Paradyn Parallel Performance Tools

LibThread Programmer's Guide

Release 3.3
January 2002

Paradyn Project
Computer Sciences Department
University of Wisconsin
Madison, WI 53706-1685
paradyn@cs.wisc.edu

Table of Contents

User Guide

1 PRELIMINARIES

The following user guide documents the interface of *libthread*, a non-preemptive user-level thread library written in and exporting a C programming interface, that provides a seamless integration of message-based, file-based and signal-based communication. The libthread tutorial [“Tutorial” on page 18] provides some simple example programs that demonstrate the use of libthread functions.

1.1 Document revision history

- ❏ **v3.2**: minor modifications.
- ❏ **v3.0**: generalized to additionally support WindowsNT threads.
- ❏ **v2.1**: minor change to update list of supported platforms and change to build process.
- ❏ **v1.0**: initial release version.

1.2 Terms

The thread library will be referred to as *libthread*, a *client* is any user (-program) of libthread, and the *user* is the person who will run the client.

1.3 Name-space

With every new package, there is an increasing possibility of a name-space conflict. Towards alleviating this problem somewhat, libthread reserves all names with external linkage that begin with the prefix `thr_`; this includes both code and data. The client may use such names iff the names are of static scope and libthread interface header `thread.h` is not included in the source.

The only functions that the client may use are those described in this document. Using anything else from libthread is an invitation for disaster.

1.4 Types

The following types are defined and exported by libthread. They can be used by clients.

`thread_t`: id for threads, same name-space as unsigned integers.

`thread_key_t`: keys used to access Thread-Specific-Data (TSD).

`tag_t`: type for message tags, same name-space as unsigned integers.

1.5 Error codes

All functions in libthread are either void or return an integer. The integer return values are:

- `THR_ERR` (== -1): indicates an error.
- `THR_OKAY` (== 0): indicates success.

- Any other return (> 0): indicates special return values.

Each thread within libthread keeps track of a local error number indicating the last error that occurred within that thread. The error number itself is not visible to clients, but is used internally by error reporting functions.

1.6 Implementation and Supported Platforms

Libthread is implemented using the `setjmp` and `longjmp` functions in the C library. This was done for simplicity, not for performance or elegance. The libthread sources are also configured to be compiled only on a small number of supported platforms although it is not difficult to port the library to a new operating system and CPU architecture.

The following platforms are currently supported by libthread, where names of the platforms are identical to the *<architecture-vendor-os>* triples used by the GNU configuration system (and reported by the `sysname` script):

```
sparc-sun-solaris2.6
i386-unknown-solaris2.6
i386-unknown-linux2.2
i386-unknown-nt4.0
mips-sgi-irix6.5
rs6000-ibm-aix4.3
```

Limited support for older platforms or platforms where ports are in progress may still be available on request.

Questions, comments, bug-reports, and porting queries and information regarding the thread library should be sent to `paradyn@cs.wisc.edu`.

2 BLOCKING AND NON-BLOCKING FUNCTIONS

As with any multi-threaded system, the libthread functions are divided into two main categories—*BLOCKING* and *NON-BLOCKING*. A BLOCKING function can block the caller and schedule the execution of other threads. A NON-BLOCKING function guarantees that no other thread will execute in the duration of the call. The above guarantee for NON-BLOCKING functions is easy to ensure for a non-preemptive thread package.

Libthread also distinguishes between different types of BLOCKING and NON-BLOCKING functions.

A *vanilla* BLOCKING function is one through which there exists at least one path in which the calling thread is forced to block and other threads made to run. The blocking is part of the semantics of the call. There may exist paths through the BLOCKING functions where a reschedule is not necessary. In such cases, the behavior of the function (whether it blocks or not) depends on the behavior of the optionally BLOCKING functions (described below).

An *optionally* BLOCKING function is one in which the blocking of the calling thread is not a necessary part of the semantics. Nevertheless, the blocking of the calling thread and the resumption of another ready thread will lead to a “fairer” scheduling of the available processor time among all the active threads. Since optional blocking does not affect the correctness of libthread or its clients, it may be selectively enabled or disabled by a compile time flag. If the libthread sources are complied with the preprocessor symbol `ALWAYS_RESCHEDULE` defined, optionally BLOCKING functions will really block (and invoke the libthread scheduler). By default, libthread is compiled to ALWAYS_RESCHEDULE.

An *optionally-optionally* BLOCKING function is one that is the same as the optionally BLOCKING function, except that the function is lightweight. It generally makes little sense to incur the overhead of a thread context switch when any of these functions are called. Optionally-optionally BLOCKING functions can be made to really block (and invoke the scheduler) if the libthread sources are compiled with the preprocessor symbol `ALWAYS_REALLY_RESCHEDULE` defined. By default, libthread is not compiled with `ALWAYS_REALLY_RESCHEDULE`.

Regardless of whether a function is optionally(-optionally) BLOCKING or not, the client should **never** make any assumptions about the exact execution order of the threads in libthread. Any consistent interleaving of the threads should be assumed possible. Relying on specific execution ordering for the correctness of an algorithm is a display of unwarranted chumminess with forces beyond the control of the client and is a sure road to ultimate grief.

A *vacuously* NON-BLOCKING function is one which does not return to the calling thread. `thr_exit()` is the only libthread function of this type. It does not make any sense to classify such function as BLOCKING or NON-BLOCKING since they break the normal control flow in the thread.

Any remaining functions in libthread are NON-BLOCKING. These functions perform trivial, light-weight tasks and it does not matter if they block the caller or not.

3 THE INTERFACE

The description of the interface to libthread is divided into 3 sections: the Thread interface [Section 3.1], the Message interface [Section 3.2], and the Signal interface [Section 3.3]. Each section describes the associated interface functions that libthread exports.

