed to the target stack. Then this module is added to the objfile event chain. (This is a list of shared objects to which the program is linked). As a result, whenever a new objectfile is detected an attempt is made to open a connection to the thread library.

6.4 Interacting with the thread debugger interface (TDI) module

The thread debug interface is implemented as a separate module and the symbols from it are read using the system call *dlsym*. Function pointers are declared in the thread debugger module for each method in the TDI module. Symbols are read from the TDI for the pointers to the TDI methods and the function pointers in the thread debugger module are set to them.

6.5 Replaying specific thread interleaving

The execution of multithreaded programs is inherently non-deterministic. As a result, thread execution sequences cannot be repeated on demand. When thread programs behave aberrantly, then replaying a particular sequence would be extremely helpful in debugging. But gdb does not provide the facility to specify a specific thread interleaving sequence for a program.

The feature to record a specific thread execution and replaying it from within the debugger has been illustrated. Also a facility has been provided wherein a user can test a hypothesis using a specific thread interleaving.

The thread library has been enhanced to provide a function that allows the user to lock the scheduler, i.e. the scheduler will no longer decide which thread is to be executed next. In addition, a function has been provided to switch context to an arbitrary thread in the run queue. These functions can be called from gdb using the *call* command. To test a particular hypothesis the user should first lock the scheduler. Then he should set breakpoints at places where he wants context switches to take place. When a breakpoint is hit the user can call the function to switch the context to any desired thread.

The user can alternately record a particular aberrant sequence and replay it from the debugger. The recording mechanism is integrated into the *gcc-2.95* compiler. The

compiler emits the code that is necessary to record a particular execution sequence. A new compiler flag *–at* was provided for achieving this. The program should be compiled with the *–at* flag so that the compiler emits the profiling code. The function *__bb_new_trace_func* is executed every time execution enters a basic block. In this function the value of basic block count is incremented for the appropriate basic block. The value of the basic block label and basic block count are stored in a global variable in the address space of the process. When an event of interest occurs in the thread library, the event specific information is printed along with the values of the current basic block label and count.

Signals can be received either synchronously or asynchronously. Signals that are received synchronously are those that were received as a result of events taking place inside the process. Signals that were received asynchronously are those that were received from outside the process. The thread library only needs to record events taking place outside the process. All signals which are delivered from outside the process are recorded in the thread library.

6.6 User interface extensions

The following are the new commands added to GDB.

Info mutex: To obtain information about the existing mutexes.

Info condition variables: To obtain information about the existing condition variables.

Runtcl: This command allows us to run a Tcl script from GDB command line in Insight [20]. This command when used from Insight can be used for creating graphical user interfaces. Using the gdb '*commands*' command Tcl scripts can be attached to breakpoints, by specifying *runtcl* as the command to be executed when the breakpoint is hit.

The following commands were added to libgdb:

Gdb_get_thread_list: To obtain the list of current threads.

Gdb_get_thread_info: To obtain information about a particular thread.

These commands were used from Tcl scripts to create Tk widgets for displaying debugging information about the threads.

6.7 Attaching Tcl scripts to breakpoints

The ability to attach Tcl scripts to breakpoints has been provided. In addition, Tcl procedures for accessing variables from the process address space from within the Tcl scripts have been implemented. These procedures allow a user to write custom debugging scripts that can be used to debug complex debugging scenarios.

6.8 Thread debugging using the enhanced debugger

6.8.1 Thread debugging commands

The following facilities can be used for debugging multithread programs [8]:

- automatic notification of new threads

- `thread *threadno*', a command to switch among threads
- `info threads', a command to inquire about existing threads
- `thread apply [*threadno*] [*all*] *args*', a command to apply a command to a list of threads
- thread-specific breakpoints
- 'info mutex', a command to inquire about existing mutexes
- 'info condition variables', a command to inquire about existing condition variables.
- 'call setschedulerlock( )' , a command which calls the scheduler locking function in the thread library. This command prevents the scheduling signal SIGPROF from being generated.
- 'call switchtothread(*threadid*)' a command which calls the function in the thread library to switch context to a desired thread.

The GDB thread debugging facility allows observing all threads while the program runs. When GDB takes control, one thread in particular is always the focus of debugging. This thread is called the *current thread*. Debugging commands show program information from the perspective of the current thread.

