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

by

PENUMARTHY SREENIVAS SUNIL

B.E., Electronics and Communication Engineering

MVSR Engineering College, 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.Arvin Agah

____

Dr.Gary Minden

Date Submitted: ____________

Dedicated to
My parents and Sisters.

Acknowledgements

I would like to take this opportunity to thank my advisor, Dr. Jerry James, for his constant guidance, motivation and support. I have learnt some key concepts that make up a good computer scientist while working under him: how a problem should be approached, how should it be analyzed and designed, how to implement and test the design and how to think in a way that makes a *difference*. I appreciate his way of dealing, giving utmost freedom coupled with timely advice that put me on the correct path whenever the situation demanded it. Thanks a lot Dr.James! I take this opportunity to thank Dr. Gary Minden and Dr. Arvin Agah for giving me a chance to work in the ACE Project that laid a foundation for my software engineering concepts.

I would like to express my thanks to my parents and sisters who have been a constant source of motivation for me always. I thank them for encouraging me to take up graduate studies and pursue higher education and for helping me in all ways.

I would like to thank my friend Guru, for helping me in all ways and Arun for being a good roommate providing me with a very good ambience at home.

I would like to thank Dr. Prasad Gogineni and Dr. Pannirselvam Kanagaratnam for hiring me as GRA during the 2002 year.

I would like to thank Raj and Satya, with whom I worked on the KUDOS and BERT projects. I had a *wonderful* experience working with them.

I would like to thank all my friends in Lawrence, with whom I had great time.

Last but not the least, I would like to thank GOD without whom it would have been virtually impossible for me to carry out this project and any goal in life.

Table Of Contents

LIST OF FIGURES

LIST OF TABLES

Abstract

Multi-threading is a widely used mechanism for realizing parallelism. It provides the capability for parallel execution of tasks, there by improving overall application performance. For example, web servers and web browsers are typically multi-threaded. However, multi-threading brings with it the tough problem of debugging. Multi-threaded programs are difficult to debug due to the possibility of random interleaving of instructions from different threads of execution. There can be race conditions and synchronization problems like deadlocks that may occur only for a particular interleaving order of threads. To provide improved debugging abilities, a thread library has to be built that is based on the many-one threading model; i.e. many threads running on top of a single process. This provides the capability of controlling thread execution sequences, since the scheduler runs at user level. This thread library has to be made an integral part of a framework that can capture all events that affect concurrency. Such events include I/O system calls, signals and timers. Such a framework is an event-driven framework that acts as the controlling point for all events that affect concurrency since all these events can be captured at this point. It is then possible to record all these events at these points so that they can be used later for replaying the execution. This thesis aims at building a POSIX compliant multi-threading library that provides the ability to test and reproduce multi-threaded program executions using the Reactor, an object oriented event de-multiplexing framework, as the underlying framework.

1. Introduction

In recent years, there has been a tremendous increase in the demand for computer software and hardware that can provide high performance computing. High performance computing has applications in diverse areas: scientific, web based and military to name a few. Parallel computing is an important concept for high performance computing. Parallelism is a desired feature for various distributed applications as it can lead to significant improvements in performance. Event-driven applications like web browsers and web servers are a class of distributed applications that benefit from parallelism. A web browser displays HTML pages that are fetched from the web server to the user. To hide communication latencies, browsers are built to display HTML pages even before they are completely received. HTML pages are updated as they are received from the web server. This improves response time experienced by the user, especially when web pages have big images that take some time to download. Parallelism can be built into a web browser so that fetching and displaying of HTML pages can be done in parallel. Web servers also need parallelism to attain high performance in terms of the number of client requests processed per unit time [1]. There are different mechanisms to provide parallelism to applications:

1. Parallel computing on uni-processor machines:
    a. Process.
    b. Thread.
2. Parallel computing on multi-processor machines

3. Distributed Computing

1. Parallel Computing on uni-processor machines:

Uni-processor machines exhibit pseudo-parallelism. At any *given point of time* there is *only one task* running. However, the tasks are switched between quickly and it appears that the system is running many tasks simultaneously. A task is a program in execution. It can be: [2]

a. Process:

A process is a program in execution. Every process has a Process Control Block (PCB) that contains information such as CPU register values, memory maps, open files, privileges, etc. When a process is created, its address space has to be initialized before it can start running. This includes copying the code into the text section, setting up stack frames, etc. When a process context switch occurs, the current CPU state, including the instruction pointer, stack pointer, general purpose registers and flag registers, has to be saved, the Translation Look-aside Buffer (TLB) needs to be flushed and the memory map in the Memory Management Unit (MMU) needs to be changed. All of these operations affect performance. Communication between processes is achieved using mechanisms provided by the underlying operating system. UNIX provides named pipes, message queues and shared memory segments to this end. However all of these InterProcess Communication (IPC) mechanisms involve a good amount of overhead due to kernel intervention.

b. Threads:

A thread is a sequential flow of execution. Threads are similar to processes. However threads keep the bare minimum information required for realizing parallelism. This includes CPU register information, stack information, thread state information and some other information like priority. All threads run in a single process and they share the process virtual address space. Threads can result in a performance gain when realizing parallelism compared to processes due to lower overhead in context switching and lower communication overhead as they can communicate using global data. However mechanisms for protecting concurrent access to shared memory have to be provided. This is not necessary for processes where care is taken by the operating system, i.e., one process cannot alter another process's address space. There are several multi-threading models for implementing a thread library. These models differ in how a thread is mapped to a process. Three different multi-threading implementation models are [2]:

i. Many-to-One model:

In a many-to-one model, many threads run on top of a single process. The advantage of this approach is that context switching takes place completely at user level. Switching context just involves saving and loading the stack pointer, CPU registers and signal mask. The overhead associated with thread creation, destruction and context switching is cheap due to minimal involvement of the kernel. Also since the scheduler runs at user level, scheduling algorithms can be tuned to meet application needs. However this approach has the disadvantage that the entire process blocks if any thread makes a blocking system call. No other thread can be scheduled until this thread unblocks. Also, this model doesn't take advantage of multi-processors.

ii. One-to-one model:

In a one-to-one model, every thread maps to a process. However all the processes share the virtual address space, signal handlers and open file descriptor table and hence thread creation and context switching have lower overhead when compared to processes that don't share anything. This approach provides more concurrency than the many-one model since if a thread makes a blocking call only the corresponding process is blocked. The kernel scheduler can then schedule another thread. The one-to-one model takes advantage of multiprocessor architectures since every thread is a process and hence it can run on a different processor. However this approach suffers from overhead in executing thread management operations like thread creation, destruction and switching since all these operations need kernel intervention. Windows NT, OS/2 and LINUX implement this thread model.

iii. Many-to-Many model:

The many-to-many model combines the best of the one-to-one model and the many-to-one model. Many threads are multiplexed onto a single process and there are a number of such processes that have threads running on them. The number of processes is less than or equal to the number of threads. This has the advantage of the many-to-one model as all thread management operations have small overhead due to minimal kernel intervention barring those that involve processes. Thread context switching takes place completely in user space and so this overhead is minimal. Processes call the thread scheduling function and access the global thread list to schedule a new thread. The thread list must be protected by some sort of lock or mutex to ensure its consistency. Also when a thread makes a blocking system call like read, another process is scheduled that schedules a thread. The processes can also run on different CPUs on a multi-processor architecture. Solaris, IRIX and Digital UNIX are based on this model.

2. Parallel Computing on multi-processor machines:

On multi-processor machines, real parallelism is present. Distinct tasks can be run on different processors and they all run in parallel *at the same time*. Tasks here can be threads or processes as described in the preceding section.

3. Distributed Computing:

Distributed computing is any computing involving multiple computers where each computer is involved in computation problem solving or information processing. When distributed computing is used to solve a problem, the problem is decomposed into tasks that are given to different machines on the network. Each computer gives some of its processor cycles for solving the problem. Since computers are present on a network, there will be network communication latencies involved when networked computers exchange information. Distributed computing is ideal when it is possible to decompose a computational problem into independent tasks.

Debugging Multi-Threaded programs:

One of the main issues with multi-threaded programs is debugging. Existing thread packages have mainly concentrated on the issues of portability, performance and POSIX 1003.1c compliance. Many libraries have provided some sort of interface to enable debugging by a standard debugger like GDB, but extensive support for this was not a major consideration. This thesis aims at building a thread library that provides better debugging capabilities to the debugger. There are two previously unaddressed aspects of debugging multi-threaded programs: testing concurrency scenarios and reproducing an execution sequence. Testing concurrency scenarios involves controlling how threads are executed. It should be possible to interleave instruction sequences from different threads so that the user can detect potential concurrency problems such as deadlocks and race conditions. It is obvious that this goal cannot be realized using a one-to-one model as the kernel scheduler schedules threads along with other processes and hence scheduling cannot be controlled. The many-to-one model is best suited for this need as the scheduler runs in user level and hence an interface can be given to the debugger to force context switches. The user can then test different concurrency scenarios by forcing context switches at appropriate points in the program. Reproducing an execution sequence of a concurrent program is another aspect of debugging multi-threaded programs. If the thread scheduler uses a preemptive scheduling policy, the timing of context switches can differ from one run of a program to another and hence a given execution cannot be reproduced. Other events, like blocking system calls and signals, affect concurrency and hence there is an element of nondeterminism in the threading model. One of the solutions to this problem is to map all threading primitives onto an event-driven framework like BERT. Then there is a single point of control, the Reactor [3], a software pattern that provides an event-driven framework, which can capture all the events that affect concurrency. A pattern is a general solution to category of problems. For a given set of software requirements, there might be a design pattern that already exists and hence can be used instead of reinventing the wheel. The Reactor is a software pattern for event-driven systems that provides capability to handle events from multiple sources in a single threaded environment. The Reactor examines a set of inputs and chooses one from the set that can be executed without blocking. The handler associated with this input is then called. The Reactor framework is implementation of the Reactor software pattern and application-specific methods for handling events of different kind are present in application-defined hook methods. The Reactor provides an event de-multiplexing framework wherein events detected on inputs can be mapped to specific event handler objects that have handler functions. The appropriate methods are called depending on the type of event detected. Necessary information for reproducing a concurrency scenario like CPU register state and the stack pointer of the thread, can be recorded and later used by the debugger for replay. By using this mechanism a specific thread execution sequence can be reproduced in the debugger. We have built a User Level Multi-Threading (ULMT) library: BERT Threads (BThreads) Library, that is preemptive and POSIX 1003.1c compliant for this purpose. A Thread Debug Interface should also be implemented so that the debugger can debug BThreads programs. Statements for recording concurrency information should also be included in the library that can be used later during replay.

The BThreads library also provides the capability of fine tuning scheduling policies according to the current application state. An event-driven application may be smoothly transitioned to a concurrent application using this library as it is built on an event-driven framework.

Chapter 2 discusses the relevant work in the area of multi-threading libraries and the debugging capabilities provided by those libraries.

Chapter 3 discusses the design of the BThreads library. The design of various interfaces for building a POSIX 1003.1c compliant preemptive multi-threading library is discussed in detail. These interfaces provide support for thread creation and destruction, synchronization, signals, cancellation and cleanup handling. Thread safety features are also discussed.

Chapter 4 discusses implementation-specific issues for implementing the design discussed in chapter 3. It also discusses how concurrency recording should be done in the BThreads library.

Chapter 5 discusses testing of the library, verifying its POSIX compliance, testing performance and different concurrency scenarios.

Chapter 6 discusses conclusions and possible future work.

2. Related Work

A number of multi-threading libraries based on different implementation models have been built in the past with various goals. GNU Portable Threads (Pth), a non-preemptive thread library, has been built with an aim to provide an interface that is portable across a wide variety of UNIX systems [4]. This goal is realized by using standard features present in UNIX systems and the ANSI C language without using any platform specific assembly code. By providing such an interface with no dependency on hardware platform, a completely portable library has been realized.

Next Generation POSIX Threading (NGPT) [5] is the brainchild of IBM. It is based on the many-to-many model. It is built on top of GNU Pth. The main goals of this multi-threading library are to provide complete POSIX compliance with an M:N threading capability that can lead to significant performance gains on Symmetric Multi-Processing (SMP) machines.

The Filaments [6] project is aimed at building very lightweight threads that provide efficient fine-grain parallelism. A fine-grain execution model defines the smallest block that supports concurrency on top of which medium-grain and coarse-grain tasks can be built. This can be used to explore implicit fine-grain parallelism present in functional or dataflow languages and iterative grid computations. However, fine-grain parallelism is inefficient and filaments overcome this by using stateless threads and overlapping communication [7]. A filament can be though of as a small unit of work that typically has a few to a few hundred instructions. For instance, a filament can be the body of a parallel `for` loop. Filaments are stackless and are executed concurrently by server threads. An application typically has thousands of filaments. Another key aspect of filaments is that they use shared-variable communication. Filaments can run on Distributed Shared Memory (DSM) systems and hence communicate by reading and writing to shared memory.

Cilk [8] is a multi-threaded programming language developed at MIT. The main goal of Cilk is to provide the programmer with an interface that allows exploiting inherent parallelism in program structures. This can be used, for instance, in recursive function calls where every recursive function is run as a separate thread of control. The run-time system takes care of scheduling these threads to ensure efficiency of computation. Cilk is built on the ANSI C language and it supports SMP, massively parallel computers. The Cilk multi-threaded language provides a debugging tool, Nondeterminator-2 [9], which detects race conditions in programs written in Cilk. It detects races that occur when a memory location, which is not protected by locks, is simultaneously accessed by multiple threads of control and at least one is writing to that location.

The FSU PThreads package [10] is a user level PThreads package that supports preemptive priority based scheduling. It also supports synchronous I/O for threads: a thread making a blocking call does not block the entire process. Asynchronous I/O is the mechanism used to implement this.

LinuxThreads [11] is an implementation of the POSIX IEEE 1003.1c standard for the LINUX platform. It is based on the one-to-one model. As pointed out earlier, debugging is difficult with this kind of implementation since threads are scheduled along with other processes and hence it is impossible to control thread scheduling. The LinuxThreads implementation does not have the ability to deliver an asynchronous signal generated externally and sent to the process as a whole to *any* thread that does not block the signal. This is due to fact that every thread has a specific pid and hence the signal can be delivered only to that thread.

The BERT interface, on top of which the BThreads library is built, was implemented as part of the BERT Project [12]. It provides an event-driven framework using the concept of the Reactor. The original Reactor pattern focused on handling Unix-style file descriptors. The BERT interface aims at applying the same concept to any event that affects concurrency and hence hampers reproducibility. These include timing of context switches, delivery of signals, blocking due to synchronous I/O calls, etc. Building a multi-threading library on top of this interface ensures reproducibility as all the information can be recorded at these deterministic points. An interface can be provided to a standard debugger like GDB to control BThreads programs. GDB can be modified to debug, test and reproduce BThread programs [13].

