# NAME

**pth** – GNU Portable Threads

# VERSION

GNU Pth PTH_VERSION_STR

# SYNOPSIS

**Global Library Management**
: pth_init, pth_kill, pth_ctrl, pth_version.

**Thread Attribute Handling**
: pth_attr_of, pth_attr_new, pth_attr_init, pth_attr_set, pth_attr_get, pth_attr_destroy.

**Thread Control**
: pth_spawn, pth_once, pth_self, pth_suspend, pth_resume, pth_yield, pth_nap, pth_wait, pth_cancel, pth_abort, pth_raise, pth_join, pth_exit.

**Utilities**
: pth_fdmode, pth_time, pth_timeout, pth_sfiodisc.

**Cancellation Management**
: pth_cancel_point, pth_cancel_state.

**Event Handling**
: pth_event, pth_event_typeof, pth_event_extract, pth_event_concat, pth_event_isolate, pth_event_walk, pth_event_status, pth_event_free.

**Key-Based Storage**
: pth_key_create, pth_key_delete, pth_key_setdata, pth_key_getdata.

**Message Port Communication**
: pth_msgport_create, pth_msgport_destroy, pth_msgport_find, pth_msgport_pending, pth_msgport_put, pth_msgport_get, pth_msgport_reply.

**Thread Cleanups**
: pth_cleanup_push, pth_cleanup_pop.

**Process Forking**
: pth_atfork_push, pth_atfork_pop, pth_fork.

**Synchronization**
: pth_mutex_init, pth_mutex_acquire, pth_mutex_release, pth_rwlock_init, pth_rwlock_acquire, pth_rwlock_release, pth_cond_init, pth_cond_await, pth_cond_notify, pth_barrier_init, pth_barrier_reach.

**User-Space Context**
: pth_uctx_create, pth_uctx_make, pth_uctx_switch, pth_uctx_destroy.

**Generalized POSIX Replacement API**
: pth_sigwait_ev, pth_accept_ev, pth_connect_ev, pth_select_ev, pth_poll_ev, pth_read_ev, pth_readv_ev, pth_write_ev, pth_writev_ev, pth_recv_ev, pth_recvfrom_ev, pth_send_ev, pth_sendto_ev.

**Standard POSIX Replacement API**
: pth_nanosleep, pth_usleep, pth_sleep, pth_waitpid, pth_system, pth_sigmask, pth_sigwait, pth_accept, pth_connect, pth_select, pth_pselect, pth_poll, pth_read, pth_readv, pth_write, pth_writev, pth_pread, pth_pwrite, pth_recv, pth_recvfrom, pth_send, pth_sendto.

# DESCRIPTION

```
  ____  _   _
 |  _ \| |_| |__
 | |_) | __| '_ \         ``Only those who attempt
 |  __/| |_| | | |          the absurd can achieve
 |_|    \__|_| |_|          the impossible.''
```

**Pth** is a very portable POSIX/ANSI–C based library for Unix platforms which provides non-preemptive priority-based scheduling for multiple threads of execution (aka 'multithreading') inside event-driven applications. All threads run in the same address space of the application process, but each thread has its own individual program counter, run-time stack, signal mask and `errno` variable.

The thread scheduling itself is done in a cooperative way, i.e., the threads are managed and dispatched by a priority− and event-driven non-preemptive scheduler. The intention is that this way both better portability and run-time performance is achieved than with preemptive scheduling. The event facility allows threads to wait until various types of internal and external events occur, including pending I/O on file descriptors, asynchronous signals, elapsed timers, pending I/O on message ports, thread and process termination, and even results of customized callback functions.

**Pth** also provides an optional emulation API for POSIX.1c threads ('Pthreads') which can be used for backward compatibility to existing multithreaded applications. See **Pth**'s *pthread* (3) manual page for details.

### Threading Background

When programming event-driven applications, usually servers, lots of regular jobs and one-shot requests have to be processed in parallel. To efficiently simulate this parallel processing on uniprocessor machines, we use 'multitasking' — that is, we have the application ask the operating system to spawn multiple instances of itself. On Unix, typically the kernel implements multitasking in a preemptive and priority-based way through heavy-weight processes spawned with *fork* (2). These processes usually do *not* share a common address space. Instead they are clearly separated from each other, and are created by direct cloning a process address space (although modern kernels use memory segment mapping and copy-on-write semantics to avoid unnecessary copying of physical memory).

The drawbacks are obvious: Sharing data between the processes is complicated, and can usually only be done efficiently through shared memory (but which itself is not very portable). Synchronization is complicated because of the preemptive nature of the Unix scheduler (one has to use *atomic* locks, etc). The machine's resources can be exhausted very quickly when the server application has to serve too many long-running requests (heavy−weight processes cost memory). And when each request spawns a sub-process to handle it, the server performance and responsiveness is horrible (heavy−weight processes cost time to spawn). Finally, the server application doesn't scale very well with the load because of these resource problems. In practice, lots of tricks are usually used to overcome these problems − ranging from pre-forked sub-process pools to semi-serialized processing, etc.

One of the most elegant ways to solve these resource− and data-sharing problems is to have multiple *light-weight* threads of execution inside a single (heavy−weight) process, i.e., to use *multithreading*. Those *threads* usually improve responsiveness and performance of the application, often improve and simplify the internal program structure, and most important, require less system resources than heavy-weight processes. Threads are neither the optimal run-time facility for all types of applications, nor can all applications benefit from them. But at least event-driven server applications usually benefit greatly from using threads.

### The World of Threading

Even though lots of documents exists which describe and define the world of threading, to understand **Pth**, you need only basic knowledge about threading. The following definitions of thread-related terms should at least help you understand thread programming enough to allow you to use **Pth**.

**o process** vs. **thread**

A process on Unix systems consists of at least the following fundamental ingredients: *virtual memory table*, *program code*, *program counter*, *heap memory*, *stack memory*, *stack pointer*, *file descriptor set*, *signal table*. On every process switch, the kernel saves and restores these ingredients for the individual processes. On the other hand, a thread consists of only a private program counter, stack memory, stack pointer and signal table. All other ingredients, in particular the virtual memory, it shares with the other threads of the same process.

o **kernel-space** vs. **user-space** threading

Threads on a Unix platform traditionally can be implemented either inside kernel-space or user−space. When threads are implemented by the kernel, the thread context switches are performed by the kernel without the application's knowledge. Similarly, when threads are implemented in user−space, the thread context switches are performed by an application library, without the kernel's knowledge. There also are hybrid threading approaches where, typically, a user-space library binds one or more user-space threads to one or more kernel-space threads (there usually called light-weight processes − or in short LWPs).

User-space threads are usually more portable and can perform faster and cheaper context switches (for instance via *swapcontext*(2) or *setjmp*(3)/*longjmp*(3)) than kernel based threads. On the other hand, kernel-space threads can take advantage of multiprocessor machines and don't have any inherent I/O blocking problems. Kernel-space threads are usually scheduled in preemptive way side-by-side with the underlying processes. User-space threads on the other hand use either preemptive or non-preemptive scheduling.

o **preemptive** vs. **non-preemptive** thread scheduling

In preemptive scheduling, the scheduler lets a thread execute until a blocking situation occurs (usually a function call which would block) or the assigned timeslice elapses. Then it detracts control from the thread without a chance for the thread to object. This is usually realized by interrupting the thread through a hardware interrupt signal (for kernel-space threads) or a software interrupt signal (for user-space threads), like `SIGALRM` or `SIGVTALRM`. In non-preemptive scheduling, once a thread received control from the scheduler it keeps it until either a blocking situation occurs (again a function call which would block and instead switches back to the scheduler) or the thread explicitly yields control back to the scheduler in a cooperative way.

o **concurrency** vs. **parallelism**

Concurrency exists when at least two threads are *in progress* at the same time. Parallelism arises when at least two threads are *executing* simultaneously. Real parallelism can be only achieved on multiprocessor machines, of course. But one also usually speaks of parallelism or *high concurrency* in the context of preemptive thread scheduling and of *low concurrency* in the context of non-preemptive thread scheduling.

o **responsiveness**

The responsiveness of a system can be described by the user visible delay until the system responses to an external request. When this delay is small enough and the user doesn't recognize a noticeable delay, the responsiveness of the system is considered good. When the user recognizes or is even annoyed by the delay, the responsiveness of the system is considered bad.

o **reentrant**, **thread-safe** and **asynchronous-safe** functions

A reentrant function is one that behaves correctly if it is called simultaneously by several threads and then also executes simultaneously. Functions that access global state, such as memory or files, of course, need to be carefully designed in order to be reentrant. Two traditional approaches to solve these problems are caller-supplied states and thread-specific data.

Thread-safety is the avoidance of *data races*, i.e., situations in which data is set to either correct or incorrect value depending upon the (unpredictable) order in which multiple threads access and modify the data. So a function is thread-safe when it still behaves semantically correct when called simultaneously by several threads (it is not required that the functions also execute simultaneously). The traditional approach to achieve thread-safety is to wrap a function body with an internal mutual exclusion lock (aka 'mutex'). As you should recognize, reentrant is a stronger attribute than thread−safe, because it is harder to achieve and results especially in no run-time contention between threads. So, a reentrant function is always thread−safe, but not vice versa.

Additionally there is a related attribute for functions named asynchronous−safe, which comes into play in conjunction with signal handlers. This is very related to the problem of reentrant functions. An asynchronous-safe function is one that can be called safe and without side-effects from within a signal handler context. Usually very few functions are of this type, because an application is very restricted in what it can perform from within a signal handler (especially what system functions it is allowed to call). The reason mainly is, because only a few system functions are officially declared by POSIX as guaranteed to be asynchronous–safe. Asynchronous-safe functions usually have to be already reentrant.

**User-Space Threads**

User-space threads can be implemented in various way. The two traditional approaches are:

1. **Matrix-based explicit dispatching between small units of execution:**

   Here the global procedures of the application are split into small execution units (each is required to not run for more than a few milliseconds) and those units are implemented by separate functions. Then a global matrix is defined which describes the execution (and perhaps even dependency) order of these functions. The main server procedure then just dispatches between these units by calling one function after each other controlled by this matrix. The threads are created by more than one jump-trail through this matrix and by switching between these jump-trails controlled by corresponding occurred events.

   This approach gives the best possible performance, because one can fine-tune the threads of execution by adjusting the matrix, and the scheduling is done explicitly by the application itself. It is also very portable, because the matrix is just an ordinary data structure, and functions are a standard feature of ANSI C.

   The disadvantage of this approach is that it is complicated to write large applications with this approach, because in those applications one quickly gets hundreds(!) of execution units and the control flow inside such an application is very hard to understand (because it is interrupted by function borders and one always has to remember the global dispatching matrix to follow it). Additionally, all threads operate on the same execution stack. Although this saves memory, it is often nasty, because one cannot switch between threads in the middle of a function. Thus the scheduling borders are the function borders.

2. **Context-based implicit scheduling between threads of execution:**

   Here the idea is that one programs the application as with forked processes, i.e., one spawns a thread of execution and this runs from the begin to the end without an interrupted control flow. But the control flow can be still interrupted – even in the middle of a function. Actually in a preemptive way, similar to what the kernel does for the heavy-weight processes, i.e., every few milliseconds the user-space scheduler switches between the threads of execution. But the thread itself doesn't recognize this and usually (except for synchronization issues) doesn't have to care about this.