Whenever a new thread is detected, GDB displays the identification for the thread with a message of the form `[New thread *threadid*]'.

For debugging purposes, GDB associates its own thread number always a single integer with each thread in a program.

- info threads

  Display a summary of all threads currently in the program. GDB displays for each thread (in this order):

  1. the thread number assigned by GDB
  2. the thread identifier of the thread
  3. the current stack frame summary for that thread

  An asterisk `*' to the left of the GDB thread number indicates the current thread. For example,

```
(gdb) info threads
3 Thread 3074(active) _thread_start (funptr=134515286)at
thread.c:1099
* 2 Thread 2049 (running)  f (message=0x8048940) at
testmultiplethreads.c:103
1 Thread 1024 (active) x4002fde8 in findnexttorun () at
scheduler.c:273
```

- thread *threadno*

  Make thread number *threadno* the current thread. The command argument *threadno* is the internal GDB thread number, as shown in the first field of the `info threads' display. GDB responds by displaying the system identifier of the selected thread , and its current stack frame summary:

- thread apply [*threadno*] [*all*] *args*

  The thread apply command permits us to apply a command to one or more threads. *Threadno* argument is used to specify the numbers of the threads that should be affected. *threadno* is the internal GDB thread number, as shown in the first field of the `info threads' display. To apply a command to all threads, "thread apply all *args* " command should be used.

- info mutex

  The info mutex command provides information about existing mutexes. For each mutex currently existing in the program, the following information is displayed:

  1. Name of the mutex
  2. Mutex state
  3. Mutex type
  4. Owner of the mutex
  5. List of threads waiting on the mutex

- info conditionvariable

The info conditionvariable command provides information about existing condition variables. For each condition variable currently existing in the program, the following information is displayed:

1. Name of the condition variable
2. List of threads waiting on the condition variable

The information obtained from the info mutex and info conditionvariables commands can be used to construct a *waitfor* graph [4] that can be used for deadlock detection.

6.8.2 Setting breakpoints in multithreaded programs

When a program has multiple threads, breakpoints can be set on all threads, or on a particular thread.

```
break linespec thread threadno
break linespec thread threadno if ...
```

linespec specifies source lines where the breakpoint needs to be set. The qualifier `thread *threadno*' with a breakpoint command should be used to specify that GDB should stop the program when a particular thread reaches this breakpoint. *threadno* is one of the numeric thread identifiers assigned by GDB, shown in the first column of the `info threads' display. If `thread *threadno*' is not specified when setting a breakpoint, the breakpoint applies to *all* threads of the program. The thread qualifier on conditional breakpoints can be used as well.

```
(gdb) break test.c:13 thread 28 if index > lim
```

6.8.3 Debugging signals

GDB has the ability to detect any occurrence of a signal in a program. The user can specify in advance what action should be taken for each kind of signal. Normally,

GDB is set up to ignore non-erroneous signals like SIGALRM, but whenever an error signal occurs the program is stopped immediately.

- info signals

  This command prints a table of all the kinds of signals and how GDB has been configured to handle each one.

- handle *signal keywords*...

  This command is used to change the way GDB handles signal *signal*. The *keywords* specify what change should be made.

The keywords that are allowed by the handle command are stop, nostop(to either stop or not stop a program when a signal happens); print, noprint (to either print or not print a message when a signal happens); pass, nopass (to either pass or not pass a signal to the program when the signal happens). Thus it is possible to control from GDB whether or not a program sees a signal when it is delivered.

- signal *signal*

  This command is used to resume execution where the program stopped, resuming the program with the signal *signal*. If *signal* is zero, execution is continued without giving a signal to the program.

6.9 Replaying a recorded scenario

The recorded scenario has entries in the following format:

PC BasicBlockLabel BasicBlockCount Signal Currenthread

PC is the program counter location where the signal was delivered to the program. *Basicblocklabel* is the label of the basic block in which the PC value comes. *Basic blockcount* is the number of times the basic block was entered before this signal was delivered. *Signal* is the signal that was delivered to the process.

The replaying mechanism works as follows:

The context switches take place due to the signal SIGPROF. To prevent context switches taking place due to SIGPROF this signal should be blocked. Also all other signals should be blocked from the process, so that they can be delivered at the desired place from the debugger.

