



A Thread Debugger for Testing and Reproducing Concurrency Scenarios

Satyavathi Malladi

Masters Thesis Defense

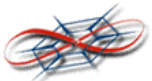
Date: 15 Jan 2003

Committee:

Dr. Jerry James (Chair)

Dr. Douglas Niehaus

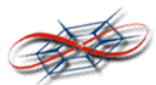
Dr. Joseph Evans





Outline of the presentation

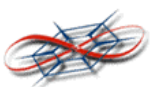
- Overview of the problem
- Our solution
- Background
- Design and implementation of the thread debugger
- Recording and replaying execution
- Testing
- Related Work
- Conclusions and future work





Overview of the problem

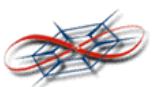
- Definitive testing of concurrency scenarios
- Execution sequence of multithreaded programs inherently nondeterministic
 - Program behavior in different executions need not be the same
 - Context switches, event ordering and synchronization are main reasons
- Mismatch between programming model and debugging model
- Capability to record, analyze and playback particular execution sequences
- Software pattern needed over which definitive replay can be done





Our Solution

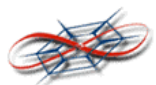
- Software pattern BERT used for building concurrent software
- BERT is based on the Reactor pattern
- Reactor provides event de-multiplexing framework
 - Events captured by the Reactor
 - Event can be I/O completion, timer expiration or signal
- Concurrent software built using the BERT uses a user level thread library(Bthreads)
- Bthreads uses the many-to-one model





Solution (Contd...)

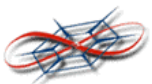
- Bthreads library provides capabilities for user specified context switching and locking the scheduler
- GNU debugger (GDB) extended to provide the capability to debug software written using the Bthreads library
- Capability to record events of interest provided within the thread library
- Capability to replay an execution sequence illustrated in the debugger





Background

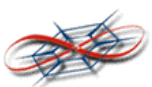
- Challenges in debugging multithreaded programs
 - Execution control interleaves between the threads
 - Threads may voluntarily suspend and resume execution or they might do it as a result of events like signals
 - Order of interleaving between threads partially determined by synchronization between the threads





Background (Contd...)

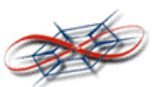
- Limitations of conventional debuggers
 - The debugger does not have sufficient information to display the state of threads and other synchronization primitives.
 - It might be necessary to forcibly suspend the execution of a thread and resume some other thread.
 - This is not possible in case of many thread libraries.
 - There is no information about threads waiting on synchronization objects.





Process Control

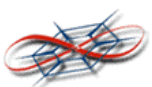
- Ability to inspect a running process and alter its execution
- Key ability needed in debuggers
 - Debuggers need to have access to the address space of the process
- Tracing process - debugging process
- Inferior process - process being debugged





Process Control (Contd...)

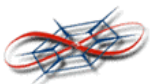
- Operating system has to provide the means to achieve process control
- Many Unix based systems provide two mechanisms for process control:
 - proc file system
 - ptrace system call





Process Control (Contd...)

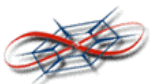
- proc file system
 - Each running process represented by a collection of files
 - Tracing achieved by accessing the files representing the inferior process
 - read, write, lseek, poll, and ioctl system calls used to modify these files
 - Directives passed to the kernel when writes are performed on these virtual file system files
 - Directives indicate what action is to be applied to the inferior process





Process Control (Contd...)

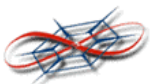
- Ptrace system call
 - Tracing process calls *ptrace* with arguments identifying the inferior process and specifying the controlling action to be applied
 - Kernel inserts the tracing process as the parent when it uses the *ptrace* system call
 - Tracing process calls the *wait* system call to receive the stopping event information
 - Upon the receipt of a signal, the *wait* system call returns
 - Based on the reason for which *wait* returned, the appropriate controlling action can be taken by the tracing process





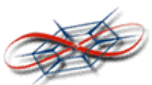
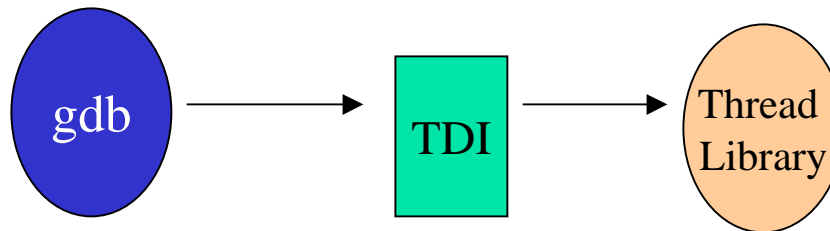
GDB Architecture