3.1 The thread interface

This subsection describes the functions that allows the client to create and manipulate threads.

3.1.1 Thread creation

```
int thr_create(void* stack, unsigned st_size, void* (*func)(void *),
               void* arg, unsigned thr_flags, thread_t* tidp)
```

The function `thr_create()` creates a new thread. `stack` is the user supplied stack (of size `st_size`) that the thread will use. A null pointer for this parameter indicates that the thread library should allocate its own stack for the thread. Too small a stack size will cause unpredictable results when stack overruns occur. Clients that expect threads to have deeply nested function call trees or deeply recursive functions should allocate their own stacks. The stack allocation by libthread should be sufficient for most threads. The trade-off needs to be evaluated carefully in each case. Currently, libthread performs a stack bounds check whenever the scheduler is entered, and a warning message is printed if the stack appears close to overflow.

Client-allocated stacks becomes the property of libthread. The `stack` cannot be reused even after the death of the thread—since libthread still saves state on this stack.

`func` is the thread function. It is the main function of the new thread and gets invoked as `func(arg)`. if you want to pass a rich argument list to the function, the best thing to do is to define a structure conforming to your argument set and pass a pointer to such `struct` variables via `arg`. The function `func` will return a similar `void*` result.

`thr_flags` is present for compatibility of the interface with Solaris-2 threads. The only value currently supported is `THR_SUSPENDED,` which suspends the newly created thread. The thread will start to run following a `thr_continue()` call. If the `THR_SUSPENDED` flag is specified, the value of `tidp` should not be null since there is no way for the thread creator to know the id of the new thread. If `tidp` is null, a warning is printed but no error is raised—a client that does not keep track of tids can create garbage threads (these threads do not execute and cannot be manipulated by the client). If multiple flags are applicable, the `thr_flags` parameter is the bitwise *OR* of the individual flags.

`tidp` is a tid pointer. If it is non-null, this is where the id of the new thread will be saved. The tid is needed to signal the new thread, wait for it to complete, start it if it is being created in a suspended state and to send messages to it.

This is as good a place as any to mention that the id of a thread should be treated as an opaque type. Thread ids need not be consecutive, begin at zero, or increase or decrease monotonically. The only property of the id of a thread is that it will never change during the lifetime of that thread. Once a thread dies, its id may be reused to identify another thread. The client should **never** use the thread id for anything other than identifying the thread to the libthread functions.

A return value of `THR_OKAY` indicates that a new thread with the given specifications was successfully created. A return value of `THR_ERR` indicates an error.

`thr_create()` is *optionally BLOCKING*.

3.1.2 Thread termination

```
void thr_exit(void* result)
```

The function `thr_exit()` should be called by client threads that wish to terminate. `result` is the return value of this terminating thread. Threads can also terminate via a `return result;` from the starting function of the thread or simply drop off the end of this function (the last of these practices is truly despicable). In the former cases, waiters for this thread will get a legal return value, in the latter, junk (and quite deservedly so).

When a thread exits (or is killed), all file descriptors and signals bound to this thread are automatically unbound (details on binding files and signals to a thread in Section 3.2.1 and Section 3.3.1). Any data available for reading on these file descriptors, or any signals received by the process will not be handled by libthread; when another thread binds the files or signals to its message queues, libthread processing of the data will resume. Any file or signal messages that are currently unprocessed will be **discarded** silently.

`thr_exit()` does not return and is hence *vacuously NON-BLOCKING*.

3.1.3 Thread synchronization

```
int thr_join(thread_t waitee, thread_t* departed, void** resultp)
```

The function `thr_join()` causes the current thread to BLOCK until `waitee` finishes. If `departed` and `resultp` are non-null, the tid of the finishing thread and its result are returned through these pointers.

Specifying a valid `waitee` indicates that the calling thread wants to wait for a specific thread. The calling thread could also wait for anonymous exiting threads (i.e., choose to join with any exiting or exited (and non-reaped) threads) by specifying `waitee` to be 0. It is an error to specify a thread that has already been reaped as the `waitee`. It is silly (and an error) to specify oneself.

A return value of `THR_OKAY` indicates that an appropriate waitee was found. A return value of `THR_ERR` indicates an error.

`thr_join()` is *BLOCKING*.

3.1.4 Thread identification

```
thread_t thr_self(void)
thread_t thr_parent(void)
const char* thr_name(const char* new_name)
```

The function `thr_self()` returns the calling client thread's id. The call is *optionally-optionally BLOCKING*.

The function `thr_parent()` returns the thread id of the calling thread's parent. This call is *optionally-optionally BLOCKING*.

The function `thr_name()` defines a string name for current thread. `new_name` can be any text string and will be used in trace records to identify the thread. In future, it may be possible to use the text name of a thread as a synonym for its numerical id.

Any input name is truncated to some small number of characters (currently 32). If `new_name` is null, the current name is not replaced. When a thread starts off, its default name is `t<tid>`.

The most recently installed valid name of the thread is always returned. Therefore a null `new_name` can be specified to get the current installed name of a thread. The caller must not use the returned pointer to modify the thread name (all modifications to the name of the thread **must** go through this function). If this protocol is not followed, no guarantees are (or can be) made by libthread.

`thr_name()` is *optionally-optionally BLOCKING*.

3.1.5 Thread context switch

```
void thr_yield(void)
```

The function `thr_yield()` causes the calling thread to give up the virtual processor voluntarily and enables the caller to be rescheduled to run later (which may be immediately).

`thr_yield()` is *BLOCKING*.

3.1.6 Thread control

```
int thr_kill(thread_t tid, int sig)
int thr_suspend(thread_t tid)
int thr_continue(thread_t tid)
```

The function `thr_kill()` sends a signal `sig` to thread `tid`. Currently, all signals have the efficacy of `SIGKILL` and will terminate the target thread. Suicide is not permitted—clients must use `thr_exit()` to terminate.

