Design and Implementation of a User Level Thread Library for Testing and Reproducing Concurrency Scenarios

Sreenivas Sunil Penumarthy
Masters Thesis Defense
December 17, 2002

Committee:
Dr. Jerry James (Chair)
Dr. Gary Minden
Dr. Arvin Agah
Outline of Presentation

• Introduction
• Motivation
• Solution
• BERT
• Design and Implementation of BThreads
• Debugging features of BThreads
• Testing and Results
• Conclusions
• Future work
Introduction

• High performance computing (HPC) one of the key requirements for scientific, web based and military applications — parallel computing one solution
• An example: web server and web browser
• Web server and web browser benefit from parallelism
• Web browser displaying and fetching HTML can be done in parallel
• Web servers need parallelism to attain maximum throughput: number of client requests processed/unit time
• Multithreading is a popular parallel model
Introduction

- Thread implementation models

<table>
<thead>
<tr>
<th>Many threads per process (M:1)</th>
<th>One thread per one kernel level thread (1:1)</th>
<th>Many threads many kernel level threads (M:N)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Easiest to debug</td>
<td>Hardest to debug</td>
<td>Intermediate in difficulty</td>
</tr>
<tr>
<td>I/O blocks all threads</td>
<td>I/O blocks one thread</td>
<td>I/O blocks some threads</td>
</tr>
</tbody>
</table>

User threads

Kernel thread

Many-to-one model  One-to-one model  Many-to-many model
Motivation

- Debugging multi-threaded programs is difficult
- The execution of program can differ from one run to another
- Multi-threaded programs don’t execute deterministically (race conditions, deadlocks)
- Sources of nondeterminism: Context switching, completion of I/O, signals, scheduler decisions
- Execution model and the debugging model are mismatched; insufficient debugging control
Solution

- Without kernel modifications, don’t have sufficient control with the one-to-one and many-to-many models.
- With the many-to-one model, we have potential for sufficient control, but current libraries don’t provide it.
- With an event-driven framework, we can provide this control.
- Such an event-driven framework has been developed at ITTC: BERT.
BERT

- BERT interface is built using REACTOR, which provides an event-demultiplexing framework
- An event is associated with a handler that has non-blocking methods that are called upon detection of event
- An event can be: I/O completion, timer expiration, signal
- All these events are captured at REACTOR
- Hence REACTOR is a single point of control
BERT

- Capture all the asynchronous events at deterministic points in REACTOR
- The information can be recorded here and later used for replay
- With many-to-one model, as scheduler runs in user level, it is possible to test different concurrency scenarios by forcing context switches from debugger
- BThreads library is based on many-to-one model, built on top of BERT interface
Design and Implementation of BThreads

- Thread creation and termination
- Thread scheduling
- Thread synchronization
- I/O
- Signals
- Thread Safety
- Pushing function call onto thread stack
- Other features implemented
- Limitations Of BThreads
Thread Creation and Termination

- Two interfaces are available for creation of user space threads:
  - Ucontext API
  - JMPBUF based functions
- Ucontext API is used in the BThreads library
- Thread creation and termination have been implemented in BThreads according to POSIX requirement
- Termination Queue holds terminated threads in detached state
  - Memory resources of threads in this queue are deallocated (reaping)
Thread Scheduling

- Default scheduling in BThreads is Round Robin
- Timers are registered with Reactor and Reactor dispatches timer to BThreads library when it expires

FIFO scheduling can be realized by turning off timers
Thread synchronization

• If a thread blocks on a synchronization variable, process as a whole may block
• Following synchronization primitives required in a POSIX compliant thread library have been provided:
  • Mutexes
  • Condition Variables
• In addition, waitlocks and spinlocks were implemented
• Used wait locks for protecting critical sections of mutex and condition variable functions
I/O

• I/O blocking is a major issue in many-to-one thread library
• If a thread blocks, process as a whole blocks
• Before entering WAITING state, register the event handler object with the reactor
• EventHandler object: `Handle_input`, `Handle_output` methods
• Event handler methods invoked upon detection of events
  – For I/O this means that thread will be put in READY state when I/O can be done without blocking
• When to invoke Reactor to check for I/O completion?
  • Whenever scheduler is invoked
Wrapper Functions in Library (I/O)

Open() or Socket() wrapper system call

- Make actual system call
- Insert the file descriptor (fd) into the global list of currently active fd’s
- Return newly created file descriptor

Read() or Write() wrapper system call

- Call readwrapper() or writewrapper() function
- Make actual System Call
- Do error checking and cleanup if necessary
- Return number of bytes read or written

Is fd valid?