- Process control mechanism used is *ptrace* system call
- Breakpoints are inserted by swapping the instruction at the breakpoint location with trap instruction.
- Supporting different debugging targets
 - *target_ops* structure
 - Each target has its own *target_ops* structure
 - Targets stacked in strata
- GDB support for debugging threads
 - GDB provides a generic framework for debugging threads of any thread library.
 - All information generic to any thread library is stored in the data structure *thread_info*.



GDB Architecture (Contd...)

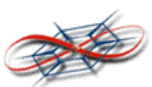
- Operations that are target thread library specific are implemented separately for each thread library
- The debugger should be aware of the thread library internals to provide the necessary debugging information to the user
- The Thread Debug Interface(TDI) used by the debugger to communicate with the thread library





Implementation

- The target specific operations for the Bthreads library need to be implemented
 - Attach
 - Detach
 - Wait
 - Resume
 - Transfer memory
 - Fetching and storing registers





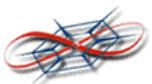
Implementation (Contd...)

- Attach

- Attach debugger to an existing process.
- *ptrace* system call used with `PTRACE_ATTACH` as the requested operation

- Detach

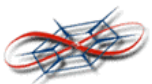
- Detaches from a process to which the debugger is already attached .
- The *ptrace* system call is used with `PTRACE_DETACH` as the requested operation.





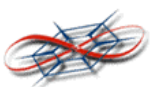
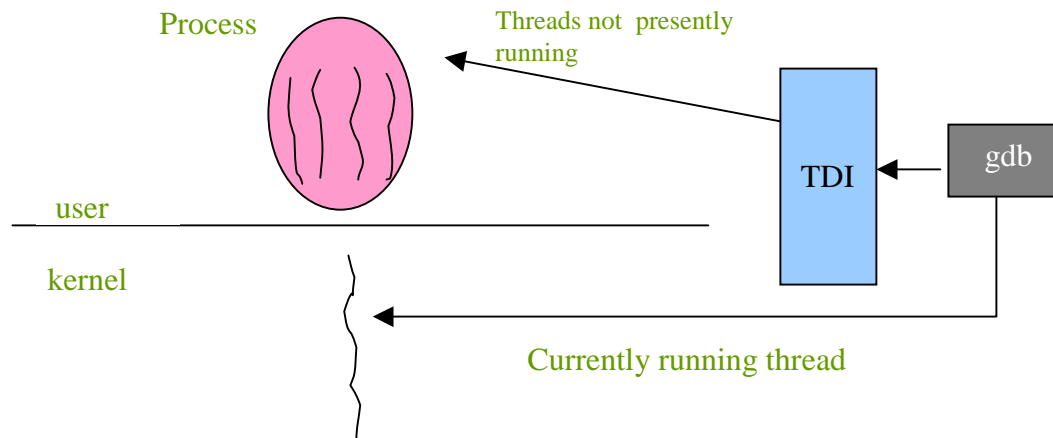
Implementation (Contd...)

- Resume
 - Resume the execution of the process.
 - Process might be resumed in either single step mode or continued
- Xfer Memory
 - Used to transfer memory between process address space and gdb address space
- Thread alive
 - TDI used to query about the state of a particular thread to determine whether it is alive.



Implementation (Contd...)

- Fetching and storing registers
 - The registers for the current running thread are obtained from the kernel. The registers for all other threads are obtained from the TDI.