A *kill* does not mean immediate termination of the target thread—specifically, the resources bound to the target may not be freed when this function returns to the caller. All that is guaranteed is that the next time libthread sees the target thread, it will terminate the thread and free resources bound to it. It is however guaranteed that the target will never execute any more client code after this call returns. It is an error to kill non-existent or exited threads.

A return value of `THR_OKAY` indicates that the target thread was terminated as specified. A return value of `THR_ERR` indicates an error.

`thr_kill()` is *optionally-optionally BLOCKING*.

The function `thr_suspend()` stops `tid` from running until a subsequent `thr_continue()`. Any thread can suspend any other—although it is an error (and makes little sense) to suspend oneself. It is okay (but ineffectual) to suspend a thread multiple times. Suspends are not queued and will be collapsed into one. A single continue will cancel all previous suspends.

Libthread guarantees that if this function returns successfully, the target will not execute any further client code until it is explicitly resumed. It is an error to suspend non-existent or exited threads.

A return value of `THR_OKAY` indicates that the target was suspended. A return value of `THR_ERR` indicates an error.

`thr_suspend()` is *optionally-optionally BLOCKING*.

The function `thr_continue()` releases a previously suspended thread `tid` and schedules it to run. This function is also used to start threads that were created in the suspended state (by specifying `THR_SUSPENDED` in the thread creation flags). The target is scheduled to run by this function; it may or may not have started to execute by the time this function returns to the caller.

It is an error to continue non-existent, exited, or non-suspended threads. Trying to resume a thread not already suspended is probably an indication of a race in the algorithm. The client is free to ignore the error return (but it will have to tolerate a warning message from libthread).

A return value of `THR_OKAY` indicates that the target thread was resumed. A return value of `THR_ERR` indicates an error.

`thr_continue()` is *optionally-optionally BLOCKING*.

3.1.7 Thread local data

Threads in libthread do not have any compiler visible local storage other than that available on the runtime stack. Global variables are truly global and are accessible by all threads within a process.

Libthread provides support for *Thread Specific Data* (TSD)—a mechanism by which libthread users can simulate storage that is module-global and thread-local, i.e., TSD separates the notion of the *name* of a data item (which is shared across a group of threads) from the *contents* of the data item that is accessible via this name (which is private to each thread in the group).

The TSD is implemented as a repository of pointers to data objects. Each thread can maintain a small number of such pointers. The allocation of the data objects associated with these pointers is the responsibility of the libthread user. Each pointer is also associated with a *key* (the key is equivalent to the name of the data item). Threads need to use the appropriate key to check pointers to data objects in and out of the TSD. The key name-space is managed by libthread.

The canonical idiom for the use of TSD is as follows. If a group of threads need to maintain a conceptually thread local variable `v`, then one of the threads (most commonly the parent of the users of `v`) creates a `key_v` for the variable `v` and stores the key (which is allocated by libthread) in a global variable.

The creation of a key does not result in storage allocation—it simply means that libthread has been made aware of the fact that sometime in the future one or more threads will use the allocated key to manage their TSD.

When each of the `v`-user threads starts up, it allocates storage of the `sizeof v` and saves its own initial copy of `v` in this area. Each thread then uses the previously created `key_v` to check the allocated pointer into its TSD. From then on, any function called from within these threads can use the same `key_v` to obtain a pointer to the thread-local copy of the data item `v` and hence access `v`.

As long as the pointers checked in by each thread are different, the copies of the variable `v` that each thread accesses will act like thread local storage. Since any function/module within a thread can access this pointer (given the proper key), the variable `v` acts like a module-global object.

The example program in the Tutorial [Section 10] demonstrates the use of TSD.

```
int thr_keycreate(thread_key_t* keyp, void (*destructor)(void *))
int thr_setspecific(thread_key_t key, void* data)
int thr_getspecific(thread_key_t key, void** datap)
```

The function `thr_keycreate()` allocates a new key and saves it in `*keyp`. Any thread may use this key to save and retrieve a pointer to a data object. The access to `*keyp` must be synchronized so that a thread does not try to use a key before it has been allocated.

`destructor` is a pointer to a function that takes a single `void *` argument and returns `void`. It is invoked on thread exit and frees up the storage associated with the data object with key `*keyp`. If the saved pointer is not free-able, then the `destructor` should be specified as null.

A return value of `THR_OKAY` indicates that a new key was successfully allocated. A return value of `THR_ERR` indicates an error.

`thr_keycreate()` is *optionally-optionally BLOCKING*.

The function `thr_setspecific()` installs the pointer `data` with key `key` in the TSD of the calling thread. The same pointer can be retrieved later by a call to `thr_getspecific()`.

It is an error to specify invalid keys. Libthread makes no consistency checks on `data`—it may be anything at all. Specifically, if `data` is a pointer to a stack object or is invalid, very nasty things can happen when the pointer is used later or when destructors are called during thread exit.

A return value of `THR_OKAY` indicates that the pointer was installed with the specified key. a return value of `THR_ERR` indicates an error.

`thr_setspecific()` is *optionally-optionally BLOCKING*.

The function `thr_getspecific()` retrieves the previously saved (via `thr_setspecific()`) pointer associated with key `key` into `*datap`. If no value was previously saved, a 0 is returned. It is an error to specify invalid keys.

A return value of `THR_OKAY` indicates that the pointer associated with the given key was retrieved. A return value of `THR_ERR` indicates an error.

`thr_getspecific()` is *optionally-optionally BLOCKING*.