The Adaptive Communication Environment (ACE) project [14] is aimed at building an Object Oriented framework that provides a powerful set of reusable software that can be used for several purposes: event demultiplexing and event handler dispatching, signal handling, concurrent execution and synchronization. The ACE Thread library [15] is implemented in C++ as a component of the ACE tool kit. Its main design goals are:

- Provide a C++ thread library that uses underlying C thread libraries. This ensures *uniformity* of programming language when using threads with C++ code.
- Provide a *portable* thread library by using templates and operator overloading features in C++.
- Reduce the number of changes needed to make an application multi-thread safe.
- Minimize subtle synchronization errors. These include the programmer forgetting to unlock a previously locked mutex.

3. BThreads Design

BThreads is a ULMT Library based on the many-to-one model; i.e., many threads run on top of a single process. To build such a library, the ability to create user-level contexts is needed. The jmpbuf based functions provide the ability to implement user level threading and are present in standard C library. Hence they are present in every libc. The ucontext based functions are part of XPG4 standard and hence are present only in XPG4 compliant libc's, such as GNU Libc (glibc). Using these as basic building blocks, a full-fledged multi-threading library can be designed that supports POSIX compliant threading interface (IEEE 1003.1c) [16]. The BThreads API is POSIX compliant and provides support for thread creation and destruction, synchronization, cancellation, cleanup handling, thread-specific data and signaling. This section outlines the design behind the development of the BThreads library.

3.1 Thread States and Queues

In a ULMT library as states of all the threads are maintained at user level, state transitions of threads can be captured and checked. The possible thread states for the BThreads Library are:

Ready:

State associated with a thread that is ready to be scheduled.

Blocked:

State associated with a thread blocked on synchronization device (for example due to mutex or condition variable). This state is also associated with a thread waiting on termination of another thread.

Running:

State associated with the currently running thread.

Sigwait:

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 a specified set is delivered.

Waiting:

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

Killed:

State associated with a terminated thread.

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 independent of any other thread. If it is in the joinable state, another thread can wait on its termination. Memory resources consumed by a detached thread (thread stack, etc.) are deallocated when it terminates.

Sigwait Queue:

This queue holds the threads that are blocked in sigwait.

The state transition diagram for BThreads is shown in figure 3.1.1.

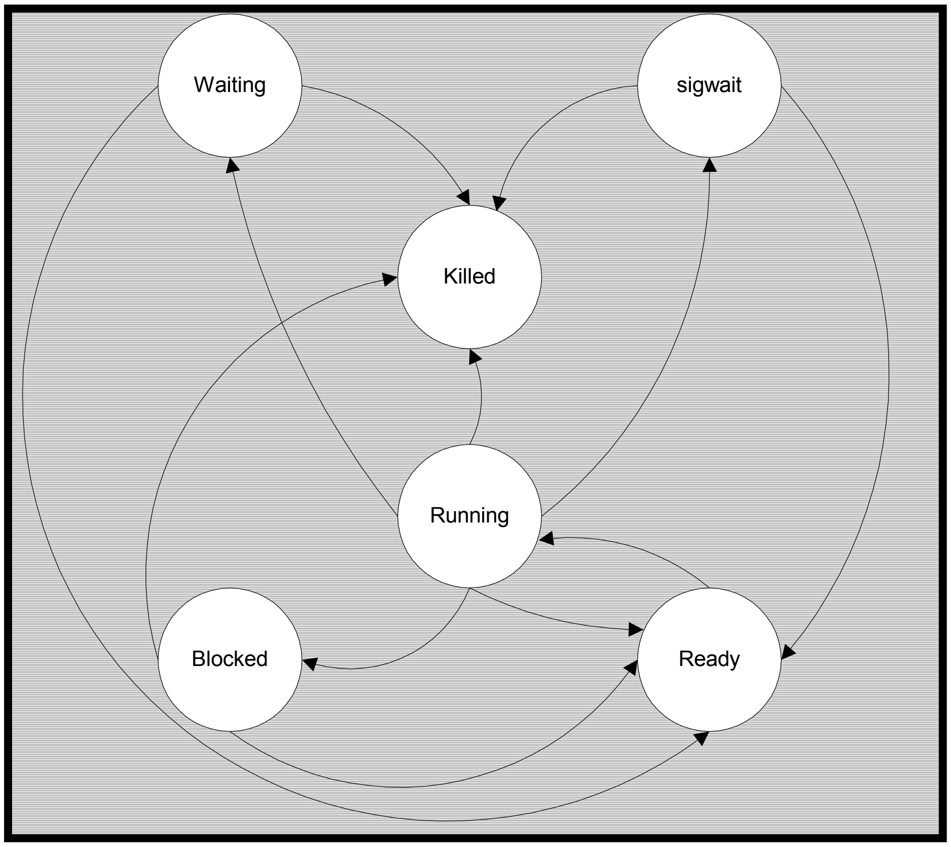


Figure 3.1.1 State Transition Diagram for BThreads Library

As shown in figure 3.1.1, 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 a transition to:

- the waiting state, by making I/O calls.

- the blocked state, due to synchronization.
- the sigwait state, by calling sigwait.
- the killed state, by calling the thread exit function or due to cancellation. Cancellation is a mechanism by which a thread can terminate execution of another thread. If cancellation is of asynchronous type, the thread receiving the cancellation request terminates immediately. If it is of deferred type, the thread receiving the cancellation request exits only when it reaches cancellation points.
- the ready state, as it ran till completion of its quantum. The BThreads library uses a Round Robin (RR) scheduling policy. RR Scheduling is a preemptive scheduling policy. The quantum is the maximum amount of time a thread can run before another thread is scheduled.

The Reactor is invoked whenever the scheduler is invoked. It checks for 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 frees the synchronization variable. A thread enters the sigwait state when it calls the sigwait function. It makes the transition to the ready state when one of the signals that it is waiting for is ready to be delivered. 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. Transition to the killed state is also possible if the thread exit function is called, or due to deferred cancellation.

3.2 Thread Creation and destruction

This section discusses thread creation and thread destruction in the BThreads library.

When a POSIX thread is created, a function is specified that marks its initial point of control. In BThreads, the function that is actually run is a wrapper function with the following structure:

```
_thread_init
call actual thread function
_thread_reap_terminated_threads
_thread_do_exit
```

By using the wrapper function, memory resources of the terminated threads in the detached state can be cleared by calling the _thread_reap_terminated_threads function. A detached thread that is terminated cannot release its own memory resources like stack, as it would be still running on that memory. So memory resources of detached threads that are terminated must be cleared from another threads stack. The wrapper function is also needed since _thread_do_exit should be called if it was not called before, as all the cleanup operations that happen when a thread exits should be done.

A separate library level thread manager (TM) thread might be necessary to clean up the memory resources like stacks of terminated threads in the detached state. The design decision that was made in this regard was to not have such a thread since there was no necessity. When a thread gets killed and is in the detached state, it is inserted into the termination queue and the memory resources of threads in this queue can be deallocated by another thread either when it is created or just before it exits. In LinuxThreads implementation, a separate TM thread is present.

3.3 Thread Scheduling

In the BThreads library, scheduling is done in 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 timer signals. A FIFO (First In First Out) scheduling policy that is inherently non-preemptive can be realized by turning off the generation of the scheduler timer signal. 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 the library operations.

3.4 Thread Synchronization

Synchronization primitives are especially important in the BThreads library to ensure data consistency. Due to the preemptive nature of the scheduling, thread context switches occur at unpredictable points of the execution. It is possible that a data structure is in an inconsistent state when a thread context switch occurs and the new thread can try to access the same data structure. To overcome this problem, synchronization primitives must be provided that ensure mutual exclusion of critical sections of code. This section discusses the synchronization primitives that are present at the library level (atomic locks, wait locks and spin locks) and the primitives that are provided to the user as part of the BThreads API (mutexes and condition variables). These primitives can be built in a layered way where each layer uses the synchronization capabilities provided by the layer below.

3.4.1 Atomic Locks

The lowest level synchronization support comes from the processor's hardware instructions. A single hardware instruction is guaranteed to be uninterrupted during its execution. Instructions like Bit Test and Set (BTS) or Exchange (XCHG) can be used for implementing atomic locks on the x86 architecture [17]. Similar instructions on other CPU's include SWAP and Fetch-And-Increment.

3.4.2 Wait Locks and Spin Locks

The next higher synchronization layer uses the underlying lower layer synchronization primitives: the atomic locks to protect their own critical sections and to provide additional synchronization capabilities. The synchronization primitives at this level are waitlocks and spinlocks [18].

When the waitlock is in the locked state, another thread that tries to acquire the lock is put in the blocked state and the scheduler is invoked. The blocked thread is woken up later when the waitlock is released.

The spinlock is conceptually similar to the waitlock, but the difference is that instead of blocking a thread when the lock is unavailable, the thread trying to acquire the lock busy-waits until it acquires the lock. This effectively wastes the thread quantum and can result in poor performance.

Waitlocks are used for protecting the critical sections of higher-level synchronization objects (mutexes and condition variables) owing to their efficiency.

3.4.3 Mutex Variables

A Mutex is a MUTual EXclusion construct that is used to ensure that only one thread is present in some critical section of code. It provides more capabilities than the locks that were discussed in the preceding sections. There are three kinds of mutexes (LINUX specific extensions to mutexes defined in the POSIX interface): the fast mutex, the error-checking mutex and the recursive mutex. These LINUX specific extensions provide additional features without loosing standard requirements for mutexes according to the POSIX. These mutexes differ in their behavior when a thread tries to acquire a mutex that is already locked by it. In case of the fast mutex, the thread deadlocks; for an error-checking mutex, an error code is returned; and for the recursive mutex, the lock is acquired again and the mutex count is incremented by one. If a thread tries to acquire a lock while some other thread is holding the mutex, it is put in the blocked state, inserted into the mutex queue and the scheduler is invoked. Similarly when a thread tries to release a mutex, for a fast mutex, it is always released irrespective of the actual owner thread; for a recursive mutex, count is decremented and only when the count reaches zero is the mutex released; and for an error-checking mutex the lock is released only if the calling thread and the owner thread are the same. During the process of releasing a lock, the thread releasing the lock checks if there are any threads that are waiting on this lock. If so it wakes up one of the threads by inserting it into the ready queue. The routines that facilitate the acquiring and releasing operations on the mutexes themselves need to be protected by the lower-level synchronization primitives to ensure their mutual exclusion. The design choice to use the waitlocks as underlying locks was made in the current case for performance reasons since the critical sections of the mutex acquire and release methods are quite big by themselves, a thread context switch is probable in the middle of these operations and the next thread scheduled can try to perform an operation on the same mutex. This would result in busy waiting of the new thread (wasting CPU) if spinlocks were used.

3.4.4 Condition Variables

Condition variables are used when a thread has to wait until some predicate or condition on the shared data becomes true. The basic operations on condition variables are waiting on a condition variable (thread_cond_wait) and signaling a condition variable to wake up any threads that are waiting on it. When signaling a condition variable, either a single thread that is waiting on it can be woken up (thread_cond_signal) or all the threads that are waiting on the condition variable can be woken up (thread_cond_broadcast). Condition variables differ from mutexes, as they are not associated with a value or a state. A thread trying to acquire a mutex can be blocked depending on whether the mutex is free or held. With a condition variable, when a thread calls thread_cond_wait it is *always* added to the condition queue and starts running only when the condition variable is signaled by another thread. The mutual exclusion requirements for condition variables are interesting. A condition variable must always be associated with an external mutex lock to avoid the race between a thread about to wait on a condition variable and another thread signaling the condition variable before the first thread actually waits on it. There must be an internal lock that is transparent to the user to atomically unlock the mutex, add the thread to the condition queue and go to the blocked state waiting for the condition. The internal lock is needed to avoid the above said race condition, due to the possibility of arbitrary interleaving of instructions in the thread_cond_wait and thread_cond_signal methods. The design choice to use the waitlocks as the internal locks was made due to performance reasons as a thread context switch is possible in the middle of these operations and the next thread scheduled can try to perform an operation on the same condition variable and can busy wait if spin locks are used.

There are two different semantics that define how to wake up a thread blocked on a condition variable: Mesa-style semantics and Hoare-style semantics [19]. In Mesa-style semantics, whenever a thread wakes up another thread that is waiting on a condition variable, it just puts the waiting thread into the ready queue. It is the responsibility of the awakened thread to reacquire the external mutex lock before proceeding further. An important implication of Mesa-style semantics is that the predicate may not hold though the condition variable has been signaled, since another thread might be scheduled before the thread blocked in thread_cond_wait is scheduled and it may modify the variables that alter the predicate. So the blocked thread needs to recheck the condition once it starts running and call thread_cond_wait if needed. In Hoare-Style semantics, the thread signaling the condition gives up control over the CPU and the external mutex lock to the blocked thread. Hence in this case we ensure that once the blocked thread starts running, it reacquires the lock and runs immediately. So the predicate must always hold true if Hoare-style semantics are used. The design choice that was made for the BThreads library was to use Mesa-style semantics since the POSIX threading interface specifies such semantics.

3.4.5 Thread Joining

A thread's detach state can be joinable or detached. When detached, it runs independent of any other thread and its memory resources are deallocated when it exits. When joinable, another thread can join on termination of this thread. So this can be viewed as a sort of synchronization operation since the thread that joins on another thread is blocked until the latter terminates. The memory resources (stack, etc.) of a joinable thread are deallocated by the joining thread.

3.5 I/O

For a ULMT library, when a thread makes a blocking I/O call like read () or recv (), the process as a whole blocks until the read request is satisfied. During this time no other thread can be scheduled. To overcome this problem, the Reactor can be used [3]. This section discusses the Reactor framework and how it can be used in the BThreads library to handle I/O requests. The BERT interface uses the Reactor framework to dispatch various kinds of events [12].

This section discusses the Reactor framework and how it can be used for I/O handling in the BThreads library. It discusses how the system calls that can block (read(), write(), etc) should be wrapped so that the Reactor is involved in I/O handling. Generic algorithms for the open(), close(), read() and write() system calls are also given.

3.5.1 Overview of the Reactor Framework

The Reactor provides an interface by which an event handler objects can be registered with it. Event handler objects implement at least three methods: handle_input, handle_output and handle_close. The handle_input method is called whenever data is ready to be received. This is associated with the read(), recv() system call. The handle_output method is called whenever the output buffer is free to send data. This is associated with the socket send() system call. The handle_close method is called to check if an event handler object has completely deregistered itself with the Reactor.