In the debugger the following commands are used for replay:

1.The signal SIGPROF is blocked for the process, so that context switches donot take place due to timer expiration. This is done using the following command:

Handle SIGPROF *nopass*

Any other signal can be blocked in a similar way.

2. A conditional breakpoint is set at the address PC.

Break PC *if* _thread_handles[currenthread].block == BasicBlockLabel &&
_thread_handles[currenthread].count == BasicBlockCount

Here *_thread_handles* is a data structure in the thread library which contains information about each thread. *Block* and *count* are members of this data structure

which are used to store the values of basic block label and basic block count respectively.

3.When the breakpoint is hit, the signal should be delivered to the process. This is done using the following command:

signal <Signal>

4.Steps 1-3 are repeated for the rest of the entries of the logfile containing the recorded events.

6.10 Obtaining a specified thread interleaving

When testing thread interleaving scenarios the user might sometimes want to specify a particular thread interleaving and see the behavior of the program under such circumstances. We now illustrate how the desired thread interleaving can be obtained.

1. First the scheduler is locked to prevent context switches from taking place at arbitrary places due to expiration of scheduling timer. This is done by using the following command in the debugger:

   call setschedulerlock(ON)

   The *setschedulerlock( )* function is a function in the thread library. The *call* command in GDB allows us to call this function from GDB.

2. Then breakpoint is set at the location where we wish context switch to take place. This is done using the following command:

   break <line#> *if* <condition>or

break *<address> *if* <condition>

3. When this breakpoint is hit, a breakpoint is set again at the location where we wish the next context switch to take place.
4. Context switch is done to the desired thread using the following command:

   call switchtothread(ThreadID)

Chapter 7

Testing

7.1 Correctness testing of the debugger

A module-by-module testing was done for each target specific thread operation that was implemented. All the thread related commands were tested for programs written using the Bthreads library.

7.2 Testing the replay of concurrency scenarios

1. The test program is designed as follows:

In the main program, two threads: thread1 and thread2 are created. There are three threads currently running. The main thread, thread1 and thread2. The scheduler schedules these threads. All the three threads perform CPU intensive operations.

The main program has the following structure:

```
Create thread1
Create thread2
for(I=0;I<50000;I++)
printf("Mainthread");
```

Thread1 and Thread2 have the following structure:

```
for(I=0;I<30000;I++)
printf("Thread");
```

Context switches take place when the main thread and each of the threads are looping in the for loop. These context switches are caused by SIGPROF, which is generated due to the expiry of the scheduling timer. During the initial run recording of the context switching events is done. Replay is done from inside gdb. The handle for the signal SIGPROF is set to no pass. Using the log conditional breakpoints are set. The breakpoints are set at the PC locations when context switches previously took place due to SIGPROF. Using this mechanism the exact execution sequence could be reproduced.

2. A test program is designed which has the following programming constructs- if, while, for and goto. When the program is run context switches are forced to take place while executing various constructs. Recording is done in each run. Then these scenarios could be exactly replayed using the debugger.

3. The third test case involves implementing the dining philosophers problem and replaying deadlock situations that occur. A solution to the dining philosopher problem is implemented. The algorithm works as follows:

A mutex variable is associated with each fork. Every philosopher thinks for a random period. Then he picks up the right fork and then the left fork. He then eats for a random period of time after which he puts down both the forks. A philosopher can pick up a fork only after acquiring the mutex associated with the fork. The mutex is released when the philosopher puts the fork down. Suppose there are n philosophers. Each philosopher is associated with a thread. If context switches take place in each thread after the philosopher (associated with the thread) picks up the fork, then each philosopher will be having a fork in his right hand and a deadlock will occur, because each philosopher will be waiting for the one on his left to put the fork down. Such a deadlock occurs only if context switches take place exactly after each philosopher acquires the right fork and before he lifts the left fork. Hence, such a sequence may not repeat across executions. This program was run for varying number of philosophers and the execution pattern recorded. Those execution sequences in which deadlocks occur were then replayed from the debugger.

The deadlock situation was also deliberately created by forcing context switches after each philosopher acquires the right fork. This was done using the procedure illustrated in section 6.10. The deadlock situation was successfully created using this procedure. Thus, the procedure for obtaining specified thread leaving was tested.