3.1.8 Thread tracing and error reporting

```
void thr_do_trace(const char* format, ...)
void thr_trace_on(void)
void thr_trace_off(void)
void thr_perror(const char* msg)
```

The actions of libthread can be traced for error reporting and for debugging purposes. Traces can be enabled in two ways—statically or under program control. Regardless of which method is used to control tracing, support for tracing must be compiled into libthread. This is done by defining the preprocessor symbol `ENABLE_TRACE` while compiling the libthread sources (this is done by default). Compiling support for tracing will slow down libthread a little (but not by a noticeable amount).

The static way to enable tracing is to define the environment variable THR_TRACEFLAG when executing the application. A value of 0 for this variable will enable the tracing of all interface functions while a value of 1 will include internal functions as well. Values greater than 1 are equivalent to a value of 1, i.e., interface and internal tracing is enabled. Traces will be written to `stderr` and each trace record is prefixed with a `[pid.tid,name]` tuple. The environment variable `THR_TRACEFILE` specifies a filename as the destination of the traces. If a valid and writable file is found, traces will be *appended* to the contents of the file (or the file created afresh).

With the above static method, tracing is enabled for the complete run of the program and may be useful for detailed trace processing or debugging small runs. For larger runs, the volume of trace data can become large and the application will slow significantly.

The second method to trace programs is under client control. `thr_trace_on()` and `thr_trace_off()` calls can be inserted anywhere in client code to enable and disable tracing. Tracing begins after the `thr_trace_on()` call and continues until the `thr_trace_off()` call. It makes little sense (although it is perfectly acceptable) to make multiple consecutive calls to either `thr_trace_on()` or `thr_trace_off()`.

As mentioned before, tracing support should be compiled into the libthread. If tracing support is not compiled in, neither the environment flags nor the above functions will work. The `thr_trace` functions will print a warning message.

Clients can make use of the tracing facility of libthread. This is nothing but a wrapper around a `printf`-like function which behaves like the trace functions that libthread uses internally. Apart from the fact that `thr_do_trace()` appends its output to the file specified by the environment variable `THR_TRACEFILE` (or to `stderr`), this function is also non- interruptible; thus different threads can use it as a safely interleaved version of `printf`.

This function prints an error message `msg` to `stderr` followed by a string indicating the most recent error that occurred within the calling thread. The error message is based on the value of an internal per-thread `errno` variable. Successful calls within a thread reset this variable (note that this different from the semantics associated with the Unix/C `errno` variable, which is not reset by successful calls).

Neither the internal error number variable nor the message table is exported by libthread—this may be done in future if there is good justification for it (for example, if there is need for an application-wide error reporting module which may wish to handle libthread error messages itself).

All of the trace and print functions in libthread are *NON-BLOCKING*.

3.2 The message interface

Libthread provides reliable, ordered, typed messages between threads. The library also integrates data from files and signal events into message streams (with special types for file and signal messages).

The message passing idiom of libthread is identical to that of write/read or send/recv in Unix. The client program is responsible for allocating message buffers of the proper size, constructing the message (with any application specific headers and field boundaries) in the buffers, and finally deallocating the buffers when they are no longer needed. Libthread does not allocate buffers on behalf of the client, or do any additional buffer management or message formatting. Libthread uses internal buffers to save copies of messages when required. This means that when a message send function successfully completes, the client can safely reuse its buffers.

Libthread messages are tagged—i.e., each message has a tag (or type name) associated with it. Threads can send messages and wait for messages with specific tags. It is the responsibility of the client to make use of the tag name-space and ensure that no conflicts occur between threads in the use of tags. Also, *tag-pairs* may be used the client to build an *RPC* abstraction on top of the message system provided by libthread. Message tags have the same name-space as unsigned integers.

Libthread reserves *three* tag names and associates special semantics with them. These tags names are explained below.

`MSG_TAG_ANY`: This is the message tag that identifies generic untyped messages and functions as a wildcard tag when receiving messages.

> *Sending*: `MSG_TAG_ANY` can be used for sending untyped messages—however, this tag also has special meaning for a receiver and the client is discouraged from using this tag as a send tag except for debugging and testing very simple programs.

Receiving: When `MSG_TAG_ANY` is specified by a receiver, it is requesting libthread to provide it with messages of *any* type—this includes *signal*, *internal*, and *file* messages. Libthread checks for signal, internal and file messages in that order before deciding to block the calling thread (which it does when there are no appropriate messages to be received). Any internal messages that were sent to a receiver with a tag of `MSG_TAG_ANY` will be dequeued if the receiver also specified this tag—one of the reasons why sending messages with this generic tag is not a good thing in programs with complex communication patterns.

`MSG_TAG_SIG`: This is the message tag that identifies signal messages.

Sending: Libthread does not permit the client to use `MSG_TAG_SIG` for sending messages.

Receiving: If a thread specifies `MSG_TAG_SIG` (or the generic `MSG_TAG_ANY`) during a receive, libthread will search for and provide (if available) messages from signals bound to the calling thread. Specifying this tag also means that libthread will not process any file or internal messages that may be waiting for the caller.

`MSG_TAG_FILE`: This is the message tag that identifies all file messages.

Sending: Libthread does not permit the client to use `MSG_TAG_FILE` for sending messages.

Receiving: If a thread specified `MSG_TAG_FILE` (or the generic `MSG_TAG_ANY`) during a receive, libthread will search for and provide (if available) messages from files bound to the calling thread. Specifying this tag also means that libthread will not process any signal or internal messages that may be waiting for the caller.

The client should not assume any numeric ordering among these three special tags—only equality and inequality testing make sense. To allow the client to safely select tags without any conflict with future special tags that may be defined by libthread, the name `MSG_TAG_USER` is also provided. All client-defined tags **must** satisfy the condition `tag >= MSG_TAG_USER` (obviously, `MSG_TAG_USER` is numerically greater than any of libthread's special tags).