   The advantage of this approach is that it's very easy to program, because the control flow and context of a thread directly follows a procedure without forced interrupts through function borders. Additionally, the programming is very similar to a traditional and well understood *fork* (2) based approach.

   The disadvantage is that although the general performance is increased, compared to using approaches based on heavy-weight processes, it is decreased compared to the matrix-approach above. Because the implicit preemptive scheduling does usually a lot more context switches (every user-space context switch costs some overhead even when it is a lot cheaper than a kernel-level context switch) than the explicit cooperative/non–preemptive scheduling. Finally, there is no really portable POSIX/ANSI–C based way to implement user-space preemptive threading. Either the platform already has threads, or one has to hope that some semi-portable package exists for it. And even those semi-portable packages usually have to deal with assembler code and other nasty internals and are not easy to port to forthcoming platforms.

So, in short: the matrix-dispatching approach is portable and fast, but nasty to program. The thread scheduling approach is easy to program, but suffers from synchronization and portability problems caused by its preemptive nature.

### The Compromise of Pth

But why not combine the good aspects of both approaches while avoiding their bad aspects? That's the goal of **Pth**. **Pth** implements easy-to-program threads of execution, but avoids the problems of preemptive scheduling by using non-preemptive scheduling instead.

This sounds like, and is, a useful approach. Nevertheless, one has to keep the implications of non-preemptive thread scheduling in mind when working with **Pth**. The following list summarizes a few essential points:

- **Pth provides maximum portability, but NOT the fanciest features**.

  This is, because it uses a nifty and portable POSIX/ANSI–C approach for thread creation (and this way doesn't require any platform dependent assembler hacks) and schedules the threads in non-preemptive way (which doesn't require unportable facilities like `SIGVTALRM`). On the other hand, this way not all fancy threading features can be implemented. Nevertheless the available facilities are enough to provide a robust and full-featured threading system.

- **Pth increases the responsiveness and concurrency of an event-driven application, but NOT the concurrency of number-crunching applications**.

  The reason is the non-preemptive scheduling. Number-crunching applications usually require preemptive scheduling to achieve concurrency because of their long CPU bursts. For them, non-preemptive scheduling (even together with explicit yielding) provides only the old concept of 'coroutines'. On the other hand, event driven applications benefit greatly from non-preemptive scheduling. They have only short CPU bursts and lots of events to wait on, and this way run faster under non-preemptive scheduling because no unnecessary context switching occurs, as it is the case for preemptive scheduling. That's why **Pth** is mainly intended for server type applications, although there is no technical restriction.

- **Pth requires thread-safe functions, but NOT reentrant functions**.

  This nice fact exists again because of the nature of non-preemptive scheduling, where a function isn't interrupted and this way cannot be reentered before it returned. This is a great portability benefit, because thread-safety can be achieved more easily than reentrance possibility. Especially this means that under **Pth** more existing third-party libraries can be used without side-effects than it's the case for other threading systems.

- **Pth doesn't require any kernel support, but can NOT benefit from multiprocessor machines**.

  This means that **Pth** runs on almost all Unix kernels, because the kernel does not need to be aware of the **Pth** threads (because they are implemented entirely in user–space). On the other hand, it cannot benefit from the existence of multiprocessors, because for this, kernel support would be needed. In practice, this is no problem, because multiprocessor systems are rare, and portability is almost more important than highest concurrency.

### The life cycle of a thread

To understand the **Pth** Application Programming Interface (API), it helps to first understand the life cycle of a thread in the **Pth** threading system. It can be illustrated with the following directed graph:

```
             NEW
              |
              V
      +---> READY ---+
      |       ^      |
      |       |      V
   WAITING <--+-- RUNNING
                     |
      :              V
   SUSPENDED       DEAD
```

When a new thread is created, it is moved into the **NEW** queue of the scheduler. On the next dispatching for this thread, the scheduler picks it up from there and moves it to the **READY** queue. This is a queue containing all threads which want to perform a CPU burst. There they are queued in priority order. On each dispatching step, the scheduler always removes the thread with the highest priority only. It then increases the priority of all remaining threads by 1, to prevent them from 'starving'.

The thread which was removed from the **READY** queue is the new **RUNNING** thread (there is always just one **RUNNING** thread, of course). The **RUNNING** thread is assigned execution control. After this thread yields execution (either explicitly by yielding execution or implicitly by calling a function which would block) there are three possibilities: Either it has terminated, then it is moved to the **DEAD** queue, or it has events on which it wants to wait, then it is moved into the **WAITING** queue. Else it is assumed it wants to perform more CPU bursts and immediately enters the **READY** queue again.

Before the next thread is taken out of the **READY** queue, the **WAITING** queue is checked for pending events. If one or more events occurred, the threads that are waiting on them are immediately moved to the **READY** queue.

The purpose of the **NEW** queue has to do with the fact that in **Pth** a thread never directly switches to another thread. A thread always yields execution to the scheduler and the scheduler dispatches to the next thread. So a freshly spawned thread has to be kept somewhere until the scheduler gets a chance to pick it up for scheduling. That is what the **NEW** queue is for.

The purpose of the **DEAD** queue is to support thread joining. When a thread is marked to be unjoinable, it is directly kicked out of the system after it terminated. But when it is joinable, it enters the **DEAD** queue. There it remains until another thread joins it.

Finally, there is a special separated queue named **SUSPENDED**, to where threads can be manually moved from the **NEW**, **READY** or **WAITING** queues by the application. The purpose of this special queue is to temporarily absorb suspended threads until they are again resumed by the application. Suspended threads do not cost scheduling or event handling resources, because they are temporarily completely out of the scheduler's scope. If a thread is resumed, it is moved back to the queue from where it originally came and this way again enters the schedulers scope.

## APPLICATION PROGRAMMING INTERFACE (API)

In the following the **Pth** *Application Programming Interface* (API) is discussed in detail. With the knowledge given above, it should now be easy to understand how to program threads with this API. In good Unix tradition, **Pth** functions use special return values (`NULL` in pointer context, `FALSE` in boolean context and `-1` in integer context) to indicate an error condition and set (or pass through) the `errno` system variable to pass more details about the error to the caller.

### Global Library Management

The following functions act on the library as a whole. They are used to initialize and shutdown the scheduler and fetch information from it.

int **pth_init**(void);

This initializes the **Pth** library. It has to be the first **Pth** API function call in an application, and is mandatory. It's usually done at the begin of the *main()* function of the application. This implicitly spawns the internal scheduler thread and transforms the single execution unit of the current process into a thread (the 'main' thread). It returns `TRUE` on success and `FALSE` on error.

int **pth_kill**(void);

This kills the **Pth** library. It should be the last **Pth** API function call in an application, but is not really required. It's usually done at the end of the main function of the application. At least, it has to be called from within the main thread. It implicitly kills all threads and transforms back the calling thread into the single execution unit of the underlying process. The usual way to terminate a **Pth** application is either a simple '`pth_exit(0);`' in the main thread (which waits for all other threads to terminate, kills the threading system and then terminates the process) or a '`pth_kill(); exit(0)`' (which immediately kills the threading system and terminates the process). The *pth_kill()* return immediately with a return code of `FALSE` if it is not called from within the main thread. Else it kills the threading system and returns `TRUE`.

long **pth_ctrl**(unsigned long *query*, ...);

This is a generalized query/control function for the **Pth** library. The argument *query* is a bitmask formed out of one or more `PTH_CTRL_`*XXXX* queries. Currently the following queries are supported:

`PTH_CTRL_GETTHREADS`

This returns the total number of threads currently in existence. This query actually is formed out of the combination of queries for threads in a particular state, i.e., the `PTH_CTRL_GETTHREADS` query is equal to the OR-combination of all the following specialized queries:

`PTH_CTRL_GETTHREADS_NEW` for the number of threads in the new queue (threads created via *pth_spawn*(3) but still not scheduled once), `PTH_CTRL_GETTHREADS_READY` for the number of threads in the ready queue (threads who want to do CPU bursts), `PTH_CTRL_GETTHREADS_RUNNING` for the number of running threads (always just one thread!), `PTH_CTRL_GETTHREADS_WAITING` for the number of threads in the waiting queue (threads waiting for events), `PTH_CTRL_GETTHREADS_SUSPENDED` for the number of threads in the suspended queue (threads waiting to be resumed) and `PTH_CTRL_GETTHREADS_DEAD` for the number of threads in the new queue (terminated threads waiting for a join).

`PTH_CTRL_GETAVLOAD`

This requires a second argument of type '`float *`' (pointer to a floating point variable). It stores a floating point value describing the exponential averaged load of the scheduler in this variable. The load is a function from the number of threads in the ready queue of the schedulers dispatching unit. So a load around 1.0 means there is only one ready thread (the standard situation when the application has no high load). A higher load value means there a more threads ready who want to do CPU bursts. The average load value updates once per second only. The return value for this query is always 0.

`PTH_CTRL_GETPRIO`

This requires a second argument of type '`pth_t`' which identifies a thread. It returns the priority (ranging from `PTH_PRIO_MIN` to `PTH_PRIO_MAX`) of the given thread.

`PTH_CTRL_GETNAME`

This requires a second argument of type '`pth_t`' which identifies a thread. It returns the name of the given thread, i.e., the return value of *pth_ctrl*(3) should be casted to a '`char *`'.

`PTH_CTRL_DUMPSTATE`

This requires a second argument of type '`FILE *`' to which a summary of the internal **Pth** library state is written to. The main information which is currently written out is the current state of the thread pool.

`PTH_CTRL_FAVOURNEW`

This requires a second argument of type '`int`' which specified whether the **GNU** **Pth** scheduler favours new threads on startup, i.e., whether they are moved from the new queue to the top (argument is `TRUE`) or middle (argument is `FALSE`) of the ready queue. The default is to favour new threads to make sure they do not starve already at startup, although this slightly violates the strict priority based scheduling.

The function returns `-1` on error.

long **pth_version**(void);

This function returns a hex-value '0x*VRRTLL*' which describes the current **Pth** library version. *V* is the version, *RR* the revisions, *LL* the level and *T* the type of the level (alphalevel=0, betalevel=1, patchlevel=2, etc). For instance **Pth** version 1.0b1 is encoded as 0x100101. The reason for this unusual mapping is that this way the version number is steadily *increasing*. The same value is also available under compile time as `PTH_VERSION`.

### Thread Attribute Handling

Attribute objects are used in **Pth** for two things: First stand−alone/unbound attribute objects are used to store attributes for to be spawned threads. Bounded attribute objects are used to modify attributes of already existing threads. The following attribute fields exists in attribute objects:

`PTH_ATTR_PRIO` (read−write) [`int`]
: Thread Priority between `PTH_PRIO_MIN` and `PTH_PRIO_MAX`. The default is `PTH_PRIO_STD`.

`PTH_ATTR_NAME` (read−write) [`char *`]
: Name of thread (up to 40 characters are stored only), mainly for debugging purposes.

`PTH_ATTR_DISPATCHES` (read−write) [`int`]
: In bounded attribute objects, this field is incremented every time the context is switched to the associated thread.

`PTH_ATTR_JOINABLE` (read−write> [`int`]
: The thread detachment type, `TRUE` indicates a joinable thread, `FALSE` indicates a detached thread. When a thread is detached, after termination it is immediately kicked out of the system instead of inserted into the dead queue.