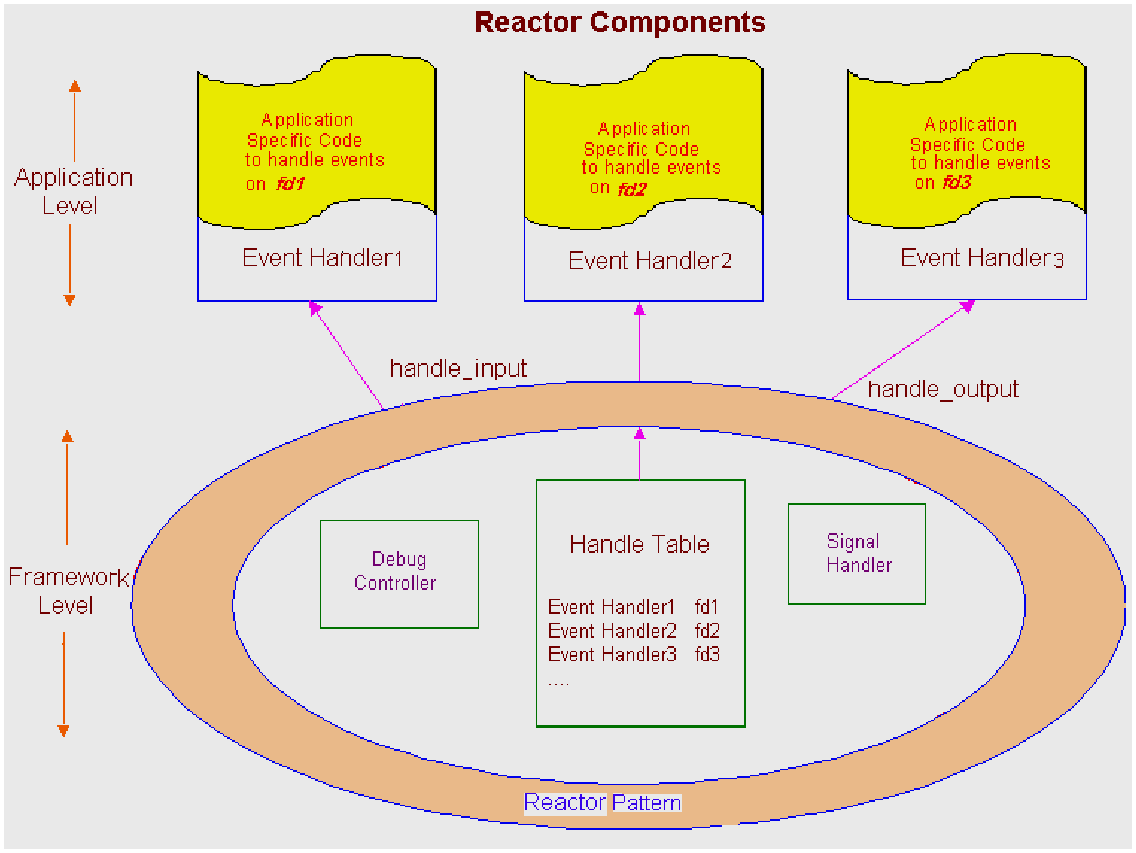


Figure 3.5.1.1 The Reactor Pattern for I/O Handling [12].

An application typically registers its handlers with the Reactor and finally calls the Reactor's handleEvents method, which checks for the availability of any pending I/O requests. If there are any pending I/O requests that can be satisfied, the callback functions that are registered by the handler are called by the Reactor. This makes the Reactor framework event-driven. The framework demultiplexes the event; i.e., maps the event detected on a file descriptor to an event handler object. There are no queues associated with the threads waiting on I/O since the callback mechanism of the Reactor ensures that the correct thread is put in the ready state if its I/O request can be satisfied immediately.

3.5.2 Using the Reactor for multi-threaded I/O

As discussed above, the primary objective of the Reactor framework is to provide the ability to register callback functions that are called when I/O is ready. For ULMT libraries that are based on the many-to-one model, the process as a whole blocks when a thread makes a blocking I/O call. No thread can be scheduled until the I/O call is complete. But using the concept of the Reactor, we can make the BThreads library non-blocking as we can register the I/O events we are interested in with the Reactor when I/O requests are received and schedule another thread. This thread is woken up later when the I/O request it made can be satisfied. One issue when using the Reactor pattern is when the Reactor method that checks for the availability of the pending I/O events should be invoked. Since BThreads is a preemptive multi-threading library we can call it whenever the scheduler is invoked, ensuring that we check for the completion of the I/O events at least every RR quantum. The scheduler is also invoked whenever a thread makes a state transition to the blocked, waiting, killed or sigwait states and hence a check for I/O completion is also made at these points.

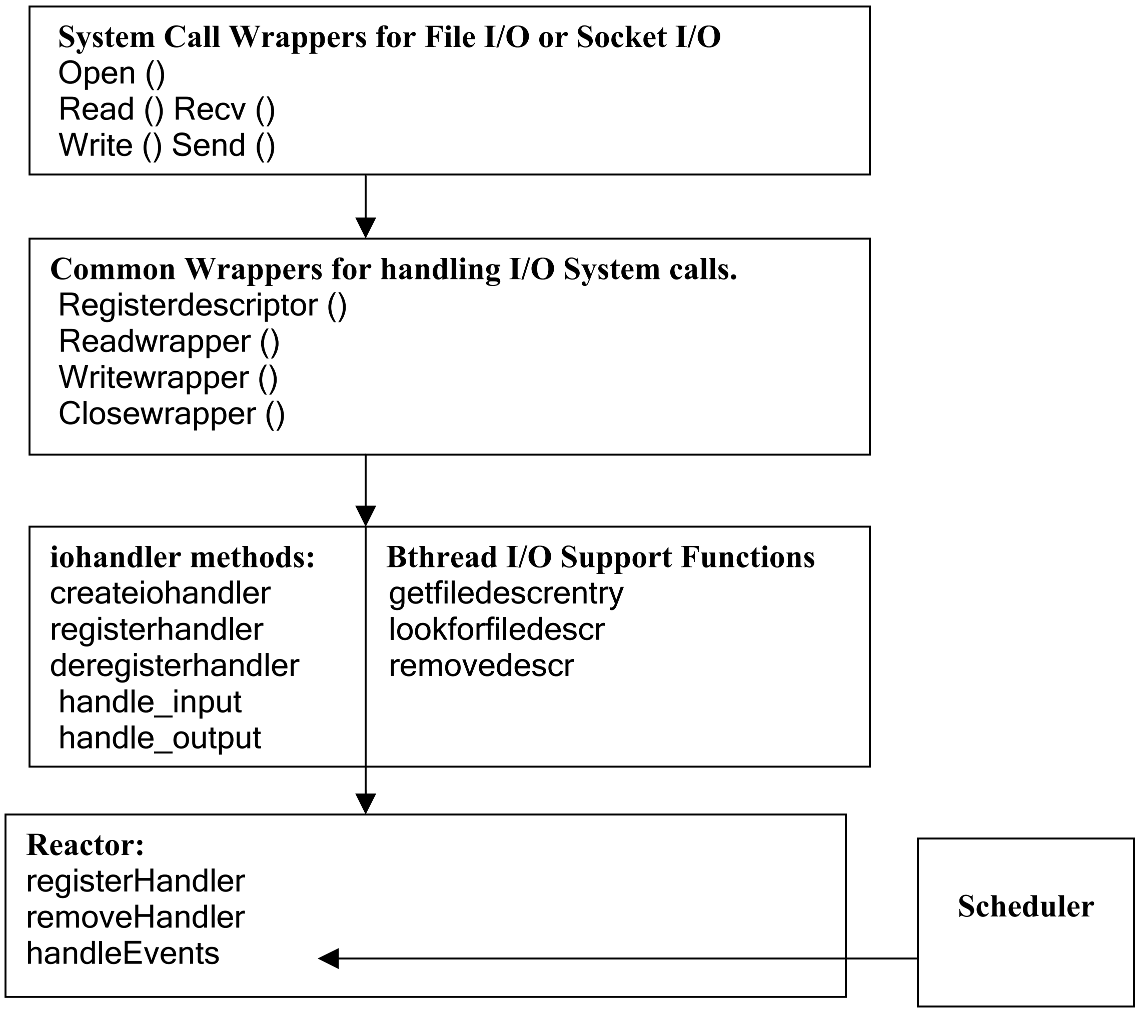
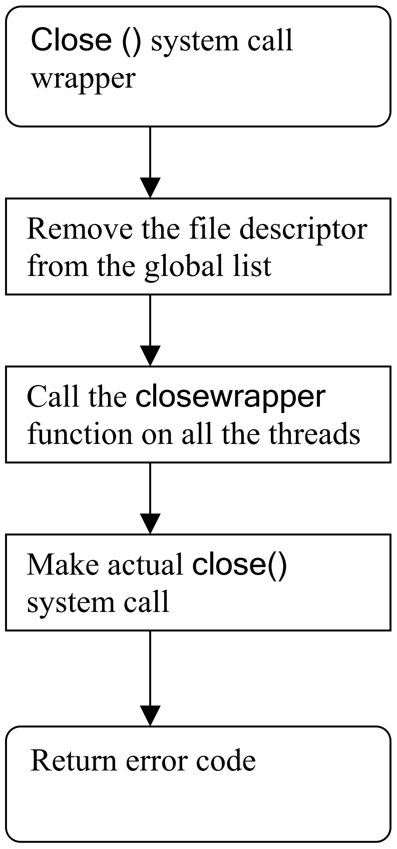


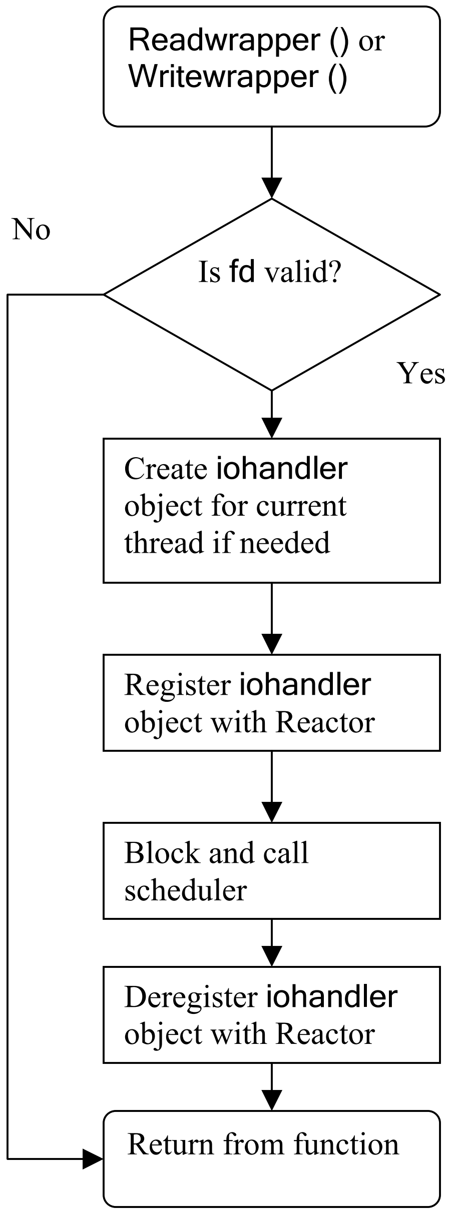
Figure 3.5.2.1 Multi-Threaded I/O in BThreads Library

As shown in figure 3.5.2.1, the system calls to open(), close(), read(), recv(), send() and write() data on file descriptors are wrapped. When a system call is made to open a new file descriptor (fd) using wrapper functions, Open() or Socket(), the actual system call is made. If the system call succeeds, the newly created fd is included in the global list of file descriptors created by all the threads running in this process.

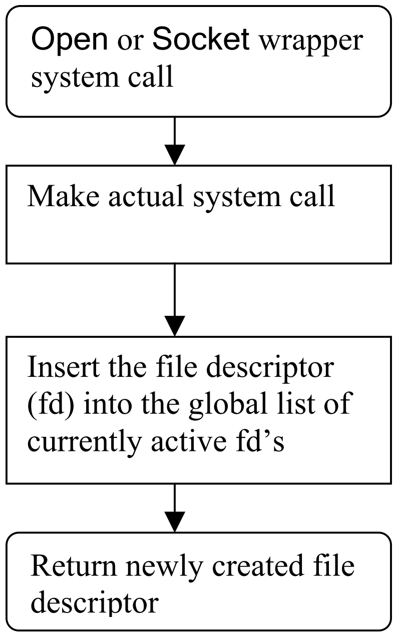
When a read() or write() system call is made, the corresponding wrapper function (readwrapper or writewrapper) is called. A check is performed if the fd is valid by checking if it is present in the global list of currently active file descriptors and an error is returned if it is invalid. If the fd is valid, a new iohandler object is created if needed, which is included in the thread's TCB. An iohandler object has all the information that is associated with the file descriptor. This includes: fd, the id of the thread that created an iohandler object, the number of times the iohandler object is registered with the Reactor. The iohandler object is registered with the Reactor before entering the WAITING state. The state of the thread is set to WAITING, the purpose of wait is set to the appropriate value, READING or WRITING and the scheduler is invoked. This thread starts running again when the Reactor calls the handle_input or handle_output method upon the detection of the availability of an I/O event. These callback functions insert the waiting thread into the ready queue. After waking up from the waiting state, the iohandler object is deregistered from the Reactor and the actual system call is made. Error checking is done based on the return value of the system call and the number of bytes read or written is returned to the user. The sequence of steps that take place when system call wrapper functions, Read(), Write(), Open() and Socket(),are called is shown in Figure 3.5.2.2 a, b.


Close () system call wrapper
Remove the file descriptor from the global list
Call the closewrapper function on all the threads
Make actual close() system call
Return error code

d. Close system call wrapper

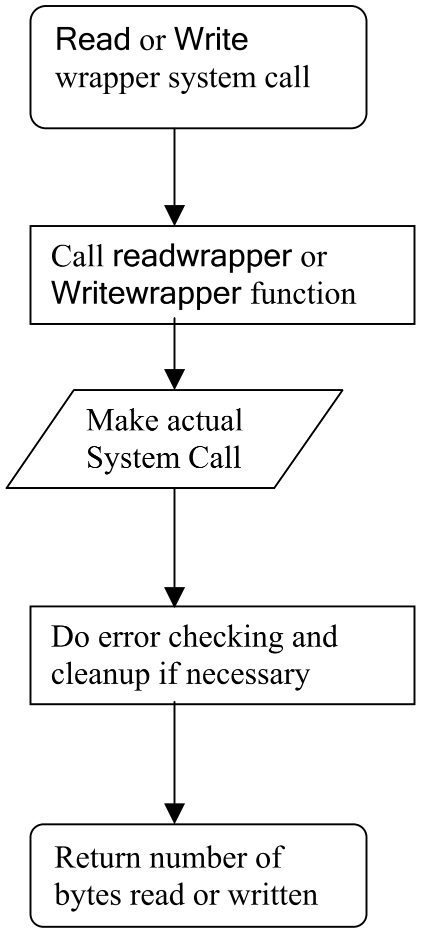

Readwrapper () or Writewrapper ()
No
Is fd valid?
Yes
Create iohandler object for current thread if needed
Register iohandler object with Reactor
Block and call scheduler
Deregister iohandler object with Reactor
Return from function

e. Readwrapper and Writewrapper functions



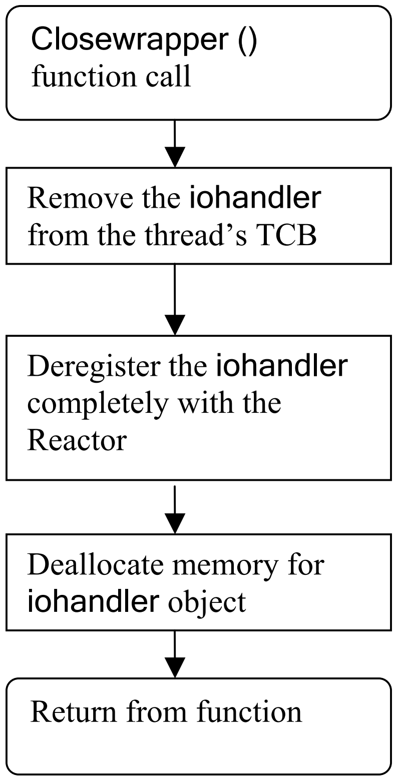
Open or Socket wrapper system call
Make actual system call
Insert the file descriptor (fd) into the global list of currently active fd's
Return newly created file descriptor

a. Open/Socket System call wrapper



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

b. Read/Write System Call wrapper



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

c. Closewrapper function

Figure 3.5.2.2 Algorithms showing control flow for I/O System calls.