In addition to the above three special tags, libthread also allows the client to poll and receive messages from a specific file descriptor. For such uses the value of the message tag should be the same as the file descriptor. This means that the tag name-space is carefully constructed so as to not conflict with the file descriptor name-space. Libthread does not permit the client to use these kinds of tags for sending messages.

3.2.1 Binding files to message queues

```
int msg_bind(int fd, unsigned special)

int msg_bind_buffered(int fd, unsigned special,
                      unsigned (*will_block)(void *), void* desc)

int msg_unbind(int fd)
```

The function `msg_bind()` binds file descriptor `fd` to the calling thread. Binding means that any data arriving on `fd` can be received as messages sent to this thread. File messages are always the last type of message to be checked for during a message receive (with a priority below signal and internal messages). The boolean flag `special` indicates if the file descriptor `fd` is *special* or not. Messages on special files are not dequeued by libthread—the client needs to do this itself. Formatted data streams (such as XDR or X-windows connections) should be treated as special files since special libraries are used to process data that arrive on these streams.

It is not possible to bind the same file descriptor to multiple threads. Mappings cannot be changed directly—the way to map an already mapped `fd` is to do an *unbind* in the *owning* thread followed by a *bind* by the new *owner*. When a thread dies, all files bound to it are automatically released by libthread—this is similar to the freeing of file descriptor resources on process exit.

A return value of `THR_OKAY` indicates that `fd` was successfully bound to the calling thread. A return value of `THR_ERR` indicates an error.

Libthread uses the Unix system call `select()` to determine if there is any data available on a bound file descriptor. If the client buffers data on these file descriptors, then the client may be ready to run (if messages are available in the buffers) but libthread will not schedule the client since the `select()` function may not indicate a file descriptor ready for reading. This can lead to a runnable thread being delayed for arbitrary lengths of time.

To allow buffered streams to be handled by libthread, the function `msg_bind_buffered()` is provided. The `fd` and `special` arguments have the same semantics as those of `msg_bind()`. In addition, this function takes two arguments, `will_block` and `desc`. `will_block` is a function that, when called with descriptor `desc` as its argument, will return a boolean that indicates if the client buffer associated with `desc` is empty. For such files, libthread uses the `will_block` function first to check for buffered messages, before polling the actual file descriptors. This function is useful for handling `stdio` and `XDR` streams.

A return value of `THR_OKAY` indicates that `fd` was successfully bound to the calling thread. A return value of `THR_ERR` indicates an error.

`msg_bind()` and `msg_bind_buffered()` are *optionally BLOCKING*.

The function `msg_unbind()` removes the currently existing binding of `fd` in the calling thread. Bindings made by another thread cannot be deleted. It is also an error (albeit an innocuous one) to unbind a non-bound file descriptor.

During an unbind (either explicitly or implicitly when a thread dies), any file messages that may be waiting on `fd` are silently ignored. Since libthread never dequeues file messages unless the client specifically posts a request for them, there is no danger of *losing* message bytes (compare with the semantics of signal messages).

A return value of `THR_OKAY` indicates that `fd` was unbound from the calling thread. a return value of `THR_ERR` indicates an error.

`msg_unbind()` is *optionally BLOCKING*.

3.2.2 Message passing

```
int msg_send(thread_t tid, tag_t tag, void* buf, unsigned size)
int msg_recv(tag_t* tagp, void* buf, unsigned* countp)
int msg_poll(tag_t* tagp, unsigned block)
```

The function `msg_send()` sends a message of type `tag` and size `size` bytes, pointed to by `buf` to thread `tid`. `tid` can also be any file descriptor that can be written to.

If `tid` refers to a file descriptor, a `write()` system call is invoked and the return value of the write becomes the return value of the send function.

If `tid` is a thread id, an internal message send is invoked. If `tid` has posted a receive for a message of the same tag or a wildcard tag `MSG_TAG_ANY`, the message is directly copied into the receiver's buffers—else the message is copied into an internal buffer and queued at the receiver.

`tag` can be any valid tag except `MSG_TAG_FILE` or `MSG_TAG_SIG` (which simply means that a thread cannot masquerade as a file or a signal).

If the size of the message is larger than what the receiver is prepared to deal with, a warning message is printed and an incomplete buffer is copied, but no error is raised (partly because this cannot be done when the incoming message is queued at the receiver). The buffer is re-usable by the sender as soon as the message send completes.

A return value of `THR_OKAY` indicates that the message was successfully sent. A return value of `THR_ERR` indicates an error.

`msg_send()` is *optionally BLOCKING*.

`msg_recv()` is a complicated function to describe since it does so many things and since the meanings of its arguments and return values are overloaded. However, this internal complexity should make the *interface* easy to use. The semantics of this function will be explained based on the different types of messages that can be received.

Signal messages: Libthread will process signal messages if `*tagp` is `MSG_TAG_ANY` or `MSG_TAG_SIG`—in the former case it will look for signal messages first and in the latter case, it will look *only* for signal messages. Libthread chooses an arbitrary signal *signo* with unprocessed messages and which is bound to the calling thread and designate this signal as the *sender* of the message. `*tagp` is set to `MSG_TAG_SIG`. `*countp` is set to the *number* of unprocessed *signo* signals received so far. `*buf` is not modified in any way. The function returns *signo*.

As mentioned in the description of signal handling, libthread does not deliver signal messages in the same order that the signals were received. It also collapses multiple signals into a single message (although it keeps tracks of multiple signals when this is supported by the underlying operating system).