`PTH_ATTR_CANCEL_STATE` (read−write) [`unsigned int`]
: The thread cancellation state, i.e., a combination of `PTH_CANCEL_ENABLE` or `PTH_CANCEL_DISABLE` and `PTH_CANCEL_DEFERRED` or `PTH_CANCEL_ASYNCHRONOUS`.

`PTH_ATTR_STACK_SIZE` (read−write) [`unsigned int`]
: The thread stack size in bytes. Use lower values than 64 KB with great care!

`PTH_ATTR_STACK_ADDR` (read−write) [`char *`]
: A pointer to the lower address of a chunk of *malloc* (3)'ed memory for the stack.

`PTH_ATTR_TIME_SPAWN` (read−only) [`pth_time_t`]
: The time when the thread was spawned. This can be queried only when the attribute object is bound to a thread.

`PTH_ATTR_TIME_LAST` (read−only) [`pth_time_t`]
: The time when the thread was last dispatched. This can be queried only when the attribute object is bound to a thread.

`PTH_ATTR_TIME_RAN` (read−only) [`pth_time_t`]
: The total time the thread was running. This can be queried only when the attribute object is bound to a thread.

`PTH_ATTR_START_FUNC` (read−only) [`void *(*)(void *)`]
: The thread start function. This can be queried only when the attribute object is bound to a thread.

`PTH_ATTR_START_ARG` (read−only) [`void *`]
: The thread start argument. This can be queried only when the attribute object is bound to a thread.

`PTH_ATTR_STATE` (read−only) [`pth_state_t`]
: The scheduling state of the thread, i.e., either `PTH_STATE_NEW`, `PTH_STATE_READY`, `PTH_STATE_WAITING`, or `PTH_STATE_DEAD` This can be queried only when the attribute object is bound to a thread.

`PTH_ATTR_EVENTS` (read−only) [`pth_event_t`]
: The event ring the thread is waiting for. This can be queried only when the attribute object is bound to a thread.

`PTH_ATTR_BOUND` (read−only) [`int`]
: Whether the attribute object is bound (`TRUE`) to a thread or not (`FALSE`).

The following API functions can be used to handle the attribute objects:

pth_attr_t **pth_attr_of**(pth_t *tid*);

This returns a new attribute object *bound* to thread *tid*. Any queries on this object directly fetch attributes from *tid*. And attribute modifications directly change *tid*. Use such attribute objects to modify existing threads.

pth_attr_t **pth_attr_new**(void);

This returns a new *unbound* attribute object. An implicit *pth_attr_init()* is done on it. Any queries on this object just fetch stored attributes from it. And attribute modifications just change the stored attributes. Use such attribute objects to pre-configure attributes for to be spawned threads.

int **pth_attr_init**(pth_attr_t *attr*);

This initializes an attribute object *attr* to the default values: `PTH_ATTR_PRIO` := `PTH_PRIO_STD`, `PTH_ATTR_NAME` := '`unknown`', `PTH_ATTR_DISPATCHES` := `0`, `PTH_ATTR_JOINABLE` := `TRUE`, `PTH_ATTR_CANCELSTATE` := `PTH_CANCEL_DEFAULT`, `PTH_ATTR_STACK_SIZE` := 64*1024 and `PTH_ATTR_STACK_ADDR` := `NULL`. All other `PTH_ATTR_*` attributes are read-only attributes and don't receive default values in *attr*, because they exists only for bounded attribute objects.

int **pth_attr_set**(pth_attr_t *attr*, int *field*, ...);

This sets the attribute field *field* in *attr* to a value specified as an additional argument on the variable argument list. The following attribute *fields* and argument pairs can be used:

```
PTH_ATTR_PRIO            int
PTH_ATTR_NAME            char *
PTH_ATTR_DISPATCHES      int
PTH_ATTR_JOINABLE        int
PTH_ATTR_CANCEL_STATE    unsigned int
PTH_ATTR_STACK_SIZE      unsigned int
PTH_ATTR_STACK_ADDR      char *
```

int **pth_attr_get**(pth_attr_t *attr*, int *field*, ...);

This retrieves the attribute field *field* in *attr* and stores its value in the variable specified through a pointer in an additional argument on the variable argument list. The following *fields* and argument pairs can be used:

```
PTH_ATTR_PRIO            int *
PTH_ATTR_NAME            char **
PTH_ATTR_DISPATCHES      int *
PTH_ATTR_JOINABLE        int *
PTH_ATTR_CANCEL_STATE    unsigned int *
PTH_ATTR_STACK_SIZE      unsigned int *
PTH_ATTR_STACK_ADDR      char **
PTH_ATTR_TIME_SPAWN      pth_time_t *
PTH_ATTR_TIME_LAST       pth_time_t *
PTH_ATTR_TIME_RAN        pth_time_t *
PTH_ATTR_START_FUNC      void *(**)(void *)
PTH_ATTR_START_ARG       void **
PTH_ATTR_STATE           pth_state_t *
PTH_ATTR_EVENTS          pth_event_t *
PTH_ATTR_BOUND           int *
```

int **pth_attr_destroy**(pth_attr_t *attr*);

This destroys a attribute object *attr*. After this *attr* is no longer a valid attribute object.

### Thread Control

The following functions control the threading itself and make up the main API of the **Pth** library.

pth_t **pth_spawn**(pth_attr_t *attr*, void *(**entry*)(void *), void **arg*);

This spawns a new thread with the attributes given in *attr* (or `PTH_ATTR_DEFAULT` for default attributes − which means that thread priority, joinability and cancel state are inherited from the current thread) with the starting point at routine *entry*; the dispatch count is not inherited from the current thread if *attr* is not specified − rather, it is initialized to zero. This entry routine is called as 'pth_exit(*entry*(*arg*))' inside the new thread unit, i.e., *entry*'s return value is fed to an implicit *pth_exit*(3). So the thread can also exit by just returning. Nevertheless the thread can also exit explicitly at any time by calling *pth_exit*(3). But keep in mind that calling the POSIX function *exit*(3) still terminates the complete process and not just the current thread.

There is no **Pth**−internal limit on the number of threads one can spawn, except the limit implied by the available virtual memory. **Pth** internally keeps track of thread in dynamic data structures. The function returns `NULL` on error.

int **pth_once**(pth_once_t **ctrlvar*, void (**func*)(void *), void **arg*);

This is a convenience function which uses a control variable of type `pth_once_t` to make sure a constructor function *func* is called only once as '*func*(*arg*)' in the system. In other words: Only the first call to *pth_once*(3) by any thread in the system succeeds. The variable referenced via *ctrlvar* should be declared as '`pth_once_t` *variable-name* = `PTH_ONCE_INIT`;' before calling this function.

pth_t **pth_self**(void);

This just returns the unique thread handle of the currently running thread. This handle itself has to be treated as an opaque entity by the application. It's usually used as an argument to other functions who require an argument of type `pth_t`.

int **pth_suspend**(pth_t *tid*);

This suspends a thread *tid* until it is manually resumed again via *pth_resume*(3). For this, the thread is moved to the **SUSPENDED** queue and this way is completely out of the scheduler's event handling and thread dispatching scope. Suspending the current thread is not allowed. The function returns `TRUE` on success and `FALSE` on errors.

int **pth_resume**(pth_t *tid*);

This function resumes a previously suspended thread *tid*, i.e. *tid* has to stay on the **SUSPENDED** queue. The thread is moved to the **NEW**, **READY** or **WAITING** queue (dependent on what its state was when the *pth_suspend*(3) call were made) and this way again enters the event handling and thread dispatching scope of the scheduler. The function returns `TRUE` on success and `FALSE` on errors.

int **pth_raise**(pth_t *tid*, int *sig*)

This function raises a signal for delivery to thread *tid* only. When one just raises a signal via *raise*(3) or *kill*(2), its delivered to an arbitrary thread which has this signal not blocked. With *pth_raise*(3) one can send a signal to a thread and its guarantees that only this thread gets the signal delivered. But keep in mind that nevertheless the signals *action* is still configured *process*−wide. When *sig* is 0 plain thread checking is performed, i.e., '`pth_raise(tid, 0)`' returns `TRUE` when thread *tid* still exists in the **PTH** system but doesn't send any signal to it.

int **pth_yield**(pth_t *tid*);

This explicitly yields back the execution control to the scheduler thread. Usually the execution is implicitly transferred back to the scheduler when a thread waits for an event. But when a thread has to do larger CPU bursts, it can be reasonable to interrupt it explicitly by doing a few *pth_yield*(3) calls to give other threads a chance to execute, too. This obviously is the cooperating part of **Pth**. A thread *has not* to yield execution, of course. But when you want to program a server application with good response times the threads should be cooperative, i.e., when they should split their CPU bursts into smaller units with this call.

Usually one specifies *tid* as `NULL` to indicate to the scheduler that it can freely decide which thread to dispatch next. But if one wants to indicate to the scheduler that a particular thread should be favored on the next dispatching step, one can specify this thread explicitly. This allows the usage of the old concept of *coroutines* where a thread/routine switches to a particular cooperating thread. If *tid* is not `NULL` and points to a *new* or *ready* thread, it is guaranteed that this thread receives execution control on the next dispatching step. If *tid* is in a different state (that is, not in `PTH_STATE_NEW` or `PTH_STATE_READY`) an error is reported.

The function usually returns `TRUE` for success and only `FALSE` (with `errno` set to `EINVAL`) if *tid* specified an invalid or still not new or ready thread.

int **pth_nap**(pth_time_t *naptime*);

This functions suspends the execution of the current thread until *naptime* is elapsed. *naptime* is of type `pth_time_t` and this way has theoretically a resolution of one microsecond. In practice you should neither rely on this nor that the thread is awakened exactly after *naptime* has elapsed. It's only guarantees that the thread will sleep at least *naptime*. But because of the non-preemptive nature of **Pth** it can last longer (when another thread kept the CPU for a long time). Additionally the resolution is dependent of the implementation of timers by the operating system and these usually have only a resolution of 10 microseconds or larger. But usually this isn't important for an application unless it tries to use this facility for real time tasks.

int **pth_wait**(pth_event_t *ev*);

This is the link between the scheduler and the event facility (see below for the various *pth_event_xxx()* functions). It's modeled like *select*(2), i.e., one gives this function one or more events (in the event ring specified by *ev*) on which the current thread wants to wait. The scheduler awakes the thread when one ore more of them occurred or failed after tagging them as such. The *ev* argument is a *pointer* to an event ring which isn't changed except for the tagging. *pth_wait*(3) returns the number of occurred or failed events and the application can use *pth_event_status*(3) to test which events occurred or failed.

int **pth_cancel**(pth_t *tid*);

This cancels a thread *tid*. How the cancellation is done depends on the cancellation state of *tid* which the thread can configure itself. When its state is `PTH_CANCEL_DISABLE` a cancellation request is just made pending. When it is `PTH_CANCEL_ENABLE` it depends on the cancellation type what is performed. When its `PTH_CANCEL_DEFERRED` again the cancellation request is just made pending. But when its `PTH_CANCEL_ASYNCHRONOUS` the thread is immediately canceled before *pth_cancel*(3) returns. The effect of a thread cancellation is equal to implicitly forcing the thread to call '`pth_exit(PTH_CANCELED)`' at one of his cancellation points. In **Pth** thread enter a cancellation point either explicitly via *pth_cancel_point*(3) or implicitly by waiting for an event.

int **pth_abort**(pth_t *tid*);

This is the cruel way to cancel a thread *tid*. When it's already dead and waits to be joined it just joins it (via '`pth_join(`*tid*`, NULL)`') and this way kicks it out of the system. Else it forces the thread to be not joinable and to allow asynchronous cancellation and then cancels it via '`pth_cancel(`*tid*`)`'.

int **pth_join**(pth_t *tid*, void ***value*);

This joins the current thread with the thread specified via *tid*. It first suspends the current thread until the *tid* thread has terminated. Then it is awakened and stores the value of *tid*'s *pth_exit*(3) call into **value* (if *value* and not `NULL`) and returns to the caller. A thread can be joined only when it has the attribute `PTH_ATTR_JOINABLE` set to `TRUE` (the default). A thread can only be joined once, i.e., after the *pth_join*(3) call the thread *tid* is completely removed from the system.

void **pth_exit**(void **value*);

This terminates the current thread. Whether it's immediately removed from the system or inserted into the dead queue of the scheduler depends on its join type which was specified at spawning time. If it has the attribute `PTH_ATTR_JOINABLE` set to `FALSE`, it's immediately removed and *value* is ignored. Else the thread is inserted into the dead queue and *value* remembered for a subsequent *pth_join*(3) call by another thread.