An iohandler object is registered with the Reactor only when a read or write request is received and is deregistered when the request is satisfied.

Whenever an error occurs during the execution of an I/O request that signifies that other the end of connection is closed (e.g. for sockets or pipes), the closewrapper function is called. It is also called when fd is closed as shown in Figure 3.5.2.2 d.

3.6 Signals

In a multi-threading library, blocking and delivery of signals must be handled in a thread-specific way. The signal handlers are shared among all threads, while the signal masks can be set on a per-thread basis. In the BThreads library, all the signals can be blocked except the scheduler timer signal that is used to implement the preemptive scheduling policy. The standard requirements for signal handling in a multi-threading library, according to the POSIX threading interface [20], can be summarized as below:

a. Synchronous Signals

These signals are generated due to the execution of an instruction by a thread. For example, division by zero generates the SIGFPE signal. These signals should be delivered to the thread that executed the instruction.

b. Asynchronous Signals

These signals are generated asynchronously; i.e., at unknown points of time. These signals can be of generated in two ways:

i. Externally Generated Asynchronous Signals (EGAS)

These signals are generated due to events external to the process. An example of this is the SIGINT signal generated from the tty interface.

ii. Asynchronous Signals Generated by Other Threads (ASGOT)

These signals are generated when a thread sends a signal to another thread using the thread_kill function. A field in the TCB is used to store requests for generating signals of the ASGOT type. The signals that are present in this field are delivered later.

The handling of the asynchronous signals differs according to how they affect the execution of the program i.e. whether they cause program termination or not.

a. Asynchronous fatal signals

This class of asynchronous signals results in termination of the process. They can be of EGAS or ASGOT type. According to the POSIX semantics, when a process receives a signal of this kind all the threads that are executing in the process must be terminated. This is the default behavior in the BThreads library. When a process terminates all the threads running in that process automatically terminate.

b. Asynchronous non-fatal signals

This class of asynchronous signals doesn't result in termination of the process. They can be of EGAS or ASGOT type. According to the POSIX semantics, when a process receives a signal of this kind, it is delivered to at most one thread that currently doesn't block the signal.

3.6.1 Signal Delivery:

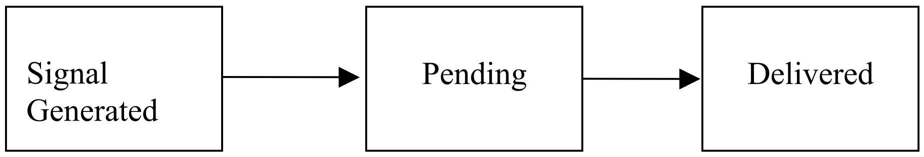


Figure 3.6.1.1 Signal Delivery in BThreads

As discussed in the preceding section, a signal can be generated due to

- Execution of an instruction by a thread. This generates a synchronous signal.
- Event external to the process as a whole. This generates a signal of EGAS type.
- The thread_kill function call. This generates a signal of ASGOT type.

When a signal is generated, it is in the pending state until it is delivered. A signal may not be delivered to a thread if it is blocked in the thread or if it is to be delivered to a specific thread that is not yet scheduled. Once a signal is delivered, the appropriate action associated with the signal takes place.

In the BThreads library, a signal is delivered in the following circumstances:

a. Signal mask of the thread is changed

Whenever the signal mask of a thread is changed, any signals of EGAS type are automatically delivered to the thread. In addition, any signals in the pending signal set of the thread that are now unblocked are delivered.

b. Before scheduling a new thread in the scheduler

Just before a context switch to a new thread in the scheduler, a check is performed if there are any signals that can be delivered to the thread that is going to be scheduled. If there are any signals of ASGOT type, the raise_threads function is inserted on the execution stack of the thread. This function calls the raise() function to deliver all the pending signals of ASGOT type that are not blocked by the thread. We cannot just insert the signal handler function on the thread stack since this is not equivalent to raising a signal. When the kernel delivers a signal, different data structures like sigcontext, etc are pushed onto the execution stack of the thread. If there are signals of EGAS type that are not blocked by the thread that is going to be scheduled, they are included in the thread's signal mask and a function call to unblock them is inserted on the thread stack. Signals of EGAS type are delivered to the thread by the kernel when the signals are unblocked. During context switching, the signal mask of the new thread comes into effect before it starts running and hence signals of EGAS type would be delivered to the old thread if they were not blocked. To avoid this, these signals are blocked and later unblocked.

c. Checking for availability of signals for threads blocked in sigwait

When a thread calls sigwait, it is blocked until one of the signals in the sigwait set is delivered. Whenever the scheduler is invoked, a check is performed if there are any threads blocked in the sigwait and if there are any pending signals of ASGOT or EGAS type that also belong to the sigwait set. If there are signals that match this criterion, the thread blocked in sigwait is scheduled next. Before scheduling the thread, signals are handled as outlined in the preceding section (3.6.1.b).

3.7 Thread Cancellation

A thread can terminate another active thread using cancellation. When a thread sends a cancellation request to another thread, the other thread can ignore the request, honor it immediately or defer it until it reaches a cancellation point. The cancellation request is executed by calling the _thread_do_exit function. Cancellation itself can be enabled or disabled. If it is disabled, the cancellation request is not honored. If it is enabled, the cancellation is executed depending on the cancellation type. If the cancellation type is asynchronous cancellation, the thread exits immediately. If it is deferred cancellation, the thread exits only at a cancellation point. Cancellation points are those points in the code where any pending cancellations requests are executed. These are: thread_join, thread_cond_wait, sigwait and thread_testcancel.

When cancellation is of the asynchronous type, a function call to the thread_exit function is inserted on the thread stack so that it exits when it starts running. When cancellation is of the deferred type, the pending cancellation requests are executed at the cancellation points.

3.7.1 Extrication Interfaces

When a thread is in the blocked state, waiting on some object, such as a condition variable, or an event, such as termination of another thread and it receives a cancellation request, it should be removed from the waiting queue of the object it is waiting on before it exits. An extrication interface is used for this purpose. An extrication interface is registered just before a thread enters the blocked state. An extrication interface has a method that removes the thread from the waiting queue. If a cancellation request is received while the thread is in the blocked state, this method is called before executing the cancellation. In case no cancellation occurs, the extrication interface is deregistered once the thread wakes up. An extrication interface is active only for the duration of the wait.

3.7.2 Cancellation points

Cancellation points are points in the code where pending cancellation requests are executed. They are of relevance when the cancellation is of the deferred type.

When a thread is woken up from the blocked state (blocked in the thread_join or sigwait function), a check is performed if it was woken up due to a cancellation and if the cancellation is enabled. If these criteria are met, thread_exit function is called. When a thread blocked in sigwait is woken up, the thread_testcancel function is called to execute any pending cancellation requests.

The functions involving mutex variables are not cancellation points. This ensures consistency in the state of the mutex at cancellation points.

3.8 Cleanup Handlers

The cleanup handler functions are called when a thread exits either due to an explicit call to the thread_exit routine or due to a cancellation request from another thread. When a thread exits, the cleanup handlers that were installed are called in reverse order of their registration (LIFO/stack discipline). The purpose of the cleanup handler functions is to release any resources that a thread might be holding before it exits. These resources include locked mutexes, open file descriptors and memory allocated on the heap with the malloc () system call.

If a thread holding a mutex exits due to cancellation, any other thread that tries to acquire the same mutex would block forever. To prevent this situation, a cleanup handler that unlocks the mutex should be installed before locking it. This ensures that the mutex is returned to the unlocked state before the thread exits. When cancellation is of the asynchronous type, a cancellation request for a thread may be received just after the cleanup handler (thread_mutex_unlock) is installed but before the mutex is locked. In that case, thread_mutex_unlock is called on an unlocked mutex. To prevent this, deferred cancellation must be used.

3.9 Thread Safety

A library is said to be thread safe if the methods in its interface can be called from multiple threads simultaneously without affecting the consistency of data structures. As the BThreads library is a preemptive library, a thread context switch can take place at any point in either user code or the library code. The next thread that is scheduled may access the same data structure that the previous thread has left in an inconsistent state. For ensuring consistency of the user level data structures, synchronization primitives like the mutexes and locks discussed in the section 3.4 can be used. The library level data structures must also be maintained in a consistent state for correct operation of the library. One solution to realize this is to make all the functions reentrant. Reentrant functions use only variables on the stack and call reentrant functions only [21]. This solution cannot be used with the BThreads library since it needs global data like the TCB and the queues, to name a few. In the design of the BThreads library, the following two approaches were taken to ensure consistency of the library level data structures:

a. Consistency using atomicity

In this solution, to enforce consistency all the signals are masked at the entry point of the critical section and then restored at its exit point. This may result in delayed delivery of the scheduling timer signal. This solution is used when operating on the data structures that are accessed in different functions in the library code and in the scheduler. The scheduler is invoked at least once every RR quantum and this can occur at random points in the library code. If the scheduler is invoked while a thread is modifying a global data structure in the library level that is protected by a lock and the scheduler tries to access the same data structure using the lock, it deadlocks. Also recursive locks cannot be used to this end as the data structure is in an inconsistent state and hence it is not proper for the scheduler to access it. To avoid this situation, the solution outlined in this section can be used. This solution may affect the scheduler semantics. If the scheduler timer signal is generated in the midst of the critical section, it will be delivered to the process only when the signals are re-enabled. Hence the RR Quantum for the current thread is effectively increased.

b. Consistency using Mutual Exclusion

In this solution, consistency is enforced using waitlocks. The waitlock ensures mutual exclusion for a critical section of code. If the scheduler timer signal occurs when a thread is in a critical section that is protected by a waitlock and the next scheduled thread tries to obtain the same waitlock, it is put in the blocked state and the scheduler is invoked. This solution can be used to enforce consistency of the library level data structures that are not shared by the scheduler and the functions in the library level code.

The following are some of the data structures in the BThreads library that need to be protected to maintain their consistency:

i. Queues

The BThreads library has internal queues (the ready queue, the termination queue and the sigwait queue) for managing library operations. The scheduler and some sections of the library code access the ready and sigwait queues. The solution outlined in section 3.9.a can be used for enforcing consistency of these queues. The termination queue is not accessed in the scheduler. Hence the solution outlined in section 3.9.b can be used for enforcing its consistency.

ii. Thread Control Block (TCB)

The TCB is analogous to the Process Control Block (PCB) that stores information related to the process. The TCB stores the thread related information necessary for the library level operations. This data structure has to be maintained in a consistent state as it is used in various routines in the BThreads library. To enforce its consistency, solution outlined in section 3.9 b can be used. TCB is associated with a waitlock that should be acquired before it is accessed.

iii. Reactor Queue:

The Reactor queue is the list of iohandler objects that are checked for the availability of I/O. The Reactor queue is accessed from two points in the library level code. When registering and deregistering iohandler objects with the Reactor, an iohandler object is added to or removed from this queue. When the handleEvents method is called to check for availability of pending

I/O events, the Reactor queue is accessed to invoke callback functions on the iohandler objects. Since the handleEvents method is called from the scheduler, the solution outlined in section 3.9.a can be used to enforce consistency of the Reactor queue.

The design of the BThreads library has been detailed in this section. The design of various modules in the BThreads API namely, thread creation and destruction, synchronization, signals, cancellation handling, cleanup handling and safety, were discussed. The details of the library level data structures the TCB and queues were not discussed in this section. Description of these data structures, testing and recording of various concurrency scenarios and implementation specific details are discussed in the next chapter.

4. BThreads Implementation.

This chapter deals with the implementation of the BThreads library. The design of the BThreads library was discussed in detail in chapter 3. This chapter discusses sections of the design that have special implementation issues. It also discusses the important library level data structures necessary to implement the design. The Thread Debug Interface (TDI) for supporting debugging of BThreads programs using GDB and testing and recording of concurrency scenarios are also discussed.

4.1 Creation of user space threads

To facilitate creation of multiple threads of control at the user level, the following interfaces present in the glibc library can be used:

- The Ucontext API, which provides the getcontext, setcontext, makecontext, swapcontext methods. It conforms to XPG4-UNIX.
- The Jmp_buf based functions, setjmp and lonjmp, or sigjmp_buf based functions, sigsetjmp and siglongjmp, conforming to the POSIX.

In the implementation of the BThreads library, the ucontext API was used since it provides an interface that is best suited for implementing cooperative multi-threading. The methods in the API can be directly used to create and switch among user space threads. It is very difficult to implement multi-threading using jmpbuf based functions.

4.2 Separation of User and Library Level interfaces and data structures

In designing a library, there has to be a clear delineation between what is visible to the user and what is hidden. The BThreads API provides the user with part of the IEEE 1003.1c POSIX compliant multi-threading interface. There are several data structures in the library level that have information about active threads. These data structures are hidden from the user.

4.2.1 Library Level Data Structures

This section discusses the library level data structures in the BThreads library. The important library level data structures in the BThreads library are: thread_handle_struct, thread_descr_struct (TCB), Queue and iohandler. These data structures are used throughout the library.

a. Thread_handle_struct:

There is a limit on the maximum number of the threads that can be created by the BThreads library. The limit depends on the initial thread stack size and the virtual memory limit for process. Threads may also allocate memory on heap as they run and hence we cannot use all virtual memory space for thread stacks. The maximum number of threads for BThreads library was fixed at 1024. Whenever a thread is created, a numeric thread identifier is returned to user, which can be used to specify all future operations on that thread. Due to the limit on the maximum number of threads, an array of data structures, thread_handles, can be defined. One of the elements of this data structure should be a pointer to the TCB. The thread identifier can be generated during thread creation time in such a way that it can be mapped to a unique index in this array. This kind of layout for the data structure facilitates easy recovery of the TCB associated with a given thread identifier since the index corresponding to the thread identifier can be recovered by applying reverse mapping. The TCB pointer can be obtained by accessing the corresponding field in the array at this index. The data structure that has a pointer to the TCB is thread_handle_struct. The fields of this data structure are: a field that specifies if it is currently being used, pointer to the TCB, the thread bottom that corresponds to the lowest address on the thread stack, a lock to ensure mutually exclusive access, and the thread start function pointer.

b. Thread Control Block (TCB) or thread_descr_struct:

The TCB data structure is the heart of the BThreads library. The fields present in the TCB are listed below:

Lock:

This is used to ensure mutually exclusive access to the TCB.