- Yes
  - Create iohandler object for current thread if needed
  - Register iohandler object with Reactor
  - Enter WAITING state and call scheduler
  - Deregister iohandler object with Reactor
  - Return from function
- No

Readwrapper() or Writewrapper()
Wrapper Functions in Library (I/O)

Close () system call
wrapper

Remove the file descriptor from the global list

Error ?

Yes

Call the closewrapper () function on all the threads

Make actual close() system call

Return

No

Closewrapper () function call

Remove the iohandler from the thread’s TCB

Deregister the iohandler completely with the Reactor

Deallocate memory for iohandler object

Return from function

(Only visible in library level)
Signals

- Delivery and masking of signals must be thread-specific
- Signal handlers are shared among all the threads
- Classification (depending on how signals are generated):

```
Signals

Synchronous
  - raise(sig)
  - Generated due to instruction

Asynchronous
  - thread_kill(tid,sig)
  - Externally. Kill or Ctrl+C
```

Ctrl+C
Signals

• POSIX requirements for delivery of signals:
  • Synchronous signal - thread that generated the signal
  • Asynchronous fatal signals - all the threads running in the process must be terminated (Default behavior with BThreads)
  • Asynchronous non-fatal signals
    – If generated due to `thread_kill` - only a specific thread
    – If generated due to `kill/TTY` - Any one thread that doesn’t block the signal
Signals

• Signal delivered in BThreads when
  • Signal mask of thread is changed `thread_sigmask`
  • When a new thread is scheduled in the scheduler and it starts running

• How asynchronous signals are delivered
  • Signals due to `thread_kill` generated by inserting `raise_threads`
  • Signals that are generated externally will be delivered automatically
Thread Safety

- Thread safe: Multiple threads can call methods simultaneously - An issue in preemptive library
- User-Level data consistency: Mutexes, Condition variables.
- How to ensure Library-Level data Consistency?
- Solution: Two ways to ensure consistency
  - Consistency using atomicity (Disable and re-enable signals)
    - Ready Queue (Accessed in thread_create, scheduler)
    - Reactor Queue (Accessed in Reactor and scheduler)
  - Consistency using mutual exclusion (Using waitlocks)
    - Termination Queue (Queue having all the terminated threads), Thread Control Block
Pushing function call onto thread stack

- This mechanism is used when
  - Delivering signals due to thread_kill
  - Calling scheduler due to generation of SIGPROF
  - Implementing asynchronous cancellation
- To allow insertion of an arbitrary function on an execution stack, \texttt{esp, eip} registers need to be modified
- Current implementation is for x86 architecture
- To support insertion of C function with arbitrary signature, an assembly wrapper function is needed
Pushing function call onto thread stack

1. ESP before inserting function call
2. ESP after inserting function call
3. ESP just before calling C function
4. ESP at return from assembly function

- Original stack
- Original EIP
- Register Information
- Flags Information
- C Function to call
- Arguments to C function
- # bytes of arguments

ESP – Stack Pointer

Enter the stackwrap assembly wrapper

Save # bytes of arguments in EBX register

Change the ESP to point to the bottom of the argument list to the C function.

Call the C function.

Pop function arguments

Restore the GPR and FLAGS

Return from the assembly wrapper. This resumes original execution.
Other Features Implemented

• Functions implemented according to standard POSIX requirements:
• Thread cancellation
• Cleanup handling
• Thread specific data
• Thread once functions
Limitations of BThreads

- Priority based scheduling
- Timed variants of condition variables and mutexes
  - `thread_cond_timedwait`
  - `thread_mutex_timedlock` (not required by POSIX).
  - These return ETIMEDOUT when timeout occurs
- Thread barrier functions (not required by POSIX)
- Thread Read/Write (R/W) locks
- Process shared or process private mutexes, R/W locks, condition variables
- Concurrency level (Only for many-to-many thread models)
Debugging features of BThreads

- Thread Debug Interface
- Testing concurrency scenarios
- Recording concurrency scenarios
Thread Debug Interface (TDI)

- **GDB** uses TDI to get information about thread library
- TDI provides ability to access and modify data structures in the inferior process
- Event enabling and reporting
- Examining thread related information
- Invoke call back functions over a set of threads that meet some criterion
- List of mutexes and condition variables
- Get and set register information
Testing Concurrency Scenarios

- User can form his/her own concurrency scenarios
- BThreads library provides ability to an arbitrary thread using `switch_to_thread` function
- This can be used by GDB debugger to switch to any thread

```
Thread1:
  thread_mutex_lock(A)
  thread_mutex_lock(B)
  thread_mutex_unlock(A)
  thread_mutex_unlock(B)

Thread2:
  thread_mutex_lock(B)
  thread_mutex_lock(A)
  thread_mutex_unlock(B)
  thread_mutex_unlock(A)
```
Recording Concurrency Scenarios