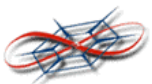




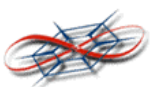
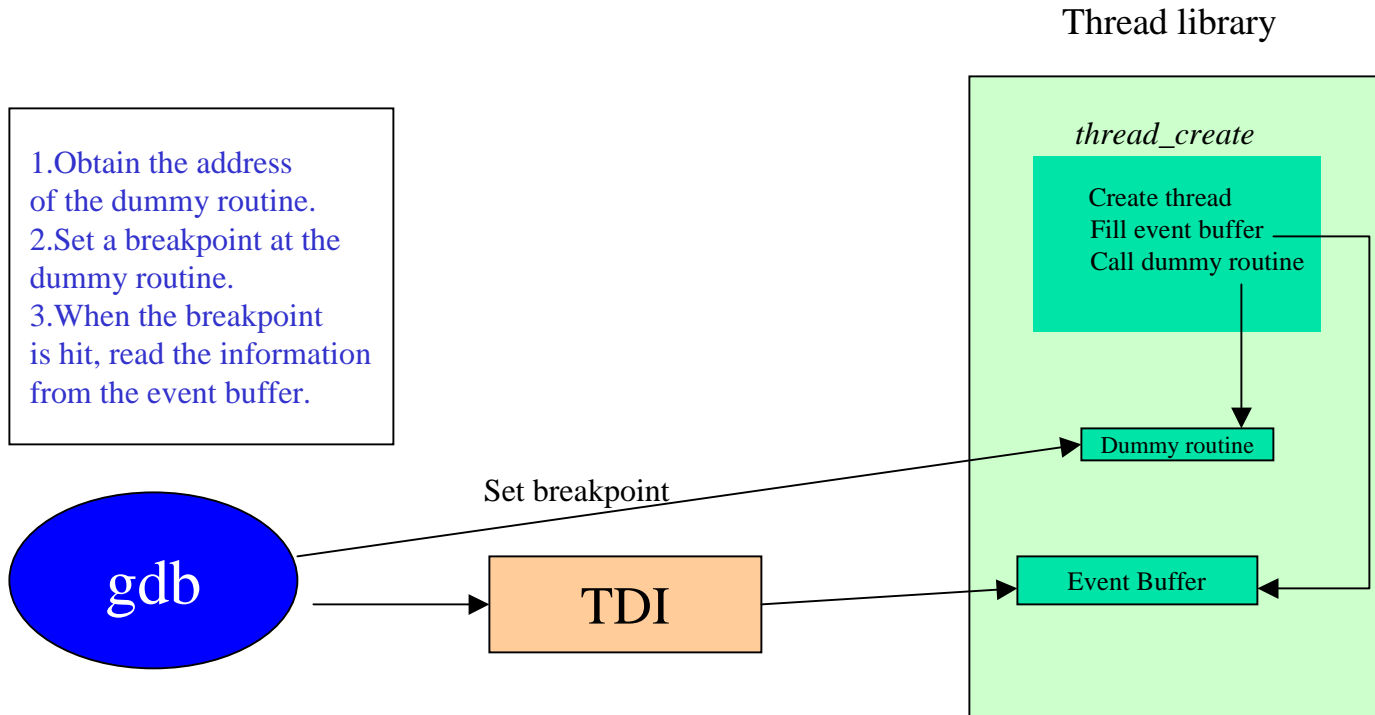
Implementation (Contd...)

- Wait

- Wait for any threads to stop due to the delivery of a signal.
 - Depending on the reason for which the thread stopped, the appropriate action is taken by the debugger.
- There are several events in the thread library that the user is interested in.
 - Thread creation event
 - Thread death event



1. Obtain the address of the dummy routine.
2. Set a breakpoint at the dummy routine.
3. When the breakpoint is hit, read the information from the event buffer.





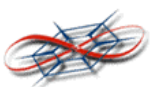
Implementation (Contd...)

- **Mutex info**

- Obtains information about the existing mutexes
- TDI used to obtain this information
 - Name
 - Mutex state
 - Mutex type
 - Owner
 - List of threads waiting on mutex

- **Condition variable info**

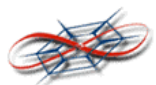
- Obtains information about the existing condition variables
- TDI is used to obtain this information
 - Name
 - List of threads waiting on the condition variable





Testing and reproducing concurrency scenarios

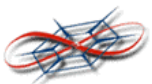
- Testing a hypothesis using specified thread interleaving
- Recording and replaying an execution sequence





Testing a particular hypothesis

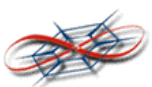
- The user might want to test the program by specifying context switches at desired locations.
- Procedure for testing a particular hypothesis:
 - Set a breakpoint at the place where a context switch is desired.
 - Switch to the desired thread using the `switchtothread()` function which is provided by the thread library.
 - Call `switchtothread(threadid)`





Recording Events

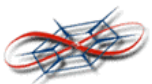
- Causes of non-determinism
 - Context switches taking place due to the expiration of the scheduling timer
 - External signals
- Capability to record context switches and signals provided within the thread library.
- Information needed while recording context switches and signals:
 - Program Counter (PC)
 - Count
 - Signal





Recording Events (Contd...)