Context information:

This includes the state of all processor registers (the floating-point registers, the general-purpose registers and the flags register), the signal mask, the stack pointer and the stack size. Whenever a thread context switch occurs, the context information of the running thread is saved in this field and the context information of the new thread that is going to run is loaded by accessing this element from that thread's TCB.

thread id:

This is the unique numeric identifier for a thread. It can be mapped to a unique index in the thread_handles array.

Cancellation Fields:

For managing thread cancellation, the following information is stored in the TCB.

Cancellation State: This field specifies whether cancellation is enabled or disabled.

Cancellation Type: This field specifies the type of cancellation. It can be asynchronous cancellation or deferred cancellation.

Canceled: This field is set if a cancellation request was sent to this thread. This is used while checking for pending cancellation requests.

Wokenup_by_cancel: This field is set if the thread was:

i. Woken up from the blocked state due to cancellation.
ii. Removed from the queue associated with the object it blocked on.

Extrint: This field specifies the extrication interface for the thread. This is set if the thread is blocked in the thread_cond_wait or thread_join.

Details of cancellation were discussed in the section 3.7.

Name:

This field specifies the name associated with the thread. It can be used for debugging purposes.

State:

This field specifies the state associated with the thread. It is one of ready, running, blocked, sigwait, waiting or killed. The details of state transitions were discussed in section 3.1.

Detach state:

This field specifies the detach state of the thread. It can be detached or joinable. When a thread is detached, it runs independent of any other thread. If joinable, another thread can wait on its termination.

Signaling elements:

The following fields are provided for supporting signal handling in BThreads:

- Pending set: The set of signals that have been sent by other threads to this thread but not yet delivered.
- Sigwaitset: The pointer to the set of signals that a thread is waiting for. This field is non-null only if the thread is blocked in sigwait.
- t_signal: The signal number of the last signal that was received by this thread. This is generally used when a thread returns from the blocked state in the sigwait function to find out the actual signal delivered.

- t_sigwaiting: This is set if the thread is blocked in sigwait. The signal handler registered by the user is not run if the thread is blocked in sigwait.

BThreads signal handling was discussed in section 3.6.

Thread Specific Data:

In a multi-threaded program, the capability to refer to a specific variable from any point in the execution of a specific thread may be needed. Global variables cannot be used for this purpose since they are visible to other threads and have the same value in all threads. Thread Specific Data (TSD) is an alternative to global data in a multi-threaded program. TSD is associated with every thread and it can be set and recovered from any point in the execution of a thread. TSD is an array of void * pointers that point to the region of memory holding the actual data. Specific details of TSD are given in section 4.8.

Scheduling Parameters for the thread:

Thread Priority: This field specifies the priority associated with this thread. This is used when the priority based scheduling policy is used. Priority based scheduling has not been implemented in the BThreads Library.

Scheduling Policy: This field specifies the scheduling policy associated with this thread. The default scheduling policy in the BThreads library is Round Robin (RR) scheduling. In the case of Linux Threads, the kernel provides scheduling support. It is of the multi-level priority feedback queuing type with support for different scheduling policies (e.g., First In First Out (FIFO), Round Robin (RR)) at each static priority level. The scheduling policies in BThreads were discussed in section 3.3.

Contention Scope: For a ULMT Library like BThreads, a thread contends for the CPU with the other threads that are running in that process. For a Kernel Level Multi Threading (KLMT) library, a thread contends for the CPU with all the other processes running on the system. The contention scope signifies the scope within which a process contends for the CPU. It is process-wide for a ULMT library, and system-wide for a KLMT library.

Inherit sched: This field decides how the scheduling policy and the scheduling parameters for a newly created thread are set. If it is inherit_sched, the scheduling policy and parameters for the newly created thread are inherited from the parent thread. If it is explicit_sched, the new thread explicitly sets them.

t_errno:

For a multi-threaded program, the errno variable should be maintained on a per-thread basis. The errno variable in a given thread of execution must refer to the error that occurred during the system call made by that thread and not another thread. For this purpose the __REENTRANT macro should be defined and the errno_location function must be defined to point to the address of the per-thread errno variable: t_errno. When the __REENTRANT macro is defined, the errno is the value at the address returned by the errno_location function, which is t_errno.

waithead:

When a thread is in the joinable state, another thread can wait on its termination. The waithead of a thread points to the TCB of the thread that is waiting on its termination. This field is observed just before a thread exits. If it is non-null, the waiting thread is woken up.

t_cleanuptop:

This field is a pointer to the top of the Last In First Out (LIFO) stack of registered cleanup handler functions. The cleanup handler functions are registered by calling the thread_cleanup_push function. These functions are called just before a thread exits in the reverse order of their registration. Cleanup handling was discussed in section 3.8.

tiodata:

This field is a pointer to a data structure that holds information about the file descriptors opened by this thread and the corresponding iohandler objects. Iohandler objects are created whenever a new file descriptor is created due to opening a file or socket. They can be registered later with the Reactor to check for the availability of I/O events. I/O handling in BThreads was discussed in section 3.5.

GDB elements:

These fields are used by GDB. They have event related information associated with this thread. Events are generated to notify GDB about the occurrence of some action in the BThreads library. The following three events may be reported to GDB:

Thread creation event:

This event signifies the creation of a new thread.

Thread death Event:

This event signifies the termination of a thread.

Thread reap event:

This event signifies deallocation of the memory resources of a detached thread.

ut_report_events:

GDB can enable or disable generic event reporting for a specific thread by setting or clearing this element.

p_eventbuf:

This field has the following elements:

Eventmask:

The mask of the specific events enabled for this thread.

Eventdata:

The data associated with an event.

Eventnum:

The numeric identifier for an event. It is TD_CREATE, TD_DEATH, and TD_REAP for creation, death and reaping events respectively.

Eventdata, eventnum and the TCB of the thread generating the event are accessed by GDB when an event is reported to it.

thread_threads_events:

The global variable that stores the event mask common to all threads.

When reporting an event, a check is performed if generic event reporting (ut_report_events) is enabled for the thread reporting the event. If it is enabled, then a check is performed if reporting of this specific event is enabled in the common event mask (thread_threads_events) or in the event mask of this thread (p_eventbuf.eventmask). If one of these is enabled, the event is reported to GDB.

Guard address and guard size:

Guard pages are used to detect thread stack overflows. When a stack is created for a thread, guard pages are appended to the stack depending on the direction of growth of the stack. The guard pages are protected and they cannot be accessed for reading, writing or executing. When a thread stack overflows, the memory in the guard pages is accessed, a segmentation violation occurs and the stack overflow is detected. The default guard size is one page. The guard address and the guard size can be set before a new thread is created. The BThreads library supports user-defined stacks and stack sizes.

t_oncep:

The mutex, the condition variable, and the thread key (for the TSD) should be initialized before use. If these variables are global and an initialization routine to initialize them can be called from different threads, care must be taken to ensure that the initialization is performed only once. The BThreads library provides the thread_once method for this purpose. This method takes two arguments: a thread_once_t argument and an initialization function that initializes the global variable. The thread_once_t argument should be statically initialized before calling the thread_once function. Before calling an initialization routine from the thread_once function, the value of the thread_once_t variable is changed. The thread_once function checks the value of the thread_once_t argument and calls an initialization function only if its value is unchanged. T_oncep is a pointer to the thread_once_t variable. This is non-null only if the thread was cancelled while it was executing an initialization routine. In that case, the value of the variable at this address is reset before the thread exits so that an initialization routine is called when another thread calls the thread_once function.

t_retval:

This is a placeholder for the return value of the thread. This is a void * pointer and is an argument to the thread_exit function. A thread in the joinable state when terminating reports its return value to the joining thread through this field.

c. Thread Queues:

In the BThreads library, queues have been implemented as Abstract Data Types (ADT). Implementation of the queues is hidden from the interface for performing operations on them. In the current implementation, the queue has been implemented as an array where the value at each index in the array is a thread id. Different queues present in the BThreads library were discussed in the section 3.1.

d. Iohandler object:

The Reactor calls the methods present in an iohandler object upon detection of the availability of I/O. Private data of an iohandler object is listed in table 4.2.1.1. The count variable is used before cleaning up an iohandler object to make sure that it is not registered with the Reactor.

If a thread is waiting for an I/O completion, the current purpose variable is set to the appropriate value (the READING or the WRITING) depending on the purpose of the wait. The current purpose variable is checked when the callback functions are invoked. The thread is inserted into the ready queue only if this field is set to the correct value in the callback function. It should be set to READING if handle_input is called and WRITING if handle_output is called.

fd	File descriptor associated with this iohandler object.
tid	thread id of the thread that created an iohandler object.
count	Number of times this iohandler object is registered with Reactor
current purpose	If thread with id, tid, is waiting for the I/O, this field specifies the purpose of the wait. It can be READING or WRITING.

Table 4.2.1.1 Private data of an iohandler object.

4.2.2 Initialization of Library Level Data Structures

During initialization of the BThreads library, initialization of the following data structures is done:

- Main thread library level data structure.
- The Ready queue, the termination queue, and the sigwait queue.
- The Reactor that handles I/O requests of the threads.
- The timer that generates the scheduler timer signal.
- The signal handler associated with the scheduler timer signal.
- The _init method for initializing the shared object is implemented in the BThreads library. So the BThreads library is initialized when the process starts running.

4.2.3 User Level Data Structures and Interface

The BThreads API provides the user with part of the IEEE 1003.1c POSIX interface [16, 20].

4.3 Pushing a function call onto a thread stack

In the BThreads library, the capability of calling a function from an arbitrary point in the thread stack is needed. For asynchronous cancellation, when a thread that received a cancellation request is resumed, it should be terminated immediately. For realizing this goal, a function call to thread_exit must be inserted on the thread stack. This section deals with how to insert a function call on the thread stack so that when a thread starts running, the specified function is called and on return from the function, execution continues according to the original flow. For implementing this, the processor hardware registers ESP (the stack pointer) and EIP (the instruction pointer) need to be modified, and architecture dependent assembly code has to be written. Since these are dependent on the processor, our implementation is tied to the processor architecture. In the present case, the implementation was done specifically for the x86 architecture. The description of the x86 processor architecture can be found in [22].

The current instruction pointer (the EIP register) must be pushed onto the stack so that on return from the inserted C function, an execution continues from the original point in the code where the C function call was inserted. When the C function is called, it may change the general-purpose registers and the flags register. Hence these registers need to be saved on the stack before calling the C function and restored on return from it. In addition to these, the C function to call, the arguments to the function and the number of bytes arguments occupy should be pushed onto the stack. Finally the stack frame is as shown in Figure 4.3.1.

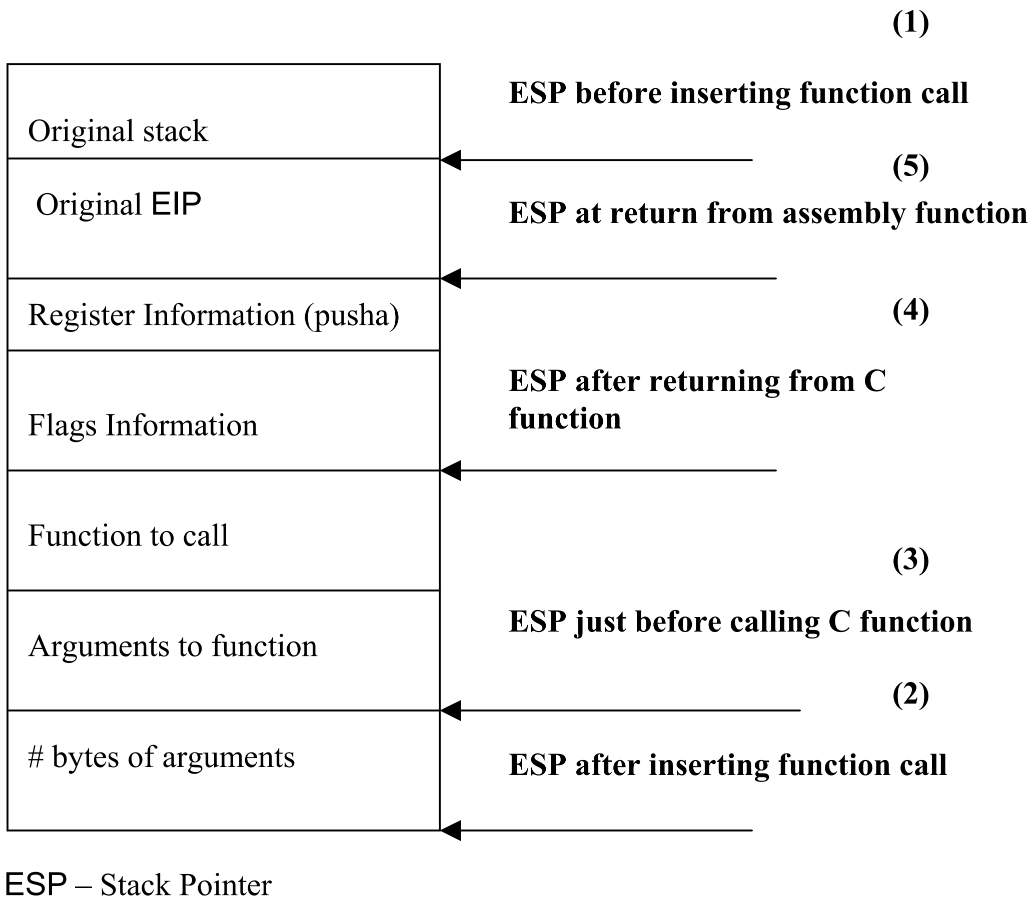

(1)
ESP before inserting function call
Original stack
(5)
Original EIP
ESP at return from assembly function
Register Information (pusha)
(4)
ESP after returning from C function
Flags Information
Function to call
(3)
ESP just before calling C function
Arguments to function
(2)
# bytes of arguments
ESP after inserting function call
ESP – Stack Pointer

Figure 4.3.1 Contents of the stack frame before and after inserting a function call on the thread stack

It is important to note that the function call is inserted inline in the original stack i.e. the EBP register (the base pointer) is not changed. In order to support an insertion of the C function with an arbitrary number of arguments, an assembly wrapper function is needed. This wrapper function calls the actual C function. The EIP register is changed so that the assembly wrapper function is called first.