Internal messages: Libthread will process internal messages if `*tagp` is `MSG_TAG_ANY` or not less than `MSG_TAG_USER`—in the former case it will look for internal messages after signal messages and before file messages and in the latter case, it will look *only* for internal messages. The first message (from some arbitrary sender *sender*) waiting in the receiver's message queue is processed. `*tagp` is set to the tag with which *sender* sent the message. Libthread assumes `*countp` to indicate the number of bytes that the receiver is willing to receive into `*buf` (`buf` must point to a buffer at least this large). The message from *sender* is copied into `*buf` and `*countp` is set to the minimum of the actual size of the message and the original value in `*countp`. The function returns *sender*.

As mentioned in the description of `msg_send()`, libthread will copy only as many bytes as specified by `*countp`. Any excess data in the actual message will be discarded (although libthread will warn of this).

File messages: Libthread will process file messages if `*tagp` is `MSG_TAG_ANY` or `MSG_TAG_FILE` or is a valid file descriptor bound to the calling thread—in the first case libthread will look for file messages (from all bound file descriptors) after signal and internal messages, in the second case, libthread will look *only* for file messages, and in the third case libthread will look for file messages *only* on the specified file. If no special file tag is specified, libthread chooses an arbitrary file `fd` with unprocessed messages and which is bound to the calling thread as the *sender* of the message. When a special file tag is specified, the *sender* becomes the specified `fd` (assuming there is a message waiting). `*tagp` is always set to `MSG_TAG_FILE`. As with internal messages, `*countp` is assumed to specify the size of `*buf`. A maximum of `*countp` bytes are read in from `fd` into `*buf` and `*countp` is set to the actual number of bytes received. The previous two sentences are true if `fd` was not bound as a special file. If `fd` is special, `*tagp` is still set to `MSG_TAG_FILE`, but no bytes are copied into `*buf` nor is `*countp` modified. In all cases, the function returns `fd`.

The above three execution pathways describe what happens when `msg_recv()` finds a waiting message of the appropriate type. However, libthread may need to block the calling thread if messages are not ready.

Wherever appropriate, the receiver is responsible for allocating buffers of the proper size and managing them.

The return value of this function indicates the *sender* of the message (this is the only libthread function that uses a rich return value). As always a return value of `THR_ERR` indicates an error.

`msg_recv()` is *optionally BLOCKING* if messages are waiting, *BLOCKING* otherwise.

Having explained the semantics of `msg_recv()`, it is easy to explain the function `msg_poll()`. Instead of copying messages into buffers, this function simply *check* for messages that may be waiting for the calling thread. If a message is found, the value of `*tagp` is set to the tag of the actual message waiting for the thread.

If no messages are immediately available, then the behavior of this function depends on the value of the boolean flag `block`. If this flag is set to 0, then the calling thread does not block and returns immediately with a value of `THR_ERR` and sets the internal `errno` to `THR_ENOMSG`. This is one case where a return of `THR_ERR` is not really an error. If the flag `block` is set to 1, then the calling thread blocks until an appropriate message is available and returns an indication of the *sender* of the message in the same way that `msg_recv()` does.

`msg_poll()` is *optionally BLOCKING* if messages are waiting (or `block` is unset), *BLOCKING* otherwise.

3.2.3 Message queue enquiry and debugging

```
void msg_dump_state(void)
```

This function prints to `stderr` the current state of libthread's messaging system. As they say, the output is self-explanatory. The client-visible state consists of the current global file descriptor set (the set of file descriptors whose owner threads are blocked waiting for messages), and the state of the *<local file descriptors, polling state, and the message queues>* of each active thread in the system.

This function can be invoked either by the client or from within a debugger. In the latter case, you may catch libthread in an inconsistent state depending on when you invoke the function.

This function is *NON-BLOCKING*.

3.3 The signal interface

Libthread also provides some support for signals. In general, distributing signal processing responsibilities between libthread and the client code is not advised—since there can be some very subtle interactions that are difficult to detect and debug. The signal handling provided by libthread is clean (and for that reason simple, since a lot of the complexities in signal handling are abstracted away from the user). Be warned.

The signal abstraction that libthread provides is similar to that of files as a type of message queue. A simple view of signals would be as follows. Threads bind signals to themselves, much like they bind files (but there are no special signals). As long as a signal remains bound to some thread in a process, libthread handles the occurrence of these signals itself. It keeps a count of the number of times each of the different bound signals occurs.

When a thread wishes to receive a message, it should specify a message tag of `MSG_TAG_ANY` or `MSG_TAG_SIG`. If there are any pending signals, these will be converted into messages for this thread. When a message tag of `MSG_TAG_ANY` is specified by the receiver, signal messages get top priority, over internal messages and finally file messages.

When a signal is unbound, libthread attempts to restore the original disposition of that signal.

The client should keep in mind three aspects of this model:

- Signal messages are *not* delivered in the order in which they were received—if the client's algorithm depends on the specific ordering of two or more signals, then these signals should be handled outside libthread.
- Multiple signals of the same type are collapsed into a single message. When the message is actually generated, the client will get a count of the number of signals of a given type that were received. Therefore if a process receives 10 signals of the type `SIGFOO`, then the thread which has bound `SIGFOO` to it may see 1 message with 10 `SIGFOO`s, 10 messages with 1 `SIGFOO` each or any combination thereof, all depending on the exact interleaving of the order of signal delivery and execution of the message receive calls.
- Libthread uses the underlying signal handling semantics of the operating system. If the kernel does not queue signals, then libthread cannot either. Libthread only simplifies the signal abstraction that the kernel provides, it does not enhance it in any way.

3.3.1 Binding signals to message queues

```
int sig_bind(int signo)
int sig_unbind(int signo)
```

The function `sig_bind()` binds signal `signo` to the calling thread. Any `signo` signals that the process receives will be tracked by libthread and will turn up as messages to the calling thread. It is an error to try to bind a non-existent signal or a signal currently bound to another thread (or to oneself).