### Utilities

Utility functions.

int **pth_fdmode**(int *fd*, int *mode*);

This switches the non-blocking mode flag on file descriptor *fd*. The argument *mode* can be `PTH_FDMODE_BLOCK` for switching *fd* into blocking I/O mode, `PTH_FDMODE_NONBLOCK` for switching *fd* into non-blocking I/O mode or `PTH_FDMODE_POLL` for just polling the current mode. The current mode is returned (either `PTH_FDMODE_BLOCK` or `PTH_FDMODE_NONBLOCK`) or `PTH_FDMODE_ERROR` on error. Keep in mind that since **Pth** 1.1 there is no longer a requirement to manually switch a file descriptor into non-blocking mode in order to use it. This is automatically done temporarily inside **Pth**. Instead when you now switch a file descriptor explicitly into non-blocking mode, *pth_read* (3) or *pth_write* (3) will never block the current thread.

pth_time_t **pth_time**(long *sec*, long *usec*);

This is a constructor for a `pth_time_t` structure which is a convenient function to avoid temporary structure values. It returns a *pth_time_t* structure which holds the absolute time value specified by *sec* and *usec*.

pth_time_t **pth_timeout**(long *sec*, long *usec*);

This is a constructor for a `pth_time_t` structure which is a convenient function to avoid temporary structure values. It returns a *pth_time_t* structure which holds the absolute time value calculated by adding *sec* and *usec* to the current time.

Sfdisc_t ***pth_sfiodisc**(void);

This functions is always available, but only reasonably usable when **Pth** was built with **Sfio** support (`--with-sfio` option) and `PTH_EXT_SFIO` is then defined by `pth.h`. It is useful for applications which want to use the comprehensive **Sfio** I/O library with the **Pth** threading library. Then this function can be used to get an **Sfio** discipline structure (`Sfdisc_t`) which can be pushed onto **Sfio** streams (`Sfio_t`) in order to let this stream use *pth_read* (3)/*pth_write* (2) instead of *read* (2)/*write* (2). The benefit is that this way I/O on the **Sfio** stream does only block the current thread instead of the whole process. The application has to *free* (3) the `Sfdisc_t` structure when it is no longer needed. The Sfio package can be found at http://www.research.att.com/sw/tools/sfio/.

### Cancellation Management

**Pth** supports POSIX style thread cancellation via *pth_cancel* (3) and the following two related functions:

void **pth_cancel_state**(int *newstate*, int **oldstate*);

This manages the cancellation state of the current thread. When *oldstate* is not `NULL` the function stores the old cancellation state under the variable pointed to by *oldstate*. When *newstate* is not `0` it sets the new cancellation state. *oldstate* is created before *newstate* is set. A state is a combination of `PTH_CANCEL_ENABLE` or `PTH_CANCEL_DISABLE` and `PTH_CANCEL_DEFERRED` or `PTH_CANCEL_ASYNCHRONOUS`. `PTH_CANCEL_ENABLE|PTH_CANCEL_DEFERRED` (or `PTH_CANCEL_DEFAULT`) is the default state where cancellation is possible but only at cancellation points. Use `PTH_CANCEL_DISABLE` to complete disable cancellation for a thread and `PTH_CANCEL_ASYNCHRONOUS` for allowing asynchronous cancellations, i.e., cancellations which can happen at any time.

void **pth_cancel_point**(void);

This explicitly enter a cancellation point. When the current cancellation state is `PTH_CANCEL_DISABLE` or no cancellation request is pending, this has no side-effect and returns immediately. Else it calls '`pth_exit(PTH_CANCELED)`'.

### Event Handling

**Pth** has a very flexible event facility which is linked into the scheduler through the *pth_wait* (3) function. The following functions provide the handling of event rings.

pth_event_t **pth_event**(unsigned long *spec*, ...);

This creates a new event ring consisting of a single initial event. The type of the generated event is specified by *spec*. The following types are available:

`PTH_EVENT_FD`

This is a file descriptor event. One or more of `PTH_UNTIL_FD_READABLE`, `PTH_UNTIL_FD_WRITEABLE` or `PTH_UNTIL_FD_EXCEPTION` have to be OR-ed into *spec* to specify on which state of the file descriptor you want to wait. The file descriptor itself has to be given as an additional argument. Example: '`pth_event(PTH_EVENT_FD|PTH_UNTIL_FD_READABLE, fd)`'.

`PTH_EVENT_SELECT`

This is a multiple file descriptor event modeled directly after the *select*(2) call (actually it is also used to implement *pth_select*(3) internally). It's a convenient way to wait for a large set of file descriptors at once and at each file descriptor for a different type of state. Additionally as a nice side-effect one receives the number of file descriptors which causes the event to be occurred (using BSD semantics, i.e., when a file descriptor occurred in two sets it's counted twice). The arguments correspond directly to the *select*(2) function arguments except that there is no timeout argument (because timeouts already can be handled via `PTH_EVENT_TIME` events).

Example: '`pth_event(PTH_EVENT_SELECT, &rc, nfd, rfds, wfds, efds)`' where `rc` has to be of type '`int *`', `nfd` has to be of type '`int`' and `rfds`, `wfds` and `efds` have to be of type '`fd_set *`' (see *select*(2)). The number of occurred file descriptors are stored in `rc`.

`PTH_EVENT_SIGS`

This is a signal set event. The two additional arguments have to be a pointer to a signal set (type '`sigset_t *`') and a pointer to a signal number variable (type '`int *`'). This event waits until one of the signals in the signal set occurred. As a result the occurred signal number is stored in the second additional argument. Keep in mind that the **Pth** scheduler doesn't block signals automatically. So when you want to wait for a signal with this event you've to block it via *sigprocmask*(2) or it will be delivered without your notice. Example: '`sigemptyset(&set); sigaddset(&set, SIGINT); pth_event(PTH_EVENT_SIG, &set, &sig);`'.

`PTH_EVENT_TIME`

This is a time point event. The additional argument has to be of type `pth_time_t` (usually on-the-fly generated via *pth_time*(3)). This events waits until the specified time point has elapsed. Keep in mind that the value is an absolute time point and not an offset. When you want to wait for a specified amount of time, you've to add the current time to the offset (usually on-the-fly achieved via *pth_timeout*(3)). Example: '`pth_event(PTH_EVENT_TIME, pth_timeout(2,0))`'.

`PTH_EVENT_MSG`

This is a message port event. The additional argument has to be of type `pth_msgport_t`. This events waits until one or more messages were received on the specified message port. Example: '`pth_event(PTH_EVENT_MSG, mp)`'.

`PTH_EVENT_TID`

This is a thread event. The additional argument has to be of type `pth_t`. One of `PTH_UNTIL_TID_NEW`, `PTH_UNTIL_TID_READY`, `PTH_UNTIL_TID_WAITING` or `PTH_UNTIL_TID_DEAD` has to be OR-ed into *spec* to specify on which state of the thread you want to wait. Example: '`pth_event(PTH_EVENT_TID|PTH_UNTIL_TID_DEAD, tid)`'.

`PTH_EVENT_FUNC`

This is a custom callback function event. Three additional arguments have to be given with the following types: '`int (*)(void *)`', '`void *`' and '`pth_time_t`'. The first is a function pointer to a check function and the second argument is a user-supplied context value which is passed to this function. The scheduler calls this function on a regular basis (on his own scheduler stack, so be very careful!) and the thread is kept sleeping while the function returns `FALSE`. Once it returned `TRUE` the thread will be awakened. The check interval is defined by the third argument, i.e., the check function is polled again not until this amount of time elapsed. Example: '`pth_event(PTH_EVENT_FUNC, func, arg, pth_time(0,500000))`'.

unsigned long **pth_event_typeof**(pth_event_t *ev*);
: This returns the type of event *ev*. It's a combination of the describing `PTH_EVENT_XX` and `PTH_UNTIL_XX` value. This is especially useful to know which arguments have to be supplied to the *pth_event_extract*(3) function.

int **pth_event_extract**(pth_event_t *ev*, ...);
: When *pth_event*(3) is treated like *sprintf*(3), then this function is *sscanf*(3), i.e., it is the inverse operation of *pth_event*(3). This means that it can be used to extract the ingredients of an event. The ingredients are stored into variables which are given as pointers on the variable argument list. Which pointers have to be present depends on the event type and has to be determined by the caller before via *pth_event_typeof*(3).

: To make it clear, when you constructed *ev* via '`ev = pth_event(PTH_EVENT_FD, fd);`' you have to extract it via '`pth_event_extract(ev, &fd)`', etc. For multiple arguments of an event the order of the pointer arguments is the same as for *pth_event*(3). But always keep in mind that you have to always supply *pointers* to *variables* and these variables have to be of the same type as the argument of *pth_event*(3) required.

pth_event_t **pth_event_concat**(pth_event_t *ev*, ...);
: This concatenates one or more additional event rings to the event ring *ev* and returns *ev*. The end of the argument list has to be marked with a `NULL` argument. Use this function to create real events rings out of the single-event rings created by *pth_event*(3).

pth_event_t **pth_event_isolate**(pth_event_t *ev*);
: This isolates the event *ev* from possibly appended events in the event ring. When in *ev* only one event exists, this returns `NULL`. When remaining events exists, they form a new event ring which is returned.

pth_event_t **pth_event_walk**(pth_event_t *ev*, int *direction*);
: This walks to the next (when *direction* is `PTH_WALK_NEXT`) or previews (when *direction* is `PTH_WALK_PREV`) event in the event ring *ev* and returns this new reached event. Additionally `PTH_UNTIL_OCCURRED` can be OR-ed into *direction* to walk to the next/previous occurred event in the ring *ev*.

pth_status_t **pth_event_status**(pth_event_t *ev*);
: This returns the status of event *ev*. This is a fast operation because only a tag on *ev* is checked which was either set or still not set by the scheduler. In other words: This doesn't check the event itself, it just checks the last knowledge of the scheduler. The possible returned status codes are: `PTH_STATUS_PENDING` (event is still pending), `PTH_STATUS_OCCURRED` (event successfully occurred), `PTH_STATUS_FAILED` (event failed).

int **pth_event_free**(pth_event_t *ev*, int *mode*);
: This deallocates the event *ev* (when *mode* is `PTH_FREE_THIS`) or all events appended to the event ring under *ev* (when *mode* is `PTH_FREE_ALL`).