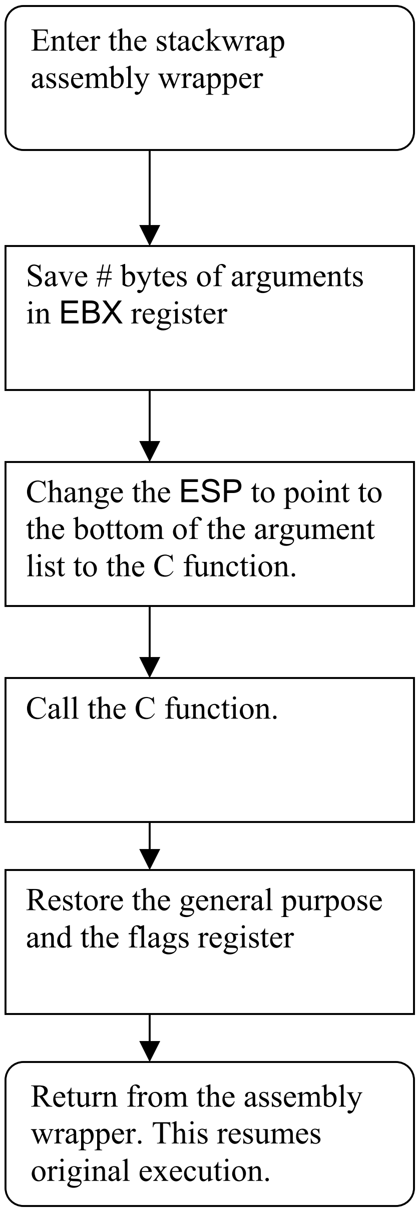

Enter the stackwrap assembly wrapper
Save # bytes of arguments in EBX register
Change the ESP to point to the bottom of the argument list to the C function.
Call the C function.
Restore the general purpose and the flags register
Return from the assembly wrapper. This resumes original execution.

Figure 4.3.2: Implementation of assembly wrapper function.

Implementation of the assembly wrapper function is shown in Figure 4.3.2. This function is implemented in assembly language.

4.4 Thread Creation and Destruction

For creating user space thread's, standard interfaces that are present in the glibc library, like the ucontext and jmpbuf functions can be used. As discussed in section 4.1, the BThreads library uses the ucontext API. Figure 4.4.1 shows the algorithms for creation and termination of a thread. For creating a new thread of execution, the makecontext function of the ucontext API can be used. This creates a user level context that can be run by calling the setcontext or swapcontext functions.

One important point regarding the creation of a thread is that memory for the TCB associated with a thread is allocated on the thread's stack. So when a thread exits and its stack is deallocated, memory for the TCB associated with the thread is automatically deallocated.

When creating a new thread, the stack for the thread has to be allocated. This is generally allocated on the process's heap. As discussed earlier, the BThreads API uses the guard page to detect thread stack overflow. The address where the guard page should be placed depends on the direction of growth of the stack. For processors like the x86 that have the stack growing in a downward direction, the guard page has to be appended at the lower end of the stack. For other processor architectures that have the stack growing in an upward direction, the guard page has to be appended at the upper end of the stack. The BThreads library takes care to append the guard page at the appropriate end of the stack depending on the direction of growth of the stack.

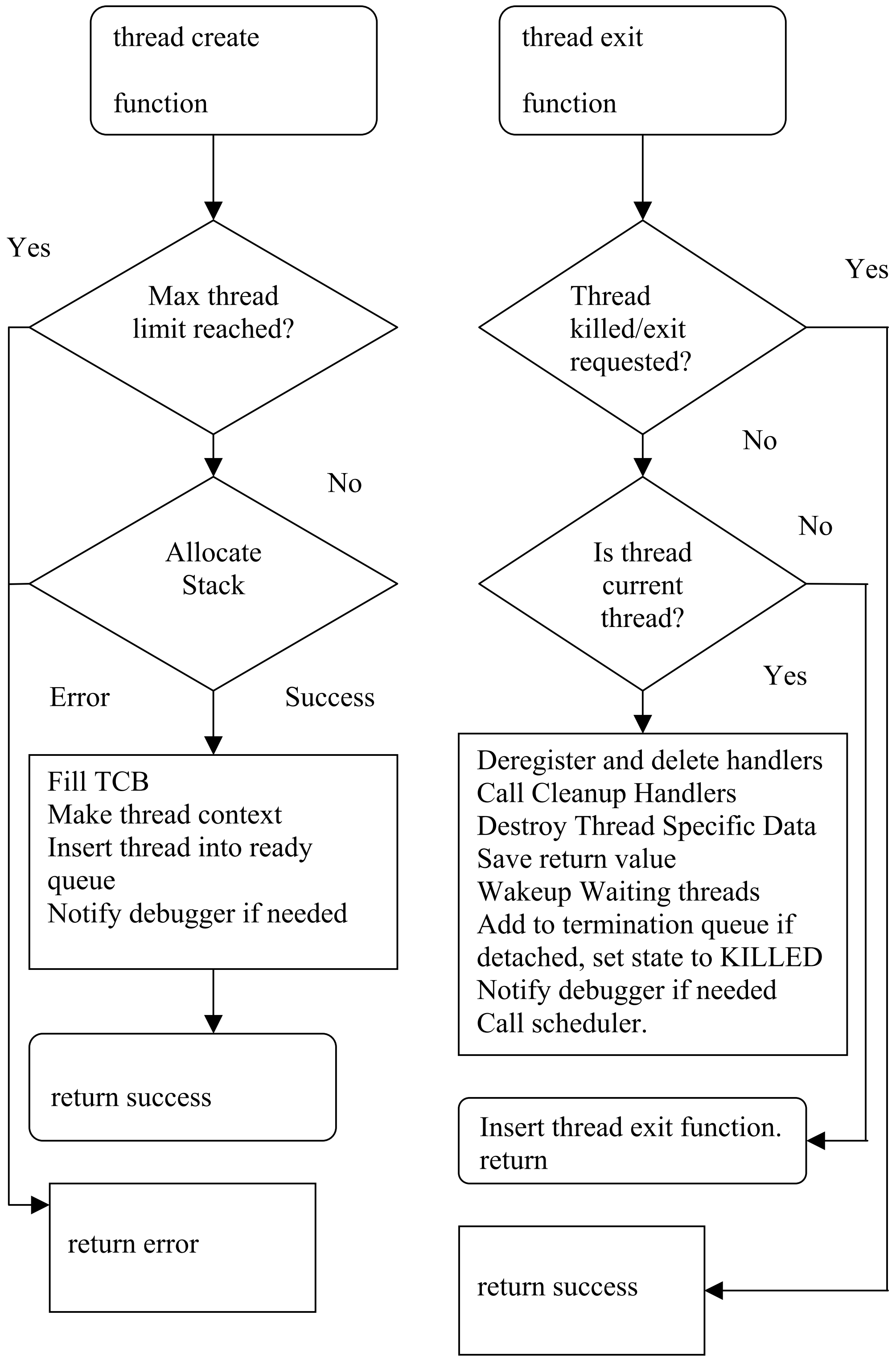

thread create function
Max thread limit reached?
Yes
No
Allocate Stack
Error
Success
Fill TCB
Make thread context
Insert thread into ready queue
Notify debugger if needed
return success
return error
thread exit function
Thread killed/exit requested?
Yes
No
Is thread current thread?
No
Yes
Deregister and delete handlers
Call Cleanup Handlers
Destroy Thread Specific Data
Save return value
Wakeup Waiting threads
Add to termination queue if detached, set state to KILLED
Notify debugger if needed
Call scheduler.
Insert thread exit function. return
return success
a. thread creation
b. thread exit

Figure 4.4.1 Algorithms for thread creation and thread exit routines

When the thread_exit routine is called and the specified thread and current thread are different, a function call to thread_exit is inserted on the thread stack of the specified thread so that it is terminated as soon as it starts running. A thread cannot be terminated from any other thread's stack since the cleanup functions registered by a thread should be run on that thread's stack.

4.5 Thread Scheduling

As discussed in section 3.3, the BThreads library uses a preemptive Round Robin (RR) scheduling policy. For realizing preemption, the interval timers present in the LINUX system can be used. There are three kinds of timers that can be set using the setitimer function. These timers decrement in different time domains.

- When itimer_real is used, the timer is decremented in the real time and SIGALRM is delivered upon the expiration of the timer.
- When itimer_virtual is used, the timer is decremented only when the process executes in user space and SIGVTALRM is delivered upon expiration of the timer.
- When itimer_prof is used, the timer is decremented both when the process executes in user space and when the system executes on behalf of the process in kernel space. SIGPROF is delivered upon expiration of the timer.

In the implementation of the BThreads library, itimer_prof was used for generating the scheduler timer signal once every RR quantum. When the signal is delivered, the scheduling function is invoked from the signal handler. The scheduling function internally uses the swapcontext function for performing the thread context switching and this function is not asynch-safe; i.e., it cannot be called from the signal handler stack. To overcome this problem, the signal handler code is executed in an alternative stack frame created using the sigaltstack function and the stack frame and the instruction pointer of the thread that received the scheduling timer signal are modified from there so that the scheduling function is called automatically on return from the signal handling code. This is implemented using the support for inserting a function call on a thread stack as described in section 4.3.

Another important point to note is that the Reactor is used for generating the scheduling timer signal at periodic intervals. So all the timer expiration events are captured initially by the Reactor and then delivered to the currently running thread in the BThreads library. The Reactor provides an interface that can be used for registering timer expiration events at specific times. Hence the expiration of timers can be captured in the Reactor. Another important point to note is that BThreads library is part of BERT infrastructure that provides a generic interface on top of which several software modules can be built like BThreads, KUDOS [12] and asynchronous I/O support.

4.6 Signals

Signal support for BThreads was discussed in section 3.6. In section 3.6.1, delivery of signals in the BThreads library was discussed. In that section, it was mentioned that just before scheduling a thread, a check is performed if there are any pending signals present that are not blocked by the thread that is going to be scheduled. If there are any such signals, they should be delivered to the thread once it starts running.

For signals of the ASGOT type, signals in the pending set of the thread that are unblocked are included in raise signal set, the set of signals that need to be delivered to the thread**.** After completing this operation, raise signal set has all the signals that can be delivered to the thread. A function call to raise_signals is inserted on the execution stack of the thread. The argument to this function is raise signal set. The Raise_signals function raises signals that are present in raise signal set. Insertion of this function on the execution stack of thread was discussed in section 4.3.

In the swapcontext function, the signal mask of the new context that is going is set before it actually starts running. So signals of the EGAS type would be delivered on the old thread's stack. To prevent this, care has to be taken so that they would be delivered only on the new thread's stack. Signals that are not masked by the thread to be scheduled and that are part of the pending set of the process are included in the block signal set. The signal mask of the thread to be scheduled is changed to mask the signals in the block signal set. This ensures that these signals are not delivered in the midst of the thread context switching operation. But since these signals are not blocked in the thread that is going to run next, a function call to unblock all these signals is inserted on the thread stack of the thread that is going to be scheduled. These signals are delivered only when the new thread starts running. Hence they are delivered on the new thread's stack.

4.7 Thread Cancellation

Design for supporting cancellation in the BThreads library has already been discussed in section 3.7. In this section we discuss any specific support that needs to be provided for implementing the design in section 3.7.

When a thread sends a cancellation request to another thread, which has asynchronous cancellation type, the other thread should be terminated as soon as it starts running. It is important to understand that the thread_exit routine should be called from that thread's stack since the cleanup handlers need to be executed only on *that thread's* stack. For this purpose, a function call to the thread_exit with the argument THREAD_CANCELED is inserted on the stack of the thread that got the cancellation request. This value is returned when another thread joins on termination of this thread. Insertion of a function call on a thread stack was discussed in section 4.3.

4.8 Thread-Specific Data (TSD)

TSD is useful when threads need to store global data that can have different values in different threads of execution. This is not possible directly with threads since they share the address space. Using TSD, we can set and retrieve data from any point in the execution of a thread and the TSD can have different values in different threads. Every thread has a private memory block, the TSD area. The TSD area is an array of type void * pointers with the TSD keys being index into this array. For a given key, the value of the TSD, the void* pointer at that key, can be different for different threads. When a thread is created, initially NULL pointers are associated with all keys for this thread. The TSD can be used only after creating keys.

There is a limitation on the maximum number of keys that can be created in the BThreads library: thread_keys_max. In the BThreads implementation, the TSD has been implemented as a sparse array to make efficient use of memory. The structure of the sparse array is shown in figure 4.8.1 (size of inner array 32):

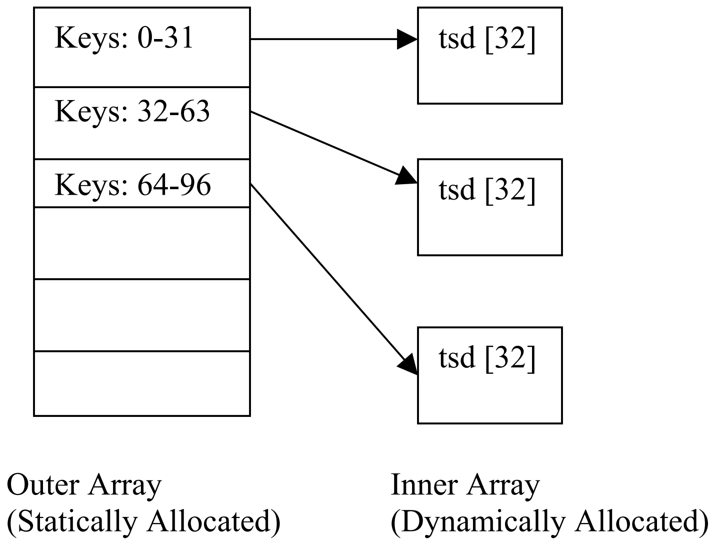


Figure 4.8.1: Sparse Array Implementation for the TSD in the BThreads.

As shown, the TSD has been implemented as a two level array where the outer array is statically allocated with a fixed size of thread_key1stlevel_size. The outer array is a void ** pointer that points to the inner array, which is an array of void * pointers. The inner array is created only on demand i.e. is dynamically allocated and is of fixed size: thread_key_2ndlevel_size. This value should not be too large. A key can be located in this two level array with an outer array index key/thread_key2ndlevel_size and an inner array index key mod thread_key2ndlevel_size. This improves the memory efficiency since we are reducing the necessity to allocate memory for holding all keys unnecessarily to some extent without losing the advantage of using arrays for faster access. Whenever the TSD is to be set at a specified key, a check is performed if the memory for the inner array at the corresponding outer index has been allocated. If not, it is allocated and the TSD at an appropriate inner index is set to the specified value.

As there is a limitation on the maximum number of keys available, an array of key data structures that holds the information about all the keys can be created statically with size thread_keys_max. This data structure has two elements: a field that specifies the validity of the key, whether it is in use or not and a field that points to the destructor function specified at key creation time. When a key is created, an unused element in this static array is returned as the key id and the user-specified destructor function is set. When a key is deleted, the validity field at the corresponding index is set to INVALID and the destructor function is set to NULL.

Then the TSD value at this key is set to NULL in all the threads. The destructor function is not called for the non-null TSD values when deleting a key.