A return value of THR_OKAY indicates that signo was bound successfully. A return value of THR_ERR indicates an error.

sig_bind() is *optionally BLOCKING*.

The function sig_unbind() removes the existing between the calling thread and signal signo. If there are any queued signals signo, they will be **DISCARDED** silently. The disposition of signo prior to the binding will be restored. Any future signals of type signo will go unnoticed and unprocessed by libthread, until the next binding occurs. It is an error to unbind an invalid signal, or one that is currently unbound or one that is bound to another thread.

A return value of THR_OKAY indicates that signo was unbound successfully. A return value of THR_ERR indicates an error.

sig_unbind() is *optionally BLOCKING*.

4 COMMENTS AND QUESTIONS

The source distribution of libthread also contains tutorial sources, test suites, *Unix* style man pages, and a *README* file. All questions and comments regarding libthread should be directed to paradyn@cs.wisc.edu.

Tutorial

5 PRELIMINARIES

The following tutorial explains writing simple C and C++ programs that use *libpdthread*. The examples are meant only to illustrate the use of libpdthread functions; error checking, and structured programming are given secondary importance. The complete interface to libpdthread functions is documented in the User Guide [Section 3].

6 BUILDING AND INSTALLING LIBPDTHREAD

Currently, the libpdthread sources are distributed only as part of the Paradyn source distribution. Assuming that the environment variable PD holds the location where the Paradyn source distribution was installed, the libpdthread library sources are in the $PD/core/thread directory.

The $PD/core/thread directory contains a subdirectory named ‘src’ containing the libpdthread source, a subdirectory named ‘h’ containing the libpdthread external header, and a set of subdirectories named for the supported platforms. (The library will be built in the appropriate platform subdirectory.) Throughout the rest of this document, we assume that the PLATFORM environment variable contains the appropriate specification for the target platform.

To build the library, change to the appropriate platform subdirectory under $PD/core/thread. Edit the Makefile in that directory to ensure that any platform-specific variables are set appropriately for your platform.

❑ **v2.1**: During compilation (of `arch-os.C`) `ARCH_STACK_DIRECTION` must be set to either `DIRECTION_DOWNWARD` or `DIRECTION_UPWARD` as appropriate for the platform, as shown by the following line added to platform Makefiles:

```
CFLAGS  += -DARCH_STACK_DIRECTION=DIRECTION_DOWNWARD
```

This replaces a previous on-the-fly test for stack direction which was found to be unreliable.

When you are ready to build the library, type ‘make’ (on UNIX systems) or ‘nmake’ (on Windows systems) to build the library. Note that although the library’s source files are named with .c extensions, the library is implemented in C++ and must be compiled with a C++ compiler.

To install the library, type ‘make install’ (on UNIX systems) or ‘nmake install’ (on Windows systems) from the $PD/core/thread/$PLATFORM directory. The library will be installed into the $PD/lib/$PLATFORM directory.

7 COMPILING A LIBPDTHREAD PROGRAM

All libpdthread programs must include the header file `"thread/h/thread.h"`. The location of this file needs to be specified via the `-I` command line option during compilation of the application program. libpdthread programs must also be linked with the library `libpdthread.a`. The location of this library needs to be specified either via the `-L` command line option or as a pathname during linking.

All libpdthread programs must include the header “thread/h/thread.h”. The location of this file must be specified to the compiler via the -I command line option. For example, to compile a C source file named ‘main.c’ that uses the libpdthread library, one could use a command line such as

```
gcc -c -I$PD/core main.c
```

where the PD environment variable contains the location of the Paradyn source distribution.

Although the library may be used with both C and C++ programs, the library is implemented in C++ and uses some

Currently, libpdthread is implemented in C++. Therefore, a C++ compiler must be used to control

linked by C++, with -lpdthread and -lsocket.

8 A “HELLO WORLD” PROGRAM

The ***“hello.c”*** program creates two threads, named `foo` and `bar`. Each thread uses the libpdthread print function `thr_do_trace()` to print a *“hello world”* message. The `main` program waits for each thread to terminate. The libpdthread functions `thr_create()` and `thr_join()` are invoked with default arguments.

Each thread loops for a different number of iterations, and voluntarily invoke a context switch each iteration. If libpdthread tracing is enabled during the execution of the program, the context switch between the two threads can be seen in the trace output.

```
#include <stdio.h>
#include <stdlib.h>
#include "thread/h/thread.h"

static void* foo(void * junk) {
      unsigned i;
      thr_name("foo"); thr_do_trace("HELLO WORLD");
      for (unsigned i = 0; i < 10; ++i) { thr_yield(); } // loop
      return 0;
}

static void* bar(void *) {
      unsigned i;
      thr_name("bar"); thr_do_trace("hello world");
      for (i = 0; i < 20; ++i) { thr_yield(); } // loop
      return 0;
}

int main() {
      thread_t tfoo, tbar;

      thr_create(0, 0, foo, 0, 0, &tfoo); // create foo
      thr_create(0, 0, bar, 0, 0, &tbar); // create bar

      thr_name("main"); thr_do_trace("tfoo=%u, tbar=%u", tfoo, tbar);

      thr_join(0, 0, 0); // join with foo or bar
      thr_join(0, 0, 0); // join with the other

      return 0;
}
```

Figure 1: Program "hello.c"

9 SENDING AND RECEIVING MESSAGES

The ***"msg.c"*** program illustrates the use of simple messages. The `main` program creates threads `reader` and `writer`. The `writer` thread sends a series of messages to the `reader` thread. The `reader` prints out the values that it receives. The `reader` and `writer` threads use the message tag `RW_TAG` to communicate.