### Key-Based Storage

The following functions provide thread-local storage through unique keys similar to the POSIX **Pthread** API. Use this for thread specific global data.

int **pth_key_create**(pth_key_t **key*, void (**func*)(void *));
: This created a new unique key and stores it in *key*. Additionally *func* can specify a destructor function which is called on the current threads termination with the *key*.

int **pth_key_delete**(pth_key_t *key*);
: This explicitly destroys a key *key*.

int **pth_key_setdata**(pth_key_t *key*, const void **value*);
: This stores *value* under *key*.

void ***pth_key_getdata**(pth_key_t *key*);
: This retrieves the value under *key*.

### Message Port Communication

The following functions provide message ports which can be used for efficient and flexible inter-thread communication.

pth_msgport_t **pth_msgport_create**(const char **name*);
: This returns a pointer to a new message port. If name *name* is not `NULL`, the *name* can be used by other threads via *pth_msgport_find*(3) to find the message port in case they do not know directly the pointer to the message port.

void **pth_msgport_destroy**(pth_msgport_t *mp*);
: This destroys a message port *mp*. Before all pending messages on it are replied to their origin message port.

pth_msgport_t **pth_msgport_find**(const char **name*);
: This finds a message port in the system by *name* and returns the pointer to it.

int **pth_msgport_pending**(pth_msgport_t *mp*);
: This returns the number of pending messages on message port *mp*.

int **pth_msgport_put**(pth_msgport_t *mp*, pth_message_t **m*);
: This puts (or sends) a message *m* to message port *mp*.

pth_message_t ***pth_msgport_get**(pth_msgport_t *mp*);
: This gets (or receives) the top message from message port *mp*. Incoming messages are always kept in a queue, so there can be more pending messages, of course.

int **pth_msgport_reply**(pth_message_t **m*);
: This replies a message *m* to the message port of the sender.

### Thread Cleanups

Per-thread cleanup functions.

int **pth_cleanup_push**(void (**handler*)(void *), void **arg*);
: This pushes the routine *handler* onto the stack of cleanup routines for the current thread. These routines are called in LIFO order when the thread terminates.

int **pth_cleanup_pop**(int *execute*);
: This pops the top-most routine from the stack of cleanup routines for the current thread. When *execute* is `TRUE` the routine is additionally called.

### Process Forking

The following functions provide some special support for process forking situations inside the threading environment.

int **pth_atfork_push**(void (**prepare*)(void *), void (*)(void **parent*), void (*)(void **child*), void **arg*);
: This function declares forking handlers to be called before and after *pth_fork*(3), in the context of the thread that called *pth_fork*(3). The *prepare* handler is called before *fork*(2) processing commences. The *parent* handler is called after *fork*(2) processing completes in the parent process. The *child* handler is called after *fork*(2) processing completed in the child process. If no handling is desired at one or more of these three points, the corresponding handler can be given as `NULL`. Each handler is called with *arg* as the argument.

: The order of calls to *pth_atfork_push*(3) is significant. The *parent* and *child* handlers are called in the order in which they were established by calls to *pth_atfork_push*(3), i.e., FIFO. The *prepare* fork handlers are called in the opposite order, i.e., LIFO.

int **pth_atfork_pop**(void);

This removes the top-most handlers on the forking handler stack which were established with the last *pth_atfork_push*(3) call. It returns `FALSE` when no more handlers couldn't be removed from the stack.

pid_t **pth_fork**(void);

This is a variant of *fork*(2) with the difference that the current thread only is forked into a separate process, i.e., in the parent process nothing changes while in the child process all threads are gone except for the scheduler and the calling thread. When you really want to duplicate all threads in the current process you should use *fork*(2) directly. But this is usually not reasonable. Additionally this function takes care of forking handlers as established by *pth_fork_push*(3).

**Synchronization**

The following functions provide synchronization support via mutual exclusion locks (**mutex**), read-write locks (**rwlock**), condition variables (**cond**) and barriers (**barrier**). Keep in mind that in a non-preemptive threading system like **Pth** this might sound unnecessary at the first look, because a thread isn't interrupted by the system. Actually when you have a critical code section which doesn't contain any *pth_xxx()* functions, you don't need any mutex to protect it, of course.

But when your critical code section contains any *pth_xxx()* function the chance is high that these temporarily switch to the scheduler. And this way other threads can make progress and enter your critical code section, too. This is especially true for critical code sections which implicitly or explicitly use the event mechanism.

int **pth_mutex_init**(pth_mutex_t **mutex*);

This dynamically initializes a mutex variable of type '`pth_mutex_t`'. Alternatively one can also use static initialization via '`pth_mutex_t mutex = PTH_MUTEX_INIT`'.

int **pth_mutex_acquire**(pth_mutex_t **mutex*, int *try*, pth_event_t *ev*);

This acquires a mutex *mutex*. If the mutex is already locked by another thread, the current threads execution is suspended until the mutex is unlocked again or additionally the extra events in *ev* occurred (when *ev* is not `NULL`). Recursive locking is explicitly supported, i.e., a thread is allowed to acquire a mutex more than once before its released. But it then also has be released the same number of times until the mutex is again lockable by others. When *try* is `TRUE` this function never suspends execution. Instead it returns `FALSE` with `errno` set to `EBUSY`.

int **pth_mutex_release**(pth_mutex_t **mutex*);

This decrements the recursion locking count on *mutex* and when it is zero it releases the mutex *mutex*.

int **pth_rwlock_init**(pth_rwlock_t **rwlock*);

This dynamically initializes a read-write lock variable of type '`pth_rwlock_t`'. Alternatively one can also use static initialization via '`pth_rwlock_t rwlock = PTH_RWLOCK_INIT`'.

int **pth_rwlock_acquire**(pth_rwlock_t **rwlock*, int *op*, int *try*, pth_event_t *ev*);

This acquires a read-only (when *op* is `PTH_RWLOCK_RD`) or a read-write (when *op* is `PTH_RWLOCK_RW`) lock *rwlock*. When the lock is only locked by other threads in read-only mode, the lock succeeds. But when one thread holds a read-write lock, all locking attempts suspend the current thread until this lock is released again. Additionally in *ev* events can be given to let the locking timeout, etc. When *try* is `TRUE` this function never suspends execution. Instead it returns `FALSE` with `errno` set to `EBUSY`.

int **pth_rwlock_release**(pth_rwlock_t **rwlock*);

This releases a previously acquired (read−only or read−write) lock.

int **pth_cond_init**(pth_cond_t **cond*);

This dynamically initializes a condition variable variable of type '`pth_cond_t`'. Alternatively one can also use static initialization via '`pth_cond_t cond = PTH_COND_INIT`'.

int **pth_cond_await**(pth_cond_t **cond*, pth_mutex_t **mutex*, pth_event_t *ev*);
: This awaits a condition situation. The caller has to follow the semantics of the POSIX condition variables: *mutex* has to be acquired before this function is called. The execution of the current thread is then suspended either until the events in *ev* occurred (when *ev* is not `NULL`) or *cond* was notified by another thread via *pth_cond_notify*(3). While the thread is waiting, *mutex* is released. Before it returns *mutex* is reacquired.

int **pth_cond_notify**(pth_cond_t **cond*, int *broadcast*);
: This notified one or all threads which are waiting on *cond*. When *broadcast* is `TRUE` all thread are notified, else only a single (unspecified) one.

int **pth_barrier_init**(pth_barrier_t **barrier*, int *threshold*);
: This dynamically initializes a barrier variable of type '`pth_barrier_t`'. Alternatively one can also use static initialization via '`pth_barrier_t barrier = PTH_BARRIER_INIT(`*threadhold*`)`'.

int **pth_barrier_reach**(pth_barrier_t **barrier*);
: This function reaches a barrier *barrier*. If this is the last thread (as specified by *threshold* on init of *barrier*) all threads are awakened. Else the current thread is suspended until the last thread reached the barrier and this way awakes all threads. The function returns (beside `FALSE` on error) the value `TRUE` for any thread which neither reached the barrier as the first nor the last thread; `PTH_BARRIER_HEADLIGHT` for the thread which reached the barrier as the first thread and `PTH_BARRIER_TAILLIGHT` for the thread which reached the barrier as the last thread.

### User-Space Context

The following functions provide a stand-alone sub-API for user-space context switching. It internally is based on the same underlying machine context switching mechanism the threads in **GNU Pth** are based on. Hence these functions you can use for implementing your own simple user-space threads. The `pth_uctx_t` context is somewhat modeled after POSIX *ucontext*(3).

The time required to create (via *pth_uctx_make*(3)) a user-space context can range from just a few microseconds up to a more dramatical time (depending on the machine context switching method which is available on the platform). On the other hand, the raw performance in switching the user-space contexts is always very good (nearly independent of the used machine context switching method). For instance, on an Intel Pentium-III CPU with 800Mhz running under FreeBSD 4 one usually achieves about 260,000 user-space context switches (via *pth_uctx_switch*(3)) per second.

int **pth_uctx_create**(pth_uctx_t **uctx*);
: This function creates a user-space context and stores it into *uctx*. There is still no underlying user-space context configured. You still have to do this with *pth_uctx_make*(3). On success, this function returns `TRUE`, else `FALSE`.

int **pth_uctx_make**(pth_uctx_t *uctx*, char **sk_addr*, size_t *sk_size*, const sigset_t **sigmask*, void (**start_func*)(void *), void **start_arg*, pth_uctx_t *uctx_after*);
: This function makes a new user-space context in *uctx* which will operate on the run-time stack *sk_addr* (which is of maximum size *sk_size*), with the signals in *sigmask* blocked (if *sigmask* is not `NULL`) and starting to execute with the call *start_func*(*start_arg*). If *sk_addr* is `NULL`, a stack is dynamically allocated. The stack size *sk_size* has to be at least 16384 (16KB). If the start function *start_func* returns and *uctx_after* is not `NULL`, an implicit user-space context switch to this context is performed. Else (if *uctx_after* is `NULL`) the process is terminated with *exit*(3). This function is somewhat modeled after POSIX *makecontext*(3). On success, this function returns `TRUE`, else `FALSE`.

int **pth_uctx_switch**(pth_uctx_t *uctx_from*, pth_uctx_t *uctx_to*);
: This function saves the current user-space context in *uctx_from* for later restoring by another call to *pth_uctx_switch*(3) and restores the new user-space context from *uctx_to*, which previously had to be set with either a previous call to *pth_uctx_switch*(3) or initially by *pth_uctx_make*(3). This function is somewhat modeled after POSIX *swapcontext*(3). If *uctx_from* or *uctx_to* are `NULL` or if *uctx_to* contains no valid user-space context, `FALSE` is returned instead of `TRUE`. These are the only errors possible.

int **pth_uctx_destroy**(pth_uctx_t *uctx*);
This function destroys the user-space context in *uctx*. The run-time stack associated with the user-space context is deallocated only if it was not given by the application (see *sk_addr* of *pth_uctx_create* (3)). If *uctx* is `NULL`, `FALSE` is returned instead of `TRUE`. This is the only error possible.