_thread_destroy_specific:

This method is called in the thread_exit function to destroy the TSD associated with this thread. Looping through all the possible keys, a check is performed if the TSD and the destructor function are not null for this key. If they are not null, the destructor function is called with the TSD value as an argument. The destructor function itself may again associate non-null values with some of the keys. To account for this, looping is repeated over all the keys for some fixed number of iterations. Once this is done, the memory that was allocated for maintaining the TSD data for this thread, i.e. memory for the inner array in the two-dimensional sparse array, is deallocated.

4.9 Debugger support

The purpose of Thread Debug Interface [23] is to provide an interface by which GDB can get information about threads in the BThreads library. Figure 4.9.1 shows how the TDI provides a necessary interface to enable the interaction between GDB and a multi-threaded program.

A program that is being debugged will run as an inferior process under control of the superior process, GDB. GDB should be provided with an interface that allows it to access and modify the data structures in the inferior process. The TDI provides this interface to GDB. It has addresses of various symbols in the BThreads library that might be of interest to GDB. These addresses are initialized when a new *thread agent* is created. The thread agent is the handle for the process as a whole that has multiple threads of execution.

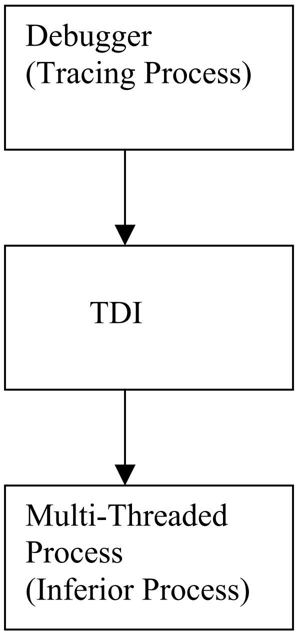


Figure 4.9.1: Interaction between GDB and a BThreads program.

The TDI provides methods that allow GDB to do the following:

Event Enabling and Reporting:

GDB can enable or disable generation of *specific* events on a per-thread basis. Reporting of a *generic* event can be disabled or enabled for a specific thread. If disabled, no event is ever reported. If enabled, then a check is performed if the specific event has been enabled either in the *global mask* of enabled events or in the mask of events enabled for this thread. If the specific event is enabled in one of these masks, the event is reported. As discussed in section 4.2.1, the following three events are generally reported by the BThreads library to GDB: thread creation, thread exit and thread reaping. GDB is notified of these events by calling a method that does nothing. But the event notification happens as GDB sets a breakpoint at this method. When this dummy method is called from the BThreads library, a SIGTRAP signal is generated. This signal is caught by GDB and depending on the address where this event was generated, GDB detects a specific event. Thread creation, death and reaping events have different dummy methods associated with them and hence the address at which the SIGTRAP is caught when these events are generated is different.

Examining the register state:

GDB must get the correct state of the processor registers like the general purpose registers and the floating point registers for different threads so that it can accurately display information about them when a command like *info threads* or *bt* is given at the GDB command prompt. The register information can be retrieved from the machine context information present in the TCB. Methods for setting the CPU registers should also be provided in the TDI. One important advantage when debugging a user-level library like BThreads is that the state of killed threads whose stack is not yet deallocated can be observed, as the complete machine context information is stored at user level. This is not possible with the KLMT library where the PCB of the thread is deallocated as soon as it exits.

Invoke Callback function over all active threads:

The TDI provides the ability to call a specific function by looping over all the threads that are active, when the specified conditions related to the thread state and the priority are met. This is used by GDB when the *info* command is issued.

List of Mutexes and Condition Variables:

The TDI provides the ability to return the information associated with the various mutex and condition variables that are currently active in the BThreads program being debugged. This can be used by GDB for constructing waiting flow graphs for deadlock detection.

Get thread information:

The TDI provides the capability to get the following information about a thread: the state of the thread, the start routine the thread is supposed to execute and whether reporting of generic events is enabled or disabled for the thread.

4.9.1 Testing concurrency scenarios

When the scheduler timer signal is turned off, scheduling can take place under the control of GDB. The BThreads library provides GDB with an interface to switch to an arbitrary thread. It is then possible to interleave the instructions from the different threads in an arbitrary way and thus detect potential synchronization problems like deadlocks and race conditions.

4.9.2 Recording a program execution

Recording of a program execution can be done that can later be used by GDB for replaying the program. During recording, it is sufficient to record only that information which disrupts the sequential flow of execution and introduces nondeterminacy in a program. These points in the BThreads program are the scheduler timer signal (the SIGPROF signal), signals, and I/O completion.

Most accurate recording of a BThreads program execution will be done when it is run directly than when running under GDB as any side effects that might happen due to running under control of GDB are removed.

Delivery of the scheduler timer signal (SIGPROF signal) can be recorded from the signal handler so that it can be used during replay. During replay, preemption is turned off and GDB synthetically generates and delivers the signal at the points in the code where the signal was delivered during execution of the original program.

For delivering signals, only signals of the EGAS type need to be recorded during the original execution of the program. These signals are later delivered from the debugger. Signals of the ASGOT type are generated automatically since they occur due to calls to the thread_kill function.

I/O system calls are one of the sources of asynchrony. They can affect the reproducibility of the program from one run to another. The reason for this is that when an I/O event becomes ready can change from one execution to another. This makes replay of I/O difficult.

Recording concurrency information need not be done at other points in the code where the scheduler might be invoked. These include: thread_mutex_lock, thread_mutex_unlock, thread_cond_wait, thread_yield, thread_do_exit. This is because if the multi-threading library had no events that can cause asynchrony (signals, scheduling preemption and I/O calls), then programs written using such a library are completely reproducible from one run to another. However, due to asynchrony introduced by the scheduler-timer signal, signals and I/O, reproducibility is hampered. It is thus sufficient if the concurrency recording is done just at these points.

4.10 Limitations

The following functions have not been implemented in the BThreads library:

- Priority based scheduling.
- The timed variants of condition variables and the mutexes:
    - thread_timed_condwait.
    - thread_mutex_timedlock. This is not required by POSIX.
- Thread barrier functions. This is not required by POSIX.
- Thread read/write locks.
- Process shared and process private attributes for mutexes, condition variables and read/write locks.

- Concurrency level, which specifies number of kernel level threads on top of which many user level threads are multiplexed. This is of significance only in a many-to-many model.

Other limitations are:

- Works only on the x86 architecture. However, it can be ported easily to other architectures
- Low-level terminal I/O is not supported

Run time limitations in the BThreads library:

- Minimum Thread Stack Size. The minimum thread stack size is 4 stack pages.
- When using unbuffered I/O streams (such as stderr), the stack size has to be increased to at least 10 pages since stderr internally allocates a buffer of 8192 bytes on the thread stack. Similarly care has to be taken so that thread stack overflow doesn't occur due to allocation of big temporary buffers.

5. Test Scenarios.

This section describes how testing of the BThreads library was done. Testing was done to verify correct operation of the library and its POSIX compliance. Basic performance testing was also done to compare the performance of BThreads with the Linux Threads implementation. As mentioned earlier, the BThreads library was built to provide better debugging capabilities. It provides ability to test and reproduce different concurrency scenarios. Basic tests verifying these capabilities of the library were also performed.

5.1 Correctness Testing

This section describes correctness testing of the BThreads library. The correctness testing was carried out in two phases. White box testing was done as the library was developed. Black box testing was done once the library was fully developed. For black box testing, multi-threading applications based on existing POSIX compliant thread library implementations were tested.

a. White-Box testing

As the BThreads library was developed, on a module-by-module basis testing of the code was done. This can be considered to be basic white-box testing since testing was done keeping in mind the internal structure of library code that implements BThreads API calls.

b. Black-Box testing and POSIX 1003.1c compliance.

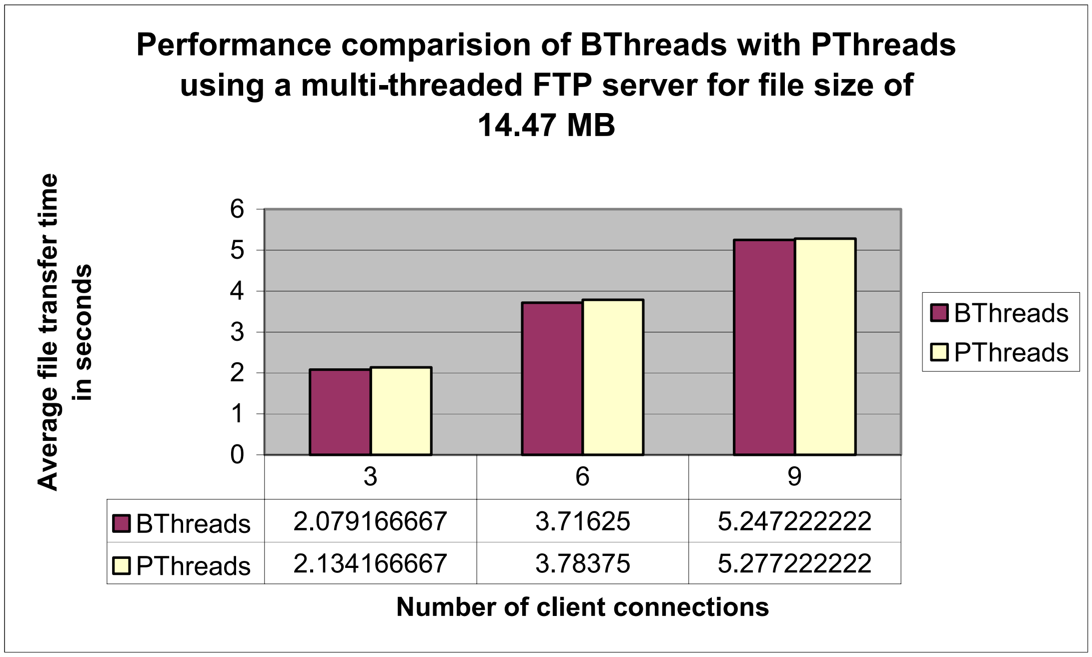
POSIX IEEE 1003.1c compliance for the library was verified by running a set of test programs from the Linux Threads library based on a one-to-one implementation model. The Linux Threads library is also POSIX compliant. Hence test programs should work correctly when linked with the BThreads library. It was verified that they do work correctly. The following features were tested:

- Basic thread creation and destruction.
- Classic Producer-Consumer problem using mutexes and condition variables.
- Multi-thread searching using mutexes, cancellation and cleanup handling. A random number is generated and multiple threads search for it concurrently. The first thread that finds the number cancels the other threads and the program exits.
- Thread specific data and thread_once initialization functions were tested. Different threads accumulate the same string in their own thread specific buffer concurrently and print them.
- Concurrent matrix multiplication of NxN matrices.

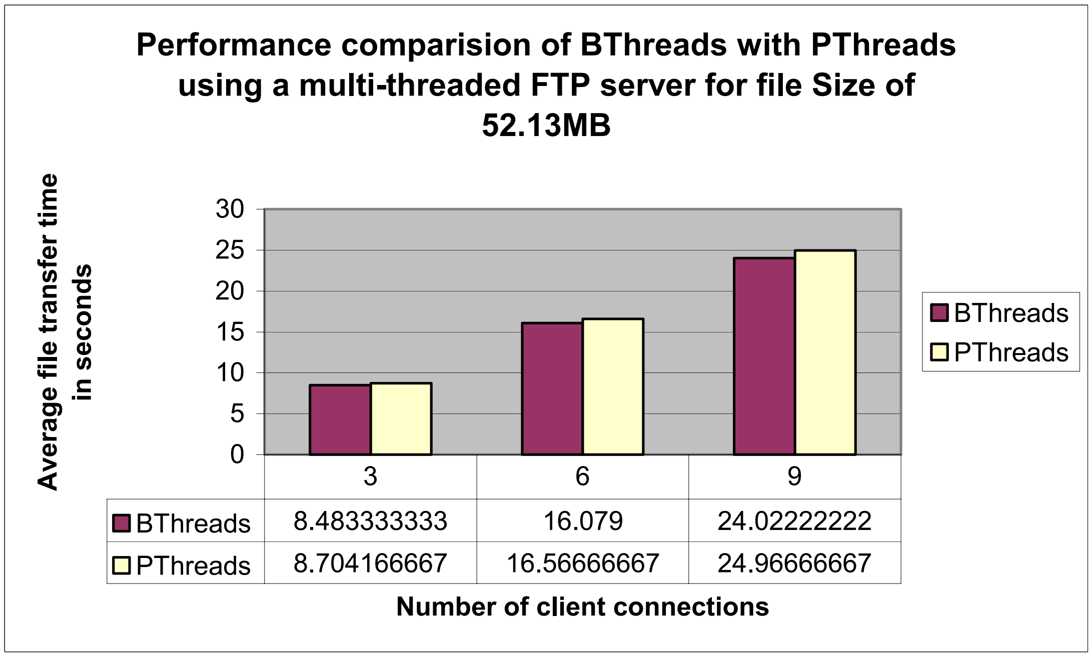
5.2 Performance Testing

The correctness and performance testing of the BThreads library was done using a multi-threaded FTP server. A multi-threaded FTP server implementation based on the Linux Threads library was taken and by minimal changes to the code, a multi-threaded version of the FTP server using the BThreads library was built. The performance of this version of the library was studied by transferring files of 14MB, 52 MB in a private network of four nodes. One node was a server and other three nodes were clients and a private network was formed among these four nodes. This network setup ensures stable network conditions. Also the process load on the machines was minimal.

As shown in figure 5.2.1, tests were performed for varying number of clients that are connected simultaneously to the ftp server and files of size 14 MB and 52MB were transferred between all the clients and the server. The file transfer time was recorded in all cases and the average values of file transfer time are plotted for the two thread library implementations for different number of client connections. It can be seen from figure 5.2.1 that both implementations take almost the same amount of time for file transfer for a file size of 14 MB but for a file size of 52 MB, there is an improvement of about 1 sec when nine clients are simultaneously connected to the thread library.



	3	6	9
■ BThreads	2.079166667	3.71625	5.247222222
□ PThreads	2.134166667	3.78375	5.277222222



	3	6	9
■ BThreads	8.483333333	16.079	24.02222222
□ PThreads	8.704166667	16.56666667	24.96666667

Figure 5.2.1 Performance comparison of the BThreads library with the PThreads library using a multi-threaded ftp server

Number of Client Connections	Mean file transfer time in seconds		Standard deviation of file transfer in seconds		Confidence Interval of file transfer in seconds at 95% level	
	BThreads	PThreads	BThreads	PThreads	BThreads	PThreads
3	2.079167	2.134167	0.080202	0.111467	0.078597	0.109236
6	3.71625	3.78375	0.22717	0.15553	0.222622	0.152416
9	5.247222	5.277222	0.195567	0.514588	0.191652	0.504286

Table 5.2.1 Table listing the confidence intervals for the average file transfer time. 95% confidence level is assumed. File size is 14.477318 MB

Number of Client Connections	Mean file transfer time in seconds		Standard deviation of file transfer in seconds		Confidence Interval of file transfer in seconds at 95% level	
	BThreads	PThreads	BThreads	PThreads	BThreads	Pthreads
3	8.483333	8.704167	0.047376	0.127145	0.046427	0.124599
6	16.079	16.56667	0.411764	0.160439	0.40352	0.157227
9	24.02222	24.96667	0.28153	0.296343	0.275894	0.29041

Table 5.2.2 Table listing the confidence intervals for the average file transfer time. 95% confidence level is assumed. File size is 52.132352 MB

Tables 5.2.1 and 5.2.2 show the confidence intervals for the mean file transfer time for file of size 14 MB and 52.132352 MB respectively. A confidence level of 95% has been assumed and as shown in the table the actual transfer times are within 0.4 seconds in the worst case for the BThreads library and within 0.5 seconds in the worst case for the Pthreads library. So the measurements taken give an accurate representation of average transfer times.