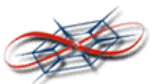
- A *basic block* is a group of hardware instructions which are always executed together.
- The basic block information is generated by the *gcc* compiler when a program is compiled with the *-ax* flag.
- The information about the basic block and count is stored in the address space of the process.





Replay using the debugger

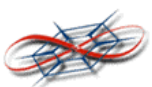
- The signal SIGPROF is responsible for scheduling to take place. Hence this signal is prevented from being delivered to the process.
 - handle SIGPROF nopass
- Condition breakpoints are set in the following way:
 - Break *pc* if `block == blk && count == cnt`
- When the breakpoint is hit, the scheduling needs to take place.
 - Signal SIGPROF
- This causes SIGPROF to be delivered to the process.





Replay (Contd...)

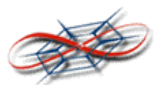
- In case the event is a signal event, then the appropriate signal is delivered using the signal command.
- This is repeated until all the events in the recorded log are exhausted.




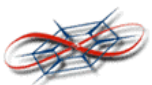
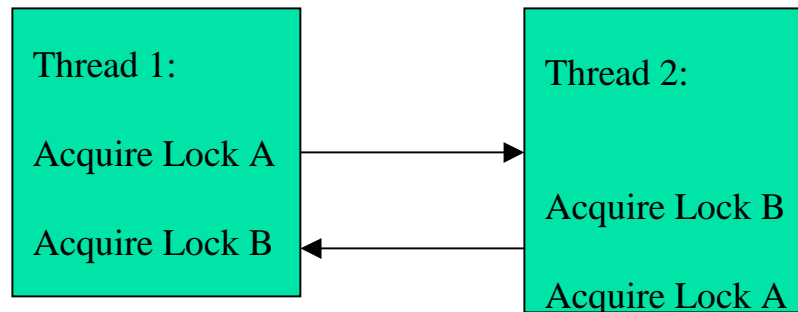


User Interface Extensions

- The following commands were added to GDB:
 - info mutex: This command is used to obtain all the attributes associated with the mutex variables in the program.
 - info conditionvariable: This command is used to obtain all the attributes associated with the condition variables in the program.
 - runtcl <tcl-script> : This command is used to run a tcl script from the gdb command line.
 - gdb 'commands' command was used to attach tcl scripts to breakpoints.
 - Scripts written for visualization purposes.
 - Scripts written for producing user specified interleaving.
 - Scripts written for automatic replay of recording scenarios.



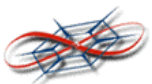
- 
- Two threads try to acquire two locks in different order





Testing

- Correctness testing of debugger
- Correctness testing of recording and replaying execution sequences
 - Case1:
 - 5 threads are created and in each thread there is a while loop that iterates for 20,000 times.
 - The context switch events are recorded. About 50 context switching events took place in different runs.
 - All the context switching events could be exactly replayed from the debugger.





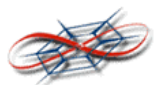
Testing (Contd...)

- Case 2: Dining philosophers problem

Algorithm:

- Each philosopher thinks, then picks up the right fork and then left fork. Eats and places back both the forks.

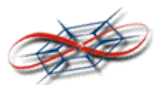
The program is run several times for different number of philosophers. When deadlocks occur they are replayed from the debugger.





Conclusions

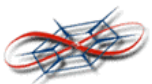
- A thread debugger is built which can debug programs using the Bthreads library.
- A recording mechanism is provided in the thread library for the context switching events and signals.
- The mechanism to replay context switching events and signals is provided.
- Using the debugging tool along with the replay tool, more definitive testing of concurrent software under development is made possible.





Related Work

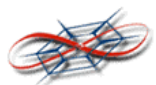
- SmartGDB
- KDB
- Mach Debugger
- ODB
- DejaVu
- ML thread debugger





Future Work

- Reproducing I/O
- Sophisticated visualization tools
- Reproducible execution of Nachos
- Running User-mode Linux on Bthreads to give reproducible operating system activities
- Extending the concept of reproducibility to distributed systems





Acknowledgements