### Generalized POSIX Replacement API

The following functions are generalized replacements functions for the POSIX API, i.e., they are similar to the functions under '**Standard POSIX Replacement API**' but all have an additional event argument which can be used for timeouts, etc.

int **pth_sigwait_ev**(const sigset_t **set*, int **sig*, pth_event_t *ev*);
This is equal to *pth_sigwait* (3) (see below), but has an additional event argument *ev*. When *pth_sigwait* (3) suspends the current threads execution it usually only uses the signal event on *set* to awake. With this function any number of extra events can be used to awake the current thread (remember that *ev* actually is an event *ring*).

int **pth_connect_ev**(int *s*, const struct sockaddr **addr*, socklen_t *addrlen*, pth_event_t *ev*);
This is equal to *pth_connect* (3) (see below), but has an additional event argument *ev*. When *pth_connect* (3) suspends the current threads execution it usually only uses the I/O event on *s* to awake. With this function any number of extra events can be used to awake the current thread (remember that *ev* actually is an event *ring*).

int **pth_accept_ev**(int *s*, struct sockaddr **addr*, socklen_t **addrlen*, pth_event_t *ev*);
This is equal to *pth_accept* (3) (see below), but has an additional event argument *ev*. When *pth_accept* (3) suspends the current threads execution it usually only uses the I/O event on *s* to awake. With this function any number of extra events can be used to awake the current thread (remember that *ev* actually is an event *ring*).

int **pth_select_ev**(int *nfd*, fd_set **rfds*, fd_set **wfds*, fd_set **efds*, struct timeval **timeout*, pth_event_t *ev*);
This is equal to *pth_select* (3) (see below), but has an additional event argument *ev*. When *pth_select* (3) suspends the current threads execution it usually only uses the I/O event on *rfds*, *wfds* and *efds* to awake. With this function any number of extra events can be used to awake the current thread (remember that *ev* actually is an event *ring*).

int **pth_poll_ev**(struct pollfd **fds*, unsigned int *nfd*, int *timeout*, pth_event_t *ev*);
This is equal to *pth_poll* (3) (see below), but has an additional event argument *ev*. When *pth_poll* (3) suspends the current threads execution it usually only uses the I/O event on *fds* to awake. With this function any number of extra events can be used to awake the current thread (remember that *ev* actually is an event *ring*).

ssize_t **pth_read_ev**(int *fd*, void **buf*, size_t *nbytes*, pth_event_t *ev*);
This is equal to *pth_read* (3) (see below), but has an additional event argument *ev*. When *pth_read* (3) suspends the current threads execution it usually only uses the I/O event on *fd* to awake. With this function any number of extra events can be used to awake the current thread (remember that *ev* actually is an event *ring*).

ssize_t **pth_readv_ev**(int *fd*, const struct iovec **iovec*, int *iovcnt*, pth_event_t *ev*);
This is equal to *pth_readv* (3) (see below), but has an additional event argument *ev*. When *pth_readv* (3) suspends the current threads execution it usually only uses the I/O event on *fd* to awake. With this function any number of extra events can be used to awake the current thread (remember that *ev* actually is an event *ring*).

ssize_t **pth_write_ev**(int *fd*, const void **buf*, size_t *nbytes*, pth_event_t *ev*);
This is equal to *pth_write* (3) (see below), but has an additional event argument *ev*. When *pth_write* (3) suspends the current threads execution it usually only uses the I/O event on *fd* to awake. With this function any number of extra events can be used to awake the current thread (remember that *ev* actually is an event *ring*).

ssize_t **pth_writev_ev**(int *fd*, const struct iovec **iovec*, int *iovcnt*, pth_event_t *ev*);

This is equal to *pth_writev*(3) (see below), but has an additional event argument *ev*. When *pth_writev*(3) suspends the current threads execution it usually only uses the I/O event on *fd* to awake. With this function any number of extra events can be used to awake the current thread (remember that *ev* actually is an event *ring*).

ssize_t **pth_recv_ev**(int *fd*, void **buf*, size_t *nbytes*, int *flags*, pth_event_t *ev*);

This is equal to *pth_recv*(3) (see below), but has an additional event argument *ev*. When *pth_recv*(3) suspends the current threads execution it usually only uses the I/O event on *fd* to awake. With this function any number of extra events can be used to awake the current thread (remember that *ev* actually is an event *ring*).

ssize_t **pth_recvfrom_ev**(int *fd*, void **buf*, size_t *nbytes*, int *flags*, struct sockaddr **from*, socklen_t **fromlen*, pth_event_t *ev*);

This is equal to *pth_recvfrom*(3) (see below), but has an additional event argument *ev*. When *pth_recvfrom*(3) suspends the current threads execution it usually only uses the I/O event on *fd* to awake. With this function any number of extra events can be used to awake the current thread (remember that *ev* actually is an event *ring*).

ssize_t **pth_send_ev**(int *fd*, const void **buf*, size_t *nbytes*, int *flags*, pth_event_t *ev*);

This is equal to *pth_send*(3) (see below), but has an additional event argument *ev*. When *pth_send*(3) suspends the current threads execution it usually only uses the I/O event on *fd* to awake. With this function any number of extra events can be used to awake the current thread (remember that *ev* actually is an event *ring*).

ssize_t **pth_sendto_ev**(int *fd*, const void **buf*, size_t *nbytes*, int *flags*, const struct sockaddr **to*, socklen_t *tolen*, pth_event_t *ev*);

This is equal to *pth_sendto*(3) (see below), but has an additional event argument *ev*. When *pth_sendto*(3) suspends the current threads execution it usually only uses the I/O event on *fd* to awake. With this function any number of extra events can be used to awake the current thread (remember that *ev* actually is an event *ring*).

### Standard POSIX Replacement API

The following functions are standard replacements functions for the POSIX API. The difference is mainly that they suspend the current thread only instead of the whole process in case the file descriptors will block.

int **pth_nanosleep**(const struct timespec **rqtp*, struct timespec **rmtp*);

This is a variant of the POSIX *nanosleep*(3) function. It suspends the current threads execution until the amount of time in *rqtp* elapsed. The thread is guaranteed to not wake up before this time, but because of the non-preemptive scheduling nature of **Pth**, it can be awakened later, of course. If *rmtp* is not `NULL`, the `timespec` structure it references is updated to contain the unslept amount (the request time minus the time actually slept time). The difference between *nanosleep*(3) and *pth_nanosleep*(3) is that that *pth_nanosleep*(3) suspends only the execution of the current thread and not the whole process.

int **pth_usleep**(unsigned int *usec*);

This is a variant of the 4.3BSD *usleep*(3) function. It suspends the current threads execution until *usec* microseconds (= *usec**1/1000000 sec) elapsed. The thread is guaranteed to not wake up before this time, but because of the non-preemptive scheduling nature of **Pth**, it can be awakened later, of course. The difference between *usleep*(3) and *pth_usleep*(3) is that that *pth_usleep*(3) suspends only the execution of the current thread and not the whole process.

unsigned int **pth_sleep**(unsigned int *sec*);

This is a variant of the POSIX *sleep*(3) function. It suspends the current threads execution until *sec* seconds elapsed. The thread is guaranteed to not wake up before this time, but because of the non-preemptive scheduling nature of **Pth**, it can be awakened later, of course. The difference between *sleep*(3) and *pth_sleep*(3) is that *pth_sleep*(3) suspends only the execution of the current thread and not the whole process.

pid_t **pth_waitpid**(pid_t *pid*, int **status*, int *options*);

This is a variant of the POSIX *waitpid*(2) function. It suspends the current threads execution until *status* information is available for a terminated child process *pid*. The difference between *waitpid*(2) and *pth_waitpid*(3) is that *pth_waitpid*(3) suspends only the execution of the current thread and not the whole process. For more details about the arguments and return code semantics see *waitpid*(2).

int **pth_system**(const char **cmd*);

This is a variant of the POSIX *system*(3) function. It executes the shell command *cmd* with Bourne Shell (`sh`) and suspends the current threads execution until this command terminates. The difference between *system*(3) and *pth_system*(3) is that *pth_system*(3) suspends only the execution of the current thread and not the whole process. For more details about the arguments and return code semantics see *system*(3).

int **pth_sigmask**(int *how*, const sigset_t **set*, sigset_t **oset*)

This is the **Pth** thread-related equivalent of POSIX *sigprocmask*(2) respectively *pthread_sigmask*(3). The arguments *how*, *set* and *oset* directly relate to *sigprocmask*(2), because **Pth** internally just uses *sigprocmask*(2) here. So alternatively you can also directly call *sigprocmask*(2), but for consistency reasons you should use this function *pth_sigmask*(3).

int **pth_sigwait**(const sigset_t **set*, int **sig*);

This is a variant of the POSIX.1c *sigwait*(3) function. It suspends the current threads execution until a signal in *set* occurred and stores the signal number in *sig*. The important point is that the signal is not delivered to a signal handler. Instead it's caught by the scheduler only in order to awake the *pth_sigwait()* call. The trick and noticeable point here is that this way you get an asynchronous aware application that is written completely synchronously. When you think about the problem of *asynchronous safe* functions you should recognize that this is a great benefit.

int **pth_connect**(int *s*, const struct sockaddr **addr*, socklen_t *addrlen*);

This is a variant of the 4.2BSD *connect*(2) function. It establishes a connection on a socket *s* to target specified in *addr* and *addrlen*. The difference between *connect*(2) and *pth_connect*(3) is that *pth_connect*(3) suspends only the execution of the current thread and not the whole process. For more details about the arguments and return code semantics see *connect*(2).

int **pth_accept**(int *s*, struct sockaddr **addr*, socklen_t **addrlen*);

This is a variant of the 4.2BSD *accept*(2) function. It accepts a connection on a socket by extracting the first connection request on the queue of pending connections, creating a new socket with the same properties of *s* and allocates a new file descriptor for the socket (which is returned). The difference between *accept*(2) and *pth_accept*(3) is that *pth_accept*(3) suspends only the execution of the current thread and not the whole process. For more details about the arguments and return code semantics see *accept*(2).

int **pth_select**(int *nfd*, fd_set **rfds*, fd_set **wfds*, fd_set **efds*, struct timeval **timeout*);

This is a variant of the 4.2BSD *select*(2) function. It examines the I/O descriptor sets whose addresses are passed in *rfds*, *wfds*, and *efds* to see if some of their descriptors are ready for reading, are ready for writing, or have an exceptional condition pending, respectively. For more details about the arguments and return code semantics see *select*(2).

int **pth_pselect**(int *nfd*, fd_set **rfds*, fd_set **wfds*, fd_set **efds*, const struct timespec **timeout*, const sigset_t **sigmask*);

This is a variant of the POSIX *pselect*(2) function, which in turn is a stronger variant of 4.2BSD *select*(2). The difference is that the higher-resolution `struct timespec` is passed instead of the lower-resolution `struct timeval` and that a signal mask is specified which is temporarily set while waiting for input. For more details about the arguments and return code semantics see *pselect*(2) and *select*(2).

int **pth_poll**(struct pollfd **fds*, unsigned int *nfd*, int *timeout*);

This is a variant of the SysV *poll*(2) function. It examines the I/O descriptors which are passed in the array *fds* to see if some of them are ready for reading, are ready for writing, or have an exceptional condition pending, respectively. For more details about the arguments and return code semantics see *poll* (2).

ssize_t **pth_read**(int *fd*, void **buf*, size_t *nbytes*);
: This is a variant of the POSIX *read* (2) function. It reads up to *nbytes* bytes into *buf* from file descriptor *fd*. The difference between *read* (2) and *pth_read* (2) is that *pth_read* (2) suspends execution of the current thread until the file descriptor is ready for reading. For more details about the arguments and return code semantics see *read* (2).