5.3 Testing and reproducing concurrency scenarios

As mentioned above, the BThreads library was designed and implemented for providing better debugging capabilities. The GDB debugger was modified [13] to support debugging of programs written using the BThreads library. In this section, a few test cases are explained that give a basic idea of how to debug a BThreads program.

5.3.1 Testing concurrency Scenarios

By testing concurrency scenarios, we mean that the user can test different possible thread interleaving sequences and detect potential deadlocks and race conditions present in the code that become visible only for a particular thread interleaving.

The following are four necessary and sufficient conditions for deadlock to occur [24]:

Mutual Exclusion:

Only one thread can access the resource at a time.

Hold and Wait:

A thread holds a resource and waits to acquire additional resources held by other threads.

No preemption:

A resource can be released only by the thread holding it. It cannot be forced to give up the resource by another thread.

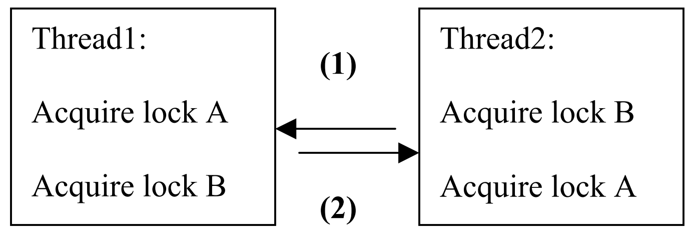
Circular Wait:

A set of threads: T_0, T_1, …, T_{N-1} are in a circular wait; i.e. T_0 is waiting for a resource held by T_1 , T_1 is waiting for a resource held by T_1 … and T_{N-1} is waiting for a resource held by T_0 .

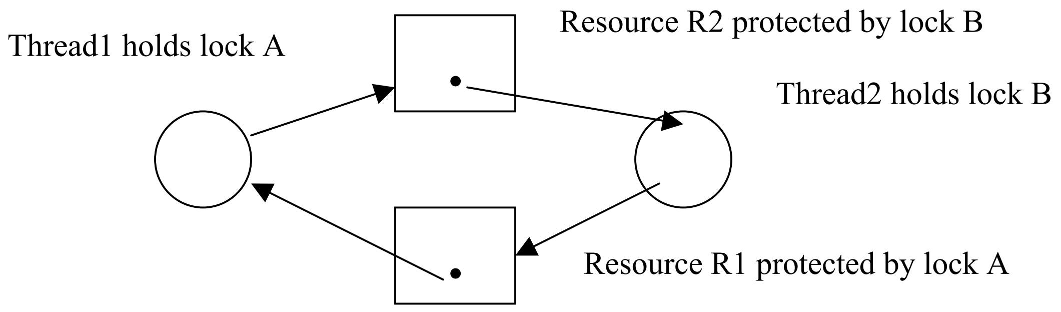
a. Test case 1

In the simple case, we can consider the scenario where we have two threads that try to acquire two locks in different orders. Lock A is used to protect resource R1 and lock B is used to protect resource R2. Thread1 tries to acquire lock A and then lock B. Thread2 tries to acquire lock B and then lock A. This *may not* always lead to a deadlock condition. However, if context switching happens at an inopportune moment i.e. when thread1 acquires lock A, but before it acquires lock B, a deadlock can occur. Thread2 acquires lock B and tries to acquire lock A that is held by thread1 and a deadlock occurs. The situation is illustrated in figure 5.3.1.1.

A test program was written using the BThreads library to implement the above example and context switching was forced at these points in the code by calling the *switch_to_thread* function from the debugger to switch to the other thread. As predicted, we could simulate the deadlock condition. While doing this testing, the scheduling signal was disabled so that context switches occur only due to calls to the *switch_to_thread* function from GDB.



i. How thread context switch order can lead to deadlock


Thread1 holds lock A
Resource R2 protected by lock B
Thread2 holds lock B
Resource R1 protected by lock A

ii. Resource allocation graph for the test case

Figure 5.3.1.1 Figure illustrating a possible deadlock scenario.

b. Test case 2 (Dining Philosophers problem)

The classic dining philosophers problem was implemented using the BThreads library. In the dining philosopher's problem, every philosopher tries to acquire the two adjacent forks and then eats for a random amount of time. After eating, a philosopher releases forks, thinks for a random amount of time and then tries to acquire the forks again. It is clear that if all the philosophers try to acquire either right fork or left fork at the *same time*, there is a *deadlock*.

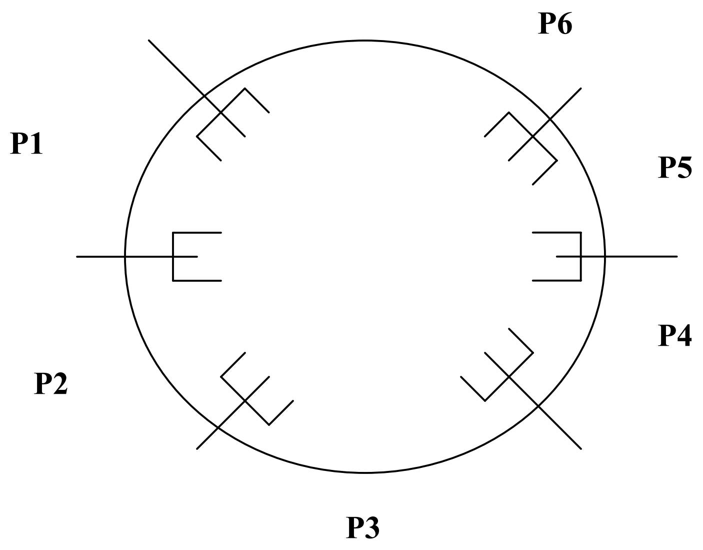


Figure 5.3.1.2 Figure shows the Dining Philosophers problem

The algorithm for implementing the dining philosophers problem is shown in figure 5.3.1.3. This algorithm *has* the possibility of causing a deadlock. When all the philosophers try to acquire the right fork at the same time, deadlock occurs. Deadlock many not always occur since the random amount of waiting may ensure that the above condition may rarely occur. In thread terms, this situation occurs when context switching occurs among the threads just after acquiring a particular fork: the right fork or the left fork. This was verified by forcing context switches at these points by calling the *switch_to_thread* function from the debugger to switch to the other thread and a deadlock was simulated.

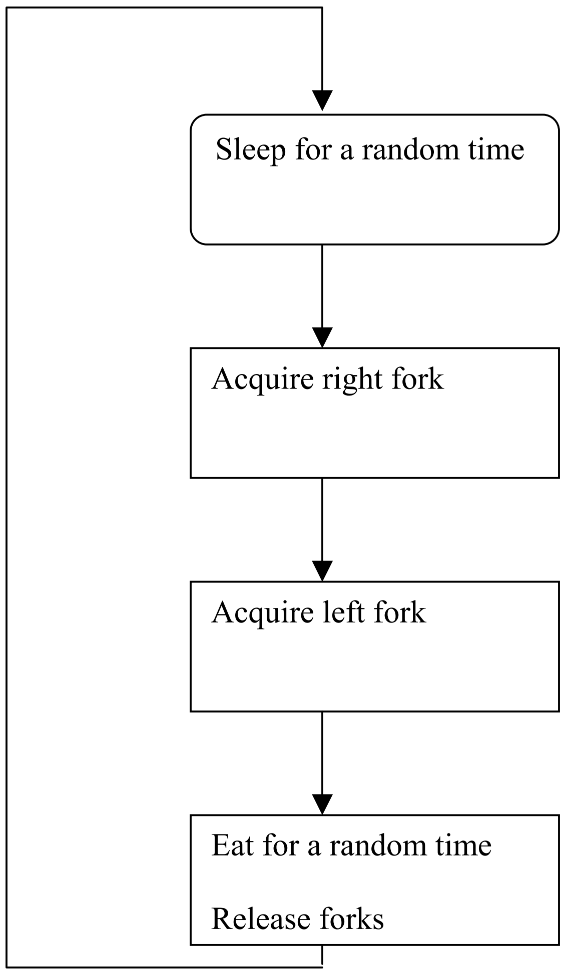


Figure 5.3.1.3 Algorithm for implementing the dining philosophers problem in the BThreads library.

5.3.2 Reproducing concurrency Scenarios

In test cases 1 and 2 described in the previous section, it was mentioned that a deadlock situation could be created by forcing context switches at inopportune moments. The context switches were forced from GDB by calling the *switch_to_thread* function provided by the thread library that allows a context switch to take place at any time. The deadlock situation also occurs rarely when the context switching actually happens at these moments. The concurrency recording capability that is built into the BThreads library can be used to record a specific sequence of context switching that causes a deadlock. All this information was recorded and the original execution was replayed later using the GDB debugger. So the actual reason for deadlock can be established by replaying an execution sequence.

6.Conclusions and Future Work

A preemptive User Level Multi-Threading library has been built that provides the ability to test and reproduce various concurrency scenarios. This ability to reproduce is provided by capturing all the events that affect concurrency using the event-driven framework provided by the BERT Interface [12]. This library is a part of the BERT Project that aims to provide reproducibility for concurrent software systems. By using the BERT interface as the underlying building block, different kinds of concurrent software can be easily integrated and controlled. A POSIX Compliant Thread Debugger Interface was built that can be used by GDB to control multi-threaded programs and it was demonstrated that GDB can use this for debugging BThreads programs. POSIX IEEE 1003.1c compliance of the library was also verified. Basic performance testing of the multi-threading library was done that showed that the performance is comparable to the Linux Threads implementation. Testing and recording of few concurrency scenarios using BThreads library was demonstrated. The recorded information was replayed using the GDB debugger for the BThreads library [13] and hence we could verify that a particular execution sequence can be replayed.

Possible future extensions for this work include identifying all the system calls that can affect concurrency other than normal blocking I/O system calls and accounting for them. Some parts of the POSIX 1003.1c interface need to be implemented. An interface should be provided to the user so that the scheduling algorithm can be fined tuned according to the current state of the application. This could lead to significant performance gains. The library also has been built only for the x86 architecture and it needs to be ported to other architectures like Solaris, Irix, and Windows NT/2000/XP. Dynamic linker tricks can be used so that executable programs that use other POSIX compliant thread libraries can be debugged by mapping all the thread API calls to this library at run time. This removes the necessity of recompiling programs. This is useful especially when debugging proprietary software that cannot be recompiled. An application that is built on an event-driven architecture can be smoothly transitioned to use concurrency using the BThreads library as BThreads by itself is built on top of an event-driven architecture.

Bibliography

[1] Andrew S.Tanenbaum and Maarten Van Steen. *Distributed Systems Principles and Paradigms*, chapter 3.Processes, pages 141-144. Prentice Hall, 2002.

[2] Andrew S.Tanenbaum and Maarten Van Steen. *Distributed Systems Principles and Paradigms*, chapter 3.Processes, pages 135-141. Prentice Hall, 2002.

[3] Douglas C.Schmidt. Reactor: An object behavioral pattern for concurrent event demultiplexing and event handler dispatching. In *Proceedings of the 1st Pattern Languages of Programs Conference*, Aug 1994.

[4] Ralf. S. Engelschall. Portable Multithreading The signal stack trick for user-space thread creation. In *Proceedings of 2000 USENIX Annual Technical Conference, June 18-23, 2000.*

[5] IBM Corporation. *Next Generation POSIX Threading*, November 2002.

[6] The University of Arizona, Tucson. *The Filaments Research Group.*

[7] Vincent W. Freeh, David K. Lowenthal, and Gregory R. Andrews. Distributed Filaments: Efficient Fine-Grain Parallelism on a Cluster of Workstations. In *First Symposium on Operating Systems Design and Implementation*, pages 201-212, Monterey, CA, Nov 14-17, 1994. USENIX.

[8] Massachusetts Institute of Technology. *The Cilk Project.*

[9] Andrew F. Stark. Debugging Multithreaded programs that Incorporate User-Level Locking. Master‘s thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, Jun 1998.

[10] Frank Mueller. *FSU POSIX Threads (Pthreads).* North Carolina State

University.

[11] Xavier Leroy. *The LinuxThreads Library***.**

[12] Rajukumar Girimaji. Reactor, a software pattern for building, simulating and debugging distributed and concurrent systems. Master's thesis, The University of Kansas, 2002.

[13] Satyavathi Malladi. A thread debugger for replaying concurrency scenarios. Master's thesis, The University of Kansas, 2003.

[14] Douglas C. Schmidt. *The ADAPTIVE Communication Environment (ACE) Project.* Washington University in St. Louis The Department of Computer Science and Engineering.

[15] Douglas C. Schmidt. An OO encapsulation of lightweight OS concurrency mechanisms in the ACE toolkit. Technical Report WUCS-95-31, Washington University in St. Louis The Department of Computer Science and Engineering, 1995.

[16] *Information Technology – Portable Operating System Interface (POSIX) -Part 1: System Application Program Interface (API) [C Language],* chapter 16: Thread Management. The Institute of Electrical and Electronics Engineers, Inc., July 1996.

[17] Intel Corporation. *Intel Architecture Software Developer's Manual, Volume 2: Instruction Set Reference.*

[18] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. *Operating System Concepts,* chapter 7: Process Synchronization, pages 201-211. John Wiley &

Sons, Inc., Jun 15 2001

[19] Dr. C. –K. Shene. *Multithreaded Programming with ThreadMentor: A tutorial.* Department of Computer Science Michigan Technological University, Apr. 2002.

[20] *Information Technology – Portable Operating System Interface (POSIX) -Part 1: System Application Program Interface (API) [C Language],* chapter 3.3: Signals, pages 72-73. The Institute of Electrical and Electronics Engineers, Inc., July 1996.

[21] *General Programming Concepts: Writing and Debugging Programs,* chapter 9: Parallel Programming. September 1999.

[22] Intel Corporation. *Intel Architecture Software Developers Manual, Volume 1: Basic Architecture.*

[23] Daniel Schulz. *A Thread Debug Interface (TDI) for implementations of the POSIX Threads (Pthreads) standard*. Humboldt University Berlin Faculty of Mathematics and Natural Sciences II, Department of Computer Science.

[24] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. *Operating System Concepts,* chapter 8: Deadlocks, pages 245-248. John Wiley & Sons, Inc., Jun 15 2001.