- In normal circumstances, recording is done when program runs without any intervention of debugger.
- Record only information, which can disrupt sequential flow:
  - Scheduling (due to SIGPROF signal)
  - Signals
  - I/O completion
Testing and Results

• Correctness Testing
• Performance Testing
• Testing different concurrency scenarios
• Recording and reproducing different concurrency scenarios
Correctness Testing

- White Box Testing for BThreads library
- POSIX Compliance testing (Linux Threads):
  - Basic thread creation and destruction
  - Classic Producer-Consumer problem (Condition variables & mutexes)
  - Multi-thread searching (mutexes, cancellation and cleanup handling)
  - Different threads accumulating their strings concurrently (TSD, thread_once functions)
  - Concurrent multiplication of NxN matrices
Performance Testing

- A multi-threaded FTP server based on Linux Threads was taken
- FTP server based on BThreads was built from it

For 95% confidence, BThreads confidence interval (worst case) 0.40 sec, PThreads 0.5 sec

File Sizes: 14 MB, 52 MB
Performance Testing

Performance comparison of BThreads with PThreads using a multi-threaded ftp server for file size of 14.47 MB

<table>
<thead>
<tr>
<th>No of Client Connections</th>
<th>Average FTT BThreads</th>
<th>Average FTT Pthreads</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>2.07917</td>
<td>2.13417</td>
</tr>
<tr>
<td>6</td>
<td>3.71625</td>
<td>3.78375</td>
</tr>
<tr>
<td>9</td>
<td>5.24722</td>
<td>5.27722</td>
</tr>
</tbody>
</table>

FTT: File Transfer Time
Performance Testing

Performance comparison of BThreads with PThreads using a multi-threaded ftp server for file size of 52.13MB

<table>
<thead>
<tr>
<th>Number of client connections</th>
<th>BThreads</th>
<th>PThreads</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>8.483</td>
<td>8.704</td>
</tr>
<tr>
<td>6</td>
<td>16.079</td>
<td>16.567</td>
</tr>
<tr>
<td>9</td>
<td>24.022</td>
<td>24.967</td>
</tr>
</tbody>
</table>

FTT: File Transfer Time
Testing and reproducing concurrency

• Deadlock conditions: Mutual exclusion, Hold & Wait, Circular wait, no preemption

• Two test programs that had possibility of deadlocks were considered:
  • Two threads trying to acquire two locks in different order

Thread1:
  Acquire lock A
  Acquire lock B

Thread2:
  Acquire lock B
  Acquire lock A

Diagram:

- T1
  - R1
  - R2
- T2
  - R1
  - R2
Testing and reproducing concurrency

Dining Philosopher’s Problem

Dining Philosopher’s algorithm

Think for a random time

Acquire right fork

Acquire left fork

Eat for a random time
Conclusions

• Built a thread library that supports most of the features in a POSIX compliant thread library
• Built TDI to support debugging of BThreads programs
• Tested POSIX compliance of the library
• Tested the basic performance
• Provided a framework that can be used to improve debugging of multi-threaded programs
  • Tested and verified basic ability to test and reproduce different concurrency scenarios for context switching at arbitrary points
Related Work

• Another student is working for providing debugger support for BThreads and reproducing concurrency scenarios.

<table>
<thead>
<tr>
<th>Library</th>
<th>Main goals</th>
</tr>
</thead>
<tbody>
<tr>
<td>FSU Threads</td>
<td>POSIX compliance. Uses asynchronous I/O</td>
</tr>
<tr>
<td>NGPT</td>
<td>Performance, POSIX compliance</td>
</tr>
<tr>
<td>Linux Threads</td>
<td>POSIX Compliance (LINUX OS)</td>
</tr>
<tr>
<td>ACE Threads. Wrapper thread library</td>
<td>Uniform programming language C++</td>
</tr>
<tr>
<td></td>
<td>Portable thread library</td>
</tr>
<tr>
<td></td>
<td>Minimize subtle synchronization errors</td>
</tr>
</tbody>
</table>
Related Work

- BThreads library is part of the BERT infrastructure
Future Work

• Identify & wrap all system calls that can block
• Make library completely POSIX compliant
• Experiment with scheduling policies
• Port implementation to other architectures: Solaris, Irix
• “Dynamic linker tricks” to debug other thread library programs
• Transition an event-driven application to concurrent application
Questions