Chapter 8

Conclusions and Future Work

Debugging is one of the most important phases in software development. Yet, it is not sufficiently formalized to make effective software development possible. Without sufficient support for debugging tools, the gap between what programmers can write from what they can test has been increasing. In this work, we presented the problems in concurrent software debugging and discussed an architecture that allows us to build concurrent software that allows us to replay concurrency scenarios. A user level thread library was built and GDB support was provided to debug programs written using this thread library. A mechanism has been provided for replaying concurrency scenarios from within the debugger.

There is scope for further work in the following areas:

1. Running User-mode Linux [17] on Bthreads to give reproducible operating system activities: By porting User-mode Linux to Bthreads, it is possible to obtain reproducible executions of the operating system.
2. Reproducible execution of Nachos [18].
3. Sophisticated visualization tools, by inserting Tcl scripts at breakpoints: Visualization tools for basic visualization of thread debugging have been

provided. More sophisticated tools for visualization can be built using Tcl/Tk scripts.

4. Reproducing I/O: There are several classes of multithreaded programs where reproducible behavior is desirable. The most primitive class has only context switching events. The next level has context switching events and signals. In this thesis, these two classes of programs were considered for obtaining reproducible behavior. The next class of multithreaded programs which need to be reproduced are those which have the events-context switches, signals and I/O. Further work can be done for reproducing multithreaded programs in this class.
5. Extending the concept of reproducible execution to distributed systems.
6. Porting the Java Virtual Machine to Bthreads to enable reproducible execution of Java programs.

Bibliography

[1] Andrew S. Tannenbaum and Maarten van Steen, *Distributed Systems: Principles and Paradigms*, Prentice Hall, 2002.

[2] Daniel Schulz and Frank Mueller, *A thread-aware debugger with an open interface*, ACM SIGSOFT Software Engineering Notes, Proceedings of the International Symposium on Software Testing and Analysis, August 2000.

[3] Jonathan T.Giffin, George S. Kola, *Linux process control via the file system*, Computer Science Dept, University of Wisconsin, Madison.

[4] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne, *Operating System Concepts*, Chapter 8: Deadlocks, John Wiley and Sons Inc, June 2001.

[5] Andrew P. Tolmach and Andrew W. Appel, *Debuggable concurrency extensions for Standard ML*. Technical Report CS-TR-352-91, Princeton University, Dept. of Computer Science, 1991.

[6] Peter A. Buhr, Martin Karsten and Jun Shih, *KDB: A multithreaded debugger for multithreaded applications*, Proceedings of SPDT'96: SIGMETRICS Symposium on Parallel and Distributed Tools, pages 80-87, Philadelphia, Pennsylvania, U.S.A., May 1996. ACM Press.

[7] Sudhir Halbhavi, *Thread Debugger Implementation and Integration with the SmartGDB Debugging Paradigm*, Master's thesis, University of Kansas, 1996.

[8] Stallman R.M. and Pesch R.H., *Debugging with GDB*, Free Software Foundation, 675 Massachussetts Avenue, Cambridge, MA 02139 USA, 1995.

[9] Sunil Penumarthy, *Design and Implementation of a User-Level Thread Library for Testing and Reproducing Concurrency Scenarios*, Master's thesis, University of Kansas, 2002.

[10] Rajukumar Girimaji, *Reactor: A software pattern for building, simulating and debugging distributed and concurrent systems*, Master's thesis, University of Kansas, 2003.

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

[12] John K. Ousterhout, *Tcl and Tk Toolkit*, Addison-Wesley Publishing Company Inc, 1994.

[13] SmartGDB. URL http://hegel.ittc.ku.edu/projects/smartgdb/

[14] *Using the GNU Compiler Collection (GCC)*, Free Software Foundation, 59 Temple Place - Suite 330, Boston, MA 02111-1307 USA.

[15] Linux Reference Manual, Section 2, ptrace.

[16] F.Mueller. *A library implementation of POSIX threads under UNIX.*, Proceedings of the USENIX conference, pages 29-41, Jan 1993.

[17] User-mode Linux, URL: http://usermodelinux.org/

[18] Nachos, URL: http://www.cs.duke.edu/~narten/110/nachos/main/main.html

[19] ODB , URL: http://java.sun.com/features/2002/08/omnidebug.html

[20] Insight, URL: http://sources.redhat.com/insight/