Since the creation of the `reader` and `writer` threads can be arbitrarily separated in time, both threads synchronize with the `main` program using message tag `RW_READY`. Once each thread receives the ready message from the `main` program, it starts exchanging messages with the other thread.

```
#include <stdio.h>
#include <stdlib.h>
#include "thread/h/thread.h"

static thread_t tmain, treader, twriter;

#define RW_READY (MSG_TAG_USER+1)
#define RW_TAG   (MSG_TAG_USER+2)

static void* reader(void * junk) {
      unsigned i; tag_t tag;
      thread_t tid = THR_TID_UNSPEC;
      msg_send(tmain, RW_READY, 0, 0); // send ready to main
      tag = RW_READY; msg_recv(&tid, &tag, 0, 0); // sync with main
      for (i = 0; i < 10; ++i) {
            unsigned msg, size;
            tag = RW_TAG; size = sizeof(msg);
            msg_recv(&tid, &tag, &msg, &size);
            thr_do_trace("msg=%u", msg); // receive and print message
      }
      return 0;
}

static void* writer(void * junk) {
      unsigned i; tag_t tag;
      thread_t tid = THR_TID_UNSPEC;
      msg_send(tmain, RW_READY, 0, 0); // send ready to main
      tag = RW_READY; msg_recv(&tid, &tag, 0, 0); // synch with main
      for (i = 0; i < 10; ++i) {
            msg_send(treader, RW_TAG, &i, sizeof i); // send message
      }
      return 0;
}

int main() {
      tag_t tag;
      tmain = thr_self();
      thr_create(0, 0, reader, 0, 0, &treader); // create reader
      thr_create(0, 0, writer, 0, 0, &twriter); // create writer

      tag = RW_READY; msg_recv(&treader, &tag, 0, 0); // wait for reader
      tag = RW_READY; msg_recv(&twriter, &tag, 0, 0); // and writer
      msg_send(treader, RW_READY, 0, 0); // release reader
      msg_send(twriter, RW_READY, 0, 0); // and writer

      thr_join(0, 0, 0);
      thr_join(0, 0, 0);

      return 0;
}
```

Figure 2: Program "msg.c"

10 USING THREAD LOCAL STORAGE

libpdthread implements thread local storage via the *Thread Specific Data* mechanism. Threads register pointers to heap data objects in a repository, associating a key with each pointer. When the thread requests a key from the repository, it gets the pointer that was checked in. In this way, different threads can use the same key to indicate a shared name, but still get access to different data objects. The ***"tsd.c"*** program demonstrates the use of thread specific data.

The `main` program creates a fresh respository key and saves it in the global variable `tsd_key`. Each `foo` thread then allocates a private heap object, saves some data in it, and registers the pointer to this data in the respository with the key `tsd_key`. Subsequent calls to function `bar` by each of the threads retrieves the distinct pointers to the heap data.

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

struct Tsd {
      int    i;
      double d;
      char   c;
      void*  p;
};

static thread_key_t tsd_key;

static void myfree(void* ptr) {
      thr_do_trace("destructor called on pointer %p", ptr);
      delete (Tsd *) ptr;
}

static void bar() {
      thread_t me = thr_self();
      Tsd*     ptr;

      thr_getspecific(tsd_key, (void **) &ptr);

      if (ptr->i != (int)    me) { thr_do_trace("i is %d", ptr->i); }
      if (ptr->d != (double) me) { thr_do_trace("d is %g", ptr->d); }
      if (ptr->c != (char)   me) { thr_do_trace("c is %d", ptr->c); }
      if (ptr->p != (void *) me) { thr_do_trace("p is %p", ptr->p); }
}

static void* foo(void *) {
      thread_t me  = thr_self();
      Tsd*     ptr = new Tsd;

      ptr->i = (int)    me;
      ptr->d = (double) me;
      ptr->c = (char)   me;
      ptr->p = (void *) me;

      thr_setspecific(tsd_key, (void *) ptr);
```

```
        thr_do_trace("installed data, yielding to others");

        for (unsigned i = 0; i < 10; ++i) { thr_yield(); }

        bar();
        thr_do_trace("finishing, verify destructor on %p", ptr);
        return 0;
}

int main() {
        thr_keycreate(&tsd_key, myfree);
        thr_do_trace("tsd key = %u", tsd_key);

        for (unsigned i = 0; i < 10; ++i) {
                thr_create(0, 0, foo, 0, 0, 0);
        }
        for (unsigned i = 0; i < 10; ++i) {
                thr_join(0, 0, 0);
        }

        return 0;
}
```

Figure 3: Program “tsd.c”

11 BINDING FILES TO MESSAGE QUEUES

libpdthread allows readable files to be bound to message queues. Data on these files will be available as messages to the thread that binds the file descriptor to its message queues. The program *"filemsg.c"* demonstrates the use of file descriptor binding in libpdthread.

The thread `foo` opens the file *"/vmunix"* (or some similarly large file on the system), and binds the associated file descriptor to its message queue using the call `msg_bind()`. The thread then reads 10 messages of size 32 bytes from this file. Since the data comes from a file, it is unformatted and unbuffered. If a thread wishes to bind streams having formatted and/or buffered messages, the descriptor must be bound as special and/or buffered.

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

static void* foo(void *) {
      int fd = open("/vmunix", O_RDONLY);
      msg_bind(fd, 0); // bind descriptor as non-special file

      for (unsigned i = 0; i < 10; ++i) {
            char     msg[32];
            tag_t    tag;
            unsigned size;
            int      who;

            tag = MSG_TAG_FILE; size = sizeof msg;
            who = msg_recv(&tag, msg, &size);

            thr_do_trace("got %u bytes from %d", size, who);
      }

      msg_unbind(fd);
      return 0;
}

int main() {
      thr_create(0, 0, foo, 0, 0, 0);
      thr_join(0, 0, 0);
      return 0;
}
```

Figure 4: Program "filemsg.c"

■