ssize_t **pth_readv**(int *fd*, const struct iovec **iovec*, int *iovcnt*);
: This is a variant of the POSIX *readv* (2) function. It reads data from file descriptor *fd* into the first *iovcnt* rows of the *iov* vector. The difference between *readv* (2) and *pth_readv* (2) is that *pth_readv* (2) suspends execution of the current thread until the file descriptor is ready for reading. For more details about the arguments and return code semantics see *readv* (2).

ssize_t **pth_write**(int *fd*, const void **buf*, size_t *nbytes*);
: This is a variant of the POSIX *write* (2) function. It writes *nbytes* bytes from *buf* to file descriptor *fd*. The difference between *write* (2) and *pth_write* (2) is that *pth_write* (2) suspends execution of the current thread until the file descriptor is ready for writing. For more details about the arguments and return code semantics see *write* (2).

ssize_t **pth_writev**(int *fd*, const struct iovec **iovec*, int *iovcnt*);
: This is a variant of the POSIX *writev* (2) function. It writes data to file descriptor *fd* from the first *iovcnt* rows of the *iov* vector. The difference between *writev* (2) and *pth_writev* (2) is that *pth_writev* (2) suspends execution of the current thread until the file descriptor is ready for reading. For more details about the arguments and return code semantics see *writev* (2).

ssize_t **pth_pread**(int *fd*, void **buf*, size_t *nbytes*, off_t *offset*);
: This is a variant of the POSIX *pread* (3) function. It performs the same action as a regular *read* (2), except that it reads from a given position in the file without changing the file pointer. The first three arguments are the same as for *pth_read* (3) with the addition of a fourth argument *offset* for the desired position inside the file.

ssize_t **pth_pwrite**(int *fd*, const void **buf*, size_t *nbytes*, off_t *offset*);
: This is a variant of the POSIX *pwrite* (3) function. It performs the same action as a regular *write* (2), except that it writes to a given position in the file without changing the file pointer. The first three arguments are the same as for *pth_write* (3) with the addition of a fourth argument *offset* for the desired position inside the file.

ssize_t **pth_recv**(int *fd*, void **buf*, size_t *nbytes*, int *flags*);
: This is a variant of the SUSv2 *recv* (2) function and equal to ''pth_recvfrom(fd, buf, nbytes, flags, NULL, 0)''.

ssize_t **pth_recvfrom**(int *fd*, void **buf*, size_t *nbytes*, int *flags*, struct sockaddr **from*, socklen_t **fromlen*);
: This is a variant of the SUSv2 *recvfrom* (2) function. It reads up to *nbytes* bytes into *buf* from file descriptor *fd* while using *flags* and *from*/*fromlen*. The difference between *recvfrom* (2) and *pth_recvfrom* (2) is that *pth_recvfrom* (2) suspends execution of the current thread until the file descriptor is ready for reading. For more details about the arguments and return code semantics see *recvfrom* (2).

ssize_t **pth_send**(int *fd*, const void **buf*, size_t *nbytes*, int *flags*);
: This is a variant of the SUSv2 *send* (2) function and equal to ''pth_sendto(fd, buf, nbytes, flags, NULL, 0)''.

ssize_t **pth_sendto**(int *fd*, const void **buf*, size_t *nbytes*, int *flags*, const struct sockaddr **to*, socklen_t *tolen*);
: This is a variant of the SUSv2 *sendto* (2) function. It writes *nbytes* bytes from *buf* to file descriptor *fd* while using *flags* and *to*/*tolen*. The difference between *sendto* (2) and *pth_sendto* (2) is that *pth_sendto* (2) suspends execution of the current thread until the file descriptor is ready for writing. For more details about the arguments and return code semantics see *sendto* (2).

## EXAMPLE

The following example is a useless server which does nothing more than listening on TCP port 12345 and displaying the current time to the socket when a connection was established. For each incoming connection a thread is spawned. Additionally, to see more multithreading, a useless ticker thread runs simultaneously which outputs the current time to `stderr` every 5 seconds. The example contains *no* error checking and is *only* intended to show you the look and feel of **Pth**.

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <signal.h>
#include <netdb.h>
#include <unistd.h>
#include "pth.h"

#define PORT 12345

/* the socket connection handler thread */
static void *handler(void *_arg)
{
    int fd = (int)_arg;
    time_t now;
    char *ct;

    now = time(NULL);
    ct = ctime(&now);
    pth_write(fd, ct, strlen(ct));
    close(fd);
    return NULL;
}

/* the stderr time ticker thread */
static void *ticker(void *_arg)
{
    time_t now;
    char *ct;
    float load;

    for (;;) {
        pth_sleep(5);
        now = time(NULL);
        ct = ctime(&now);
        ct[strlen(ct)-1] = '\0';
        pth_ctrl(PTH_CTRL_GETAVLOAD, &load);
        printf("ticker: time: %s, average load: %.2f\n", ct, load);
    }
}
```

```
/* the main thread/procedure */
int main(int argc, char *argv[])
{
    pth_attr_t attr;
    struct sockaddr_in sar;
    struct protoent *pe;
    struct sockaddr_in peer_addr;
    int peer_len;
    int sa, sw;
    int port;

    pth_init();
    signal(SIGPIPE, SIG_IGN);

    attr = pth_attr_new();
    pth_attr_set(attr, PTH_ATTR_NAME, "ticker");
    pth_attr_set(attr, PTH_ATTR_STACK_SIZE, 64*1024);
    pth_attr_set(attr, PTH_ATTR_JOINABLE, FALSE);
    pth_spawn(attr, ticker, NULL);

    pe = getprotobyname("tcp");
    sa = socket(AF_INET, SOCK_STREAM, pe->p_proto);
    sar.sin_family = AF_INET;
    sar.sin_addr.s_addr = INADDR_ANY;
    sar.sin_port = htons(PORT);
    bind(sa, (struct sockaddr *)&sar, sizeof(struct sockaddr_in));
    listen(sa, 10);

    pth_attr_set(attr, PTH_ATTR_NAME, "handler");
    for (;;) {
        peer_len = sizeof(peer_addr);
        sw = pth_accept(sa, (struct sockaddr *)&peer_addr, &peer_len);
        pth_spawn(attr, handler, (void *)sw);
    }
}
```

## BUILD ENVIRONMENTS

In this section we will discuss the canonical ways to establish the build environment for a **Pth** based program. The possibilities supported by **Pth** range from very simple environments to rather complex ones.

### Manual Build Environment (Novice)

As a first example, assume we have the above test program staying in the source file `foo.c`. Then we can create a very simple build environment by just adding the following `Makefile`:

```
$ vi Makefile
| CC      = cc
| CFLAGS  = `pth-config --cflags`
| LDFLAGS = `pth-config --ldflags`
| LIBS    = `pth-config --libs`
|
| all: foo
| foo: foo.o
|     $(CC) $(LDFLAGS) -o foo foo.o $(LIBS)
| foo.o: foo.c
|     $(CC) $(CFLAGS) -c foo.c
| clean:
|     rm -f foo foo.o
```

This imports the necessary compiler and linker flags on-the-fly from the **Pth** installation via its `pth-config` program. This approach is straight-forward and works fine for small projects.

**Autoconf Build Environment (Advanced)**

The previous approach is simple but inflexible. First, to speed up building, it would be nice to not expand the compiler and linker flags every time the compiler is started. Second, it would be useful to also be able to build against uninstalled **Pth**, that is, against a **Pth** source tree which was just configured and built, but not installed. Third, it would be also useful to allow checking of the **Pth** version to make sure it is at least a minimum required version. And finally, it would be also great to make sure **Pth** works correctly by first performing some sanity compile and run-time checks. All this can be done if we use GNU **autoconf** and the `AC_CHECK_PTH` macro provided by **Pth**. For this, we establish the following three files:

First we again need the `Makefile`, but this time it contains **autoconf** placeholders and additional cleanup targets. And we create it under the name `Makefile.in`, because it is now an input file for **autoconf**:

```
$ vi Makefile.in
| CC      = @CC@
| CFLAGS  = @CFLAGS@
| LDFLAGS = @LDFLAGS@
| LIBS    = @LIBS@
|
| all: foo
| foo: foo.o
|     $(CC) $(LDFLAGS) -o foo foo.o $(LIBS)
| foo.o: foo.c
|     $(CC) $(CFLAGS) -c foo.c
| clean:
|     rm -f foo foo.o
| distclean:
|     rm -f foo foo.o
|     rm -f config.log config.status config.cache
|     rm -f Makefile
```

Because **autoconf** generates additional files, we added a canonical `distclean` target which cleans this up. Secondly, we wrote `configure.ac`, a (minimal) **autoconf** script specification:

```
$ vi configure.ac
| AC_INIT(Makefile.in)
| AC_CHECK_PTH(1.3.0)
| AC_OUTPUT(Makefile)
```

Then we let **autoconf**'s `aclocal` program generate for us an `aclocal.m4` file containing **Pth**'s `AC_CHECK_PTH` macro. Then we generate the final `configure` script out of this `aclocal.m4` file and the `configure.ac` file:

```
$ aclocal --acdir=`pth-config --acdir`
$ autoconf
```

After these steps, the working directory should look similar to this:

```
$ ls -l
-rw-r--r--  1 rse  users    176 Nov  3 11:11 Makefile.in
-rw-r--r--  1 rse  users  15314 Nov  3 11:16 aclocal.m4
-rwxr-xr-x  1 rse  users  52045 Nov  3 11:16 configure
-rw-r--r--  1 rse  users     63 Nov  3 11:11 configure.ac
-rw-r--r--  1 rse  users   4227 Nov  3 11:11 foo.c
```

If we now run `configure` we get a correct `Makefile` which immediately can be used to build `foo` (assuming that **Pth** is already installed somewhere, so that `pth-config` is in `$PATH`):

```
$ ./configure
creating cache ./config.cache
checking for gcc... gcc
checking whether the C compiler (gcc   ) works... yes
checking whether the C compiler (gcc   ) is a cross-compiler... no
checking whether we are using GNU C... yes
checking whether gcc accepts -g... yes
checking how to run the C preprocessor... gcc -E
checking for GNU Pth... version 1.3.0, installed under /usr/local
updating cache ./config.cache
creating ./config.status
creating Makefile
rse@en1:/e/gnu/pth/ac
$ make
gcc -g -O2 -I/usr/local/include -c foo.c
gcc -L/usr/local/lib -o foo foo.o -lpth
```

If **Pth** is installed in non-standard locations or `pth-config` is not in `$PATH`, one just has to drop the `configure` script a note about the location by running `configure` with the option `--with-pth=`*dir* (where *dir* is the argument which was used with the `--prefix` option when **Pth** was installed).

### Autoconf Build Environment with Local Copy of Pth (Expert)

Finally let us assume the `foo` program stays under either a *GPL* or *LGPL* distribution license and we want to make it a stand-alone package for easier distribution and installation. That is, we don't want to oblige the end-user to install **Pth** just to allow our `foo` package to compile. For this, it is a convenient practice to include the required libraries (here **Pth**) into the source tree of the package (here `foo`). **Pth** ships with all necessary support to allow us to easily achieve this approach. Say, we want **Pth** in a subdirectory named `pth/` and this directory should be seamlessly integrated into the configuration and build process of `foo`.

First we again start with the `Makefile.in`, but this time it is a more advanced version which supports subdirectory movement:

```
$ vi Makefile.in
| CC      = @CC@
| CFLAGS  = @CFLAGS@
| LDFLAGS = @LDFLAGS@
| LIBS    = @LIBS@
|
| SUBDIRS = pth
|
| all: subdirs_all foo
|
| subdirs_all:
|     @$(MAKE) $(MFLAGS) subdirs TARGET=all
| subdirs_clean:
|     @$(MAKE) $(MFLAGS) subdirs TARGET=clean
| subdirs_distclean:
|     @$(MAKE) $(MFLAGS) subdirs TARGET=distclean
| subdirs:
|     @for subdir in $(SUBDIRS); do \
|         echo "===> $$subdir ($(TARGET))"; \
|         (cd $$subdir; $(MAKE) $(MFLAGS) $(TARGET) || exit 1) || exit 1; \
|         echo "<=== $$subdir"; \
|     done
|
| foo: foo.o
|     $(CC) $(LDFLAGS) -o foo foo.o $(LIBS)
| foo.o: foo.c
|     $(CC) $(CFLAGS) -c foo.c
|
| clean: subdirs_clean
|     rm -f foo foo.o
| distclean: subdirs_distclean
|     rm -f foo foo.o
|     rm -f config.log config.status config.cache
|     rm -f Makefile
```

Then we create a slightly different **autoconf** script `configure.ac`:

```
$ vi configure.ac
| AC_INIT(Makefile.in)
| AC_CONFIG_AUX_DIR(pth)
| AC_CHECK_PTH(1.3.0, subdir:pth --disable-tests)
| AC_CONFIG_SUBDIRS(pth)
| AC_OUTPUT(Makefile)
```

Here we provided a default value for `foo`'s `--with-pth` option as the second argument to `AC_CHECK_PTH` which indicates that **Pth** can be found in the subdirectory named `pth/`. Additionally we specified that the `--disable-tests` option of **Pth** should be passed to the `pth/` subdirectory, because we need only to build the **Pth** library itself. And we added a `AC_CONFIG_SUBDIR` call which indicates to **autoconf** that it should configure the `pth/` subdirectory, too. The `AC_CONFIG_AUX_DIR` directive was added just to make **autoconf** happy, because it wants to find a `install.sh` or `shtool` script if `AC_CONFIG_SUBDIRS` is used.

Now we let **autoconf**'s `aclocal` program again generate for us an `aclocal.m4` file with the contents of **Pth**'s `AC_CHECK_PTH` macro. Finally we generate the `configure` script out of this `aclocal.m4` file and the `configure.ac` file.

```
$ aclocal --acdir=`pth-config --acdir`
$ autoconf
```

Now we have to create the `pth/` subdirectory itself. For this, we extract the **Pth** distribution to the `foo` source tree and just rename it to `pth/`:

```
$ gunzip <pth-X.Y.Z.tar.gz | tar xvf -
$ mv pth-X.Y.Z pth
```

Optionally to reduce the size of the `pth/` subdirectory, we can strip down the **Pth** sources to a minimum with the *striptease* feature:

```
$ cd pth
$ ./configure
$ make striptease
$ cd ..
```

After this the source tree of `foo` should look similar to this:

```
$ ls -l
-rw-r--r--  1 rse  users    709 Nov  3 11:51 Makefile.in
-rw-r--r--  1 rse  users  16431 Nov  3 12:20 aclocal.m4
-rwxr-xr-x  1 rse  users  57403 Nov  3 12:21 configure
-rw-r--r--  1 rse  users    129 Nov  3 12:21 configure.ac
-rw-r--r--  1 rse  users   4227 Nov  3 11:11 foo.c
drwxr-xr-x  2 rse  users   3584 Nov  3 12:36 pth
$ ls -l pth/
-rw-rw-r--  1 rse  users   26344 Nov  1 20:12 COPYING
-rw-rw-r--  1 rse  users    2042 Nov  3 12:36 Makefile.in
-rw-rw-r--  1 rse  users    3967 Nov  1 19:48 README
-rw-rw-r--  1 rse  users     340 Nov  3 12:36 README.1st
-rw-rw-r--  1 rse  users   28719 Oct 31 17:06 config.guess
-rw-rw-r--  1 rse  users   24274 Aug 18 13:31 config.sub
-rwxrwxr-x  1 rse  users  155141 Nov  3 12:36 configure
-rw-rw-r--  1 rse  users  162021 Nov  3 12:36 pth.c
-rw-rw-r--  1 rse  users   18687 Nov  2 15:19 pth.h.in
-rw-rw-r--  1 rse  users    5251 Oct 31 12:46 pth_acdef.h.in
-rw-rw-r--  1 rse  users    2120 Nov  1 11:27 pth_acmac.h.in
-rw-rw-r--  1 rse  users    2323 Nov  1 11:27 pth_p.h.in
-rw-rw-r--  1 rse  users     946 Nov  1 11:27 pth_vers.c
-rw-rw-r--  1 rse  users   26848 Nov  1 11:27 pthread.c
-rw-rw-r--  1 rse  users   18772 Nov  1 11:27 pthread.h.in
-rwxrwxr-x  1 rse  users   26188 Nov  3 12:36 shtool
```

Now when we configure and build the `foo` package it looks similar to this:

```
$ ./configure
creating cache ./config.cache
checking for gcc... gcc
checking whether the C compiler (gcc   ) works... yes
checking whether the C compiler (gcc   ) is a cross-compiler... no
checking whether we are using GNU C... yes
checking whether gcc accepts -g... yes
checking how to run the C preprocessor... gcc -E
checking for GNU Pth... version 1.3.0, local under pth
updating cache ./config.cache
creating ./config.status
creating Makefile
configuring in pth
running /bin/sh ./configure  --enable-subdir --enable-batch
--disable-tests --cache-file=.././config.cache --srcdir=.
loading cache .././config.cache
checking for gcc... (cached) gcc
checking whether the C compiler (gcc   ) works... yes
checking whether the C compiler (gcc   ) is a cross-compiler... no
[...]
$ make
===> pth (all)
./shtool scpp -o pth_p.h -t pth_p.h.in -Dcpp -Cintern -M '==#==' pth.c
pth_vers.c
gcc -c -I. -O2 -pipe pth.c
gcc -c -I. -O2 -pipe pth_vers.c
ar rc libpth.a pth.o pth_vers.o
ranlib libpth.a
<=== pth
gcc -g -O2 -Ipth -c foo.c
gcc -Lpth -o foo foo.o -lpth
```

As you can see, **autoconf** now automatically configures the local (stripped down) copy of **Pth** in the subdirectory `pth/` and the `Makefile` automatically builds the subdirectory, too.

## SYSTEM CALL WRAPPER FACILITY

**Pth** per default uses an explicit API, including the system calls. For instance you've to explicitly use *pth_read* (3) when you need a thread-aware *read* (3) and cannot expect that by just calling *read* (3) only the current thread is blocked. Instead with the standard *read* (3) call the whole process will be blocked. But because for some applications (mainly those consisting of lots of third-party stuff) this can be inconvenient. Here it's required that a call to *read* (3) 'magically' means *pth_read* (3). The problem here is that such magic **Pth** cannot provide per default because it's not really portable. Nevertheless **Pth** provides a two step approach to solve this problem:

### Soft System Call Mapping

This variant is available on all platforms and can *always* be enabled by building **Pth** with `--enable-syscall-soft`. This then triggers some `#define`'s in the `pth.h` header which map for instance *read* (3) to *pth_read* (3), etc. Currently the following functions are mapped: *fork* (2), *nanosleep* (3), *usleep* (3), *sleep* (3), *sigwait* (3), *waitpid* (2), *system* (3), *select* (2), *poll* (2), *connect* (2), *accept* (2), *read* (2), *write* (2), *recv* (2), *send* (2), *recvfrom* (2), *sendto* (2).

The drawback of this approach is just that really all source files of the application where these function calls occur have to include `pth.h`, of course. And this also means that existing libraries, including the vendor's **stdio**, usually will still block the whole process if one of its I/O functions block.

### Hard System Call Mapping

This variant is available only on those platforms where the *syscall*(2) function exists and there it can be enabled by building **Pth** with `--enable-syscall-hard`. This then builds wrapper functions (for instances *read*(3)) into the **Pth** library which internally call the real **Pth** replacement functions (*pth_read*(3)). Currently the following functions are mapped: *fork*(2), *nanosleep*(3), *usleep*(3), *sleep*(3), *waitpid*(2), *system*(3), *select*(2), *poll*(2), *connect*(2), *accept*(2), *read*(2), *write*(2).

The drawback of this approach is that it depends on *syscall*(2) interface and prototype conflicts can occur while building the wrapper functions due to different function signatures in the vendor C header files. But the advantage of this mapping variant is that the source files of the application where these function calls occur have not to include `pth.h` and that existing libraries, including the vendor's **stdio**, magically become thread-aware (and then block only the current thread).

## IMPLEMENTATION NOTES

**Pth** is very portable because it has only one part which perhaps has to be ported to new platforms (the machine context initialization). But it is written in a way which works on mostly all Unix platforms which support *makecontext*(2) or at least *sigstack*(2) or *sigaltstack*(2) [see `pth_mctx.c` for details]. Any other **Pth** code is POSIX and ANSI C based only.

The context switching is done via either SUSv2 *makecontext*(2) or POSIX make[sig]*setjmp*(3) and [sig]*longjmp*(3). Here all CPU registers, the program counter and the stack pointer are switched. Additionally the **Pth** dispatcher switches also the global Unix `errno` variable [see `pth_mctx.c` for details] and the signal mask (either implicitly via *sigsetjmp*(3) or in an emulated way via explicit *setprocmask*(2) calls).

The **Pth** event manager is mainly *select*(2) and *gettimeofday*(2) based, i.e., the current time is fetched via *gettimeofday*(2) once per context switch for time calculations and all I/O events are implemented via a single central *select*(2) call [see `pth_sched.c` for details].

The thread control block management is done via virtual priority queues without any additional data structure overhead. For this, the queue linkage attributes are part of the thread control blocks and the queues are actually implemented as rings with a selected element as the entry point [see `pth_tcb.h` and `pth_pqueue.c` for details].

Most time critical code sections (especially the dispatcher and event manager) are speeded up by inline functions (implemented as ANSI C pre-processor macros). Additionally any debugging code is *completely* removed from the source when not built with `-DPTH_DEBUG` (see Autoconf `--enable-debug` option), i.e., not only stub functions remain [see `pth_debug.c` for details].

## RESTRICTIONS

**Pth** (intentionally) provides no replacements for non-thread-safe functions (like *strtok*(3) which uses a static internal buffer) or synchronous system functions (like *gethostbyname*(3) which doesn't provide an asynchronous mode where it doesn't block). When you want to use those functions in your server application together with threads, you've to either link the application against special third-party libraries (or for thread−safe/reentrant functions possibly against an existing `libc_r` of the platform vendor). For an asynchronous DNS resolver library use the GNU **adns** package from Ian Jackson ( see http://www.gnu.org/software/adns/adns.html ).

## HISTORY

The **Pth** library was designed and implemented between February and July 1999 by *Ralf S. Engelschall* after evaluating numerous (mostly preemptive) thread libraries and after intensive discussions with *Peter Simons*, *Martin Kraemer*, *Lars Eilebrecht* and *Ralph Babel* related to an experimental (matrix based) non-preemptive C++ scheduler class written by *Peter Simons*.

**Pth** was then implemented in order to combine the *non-preemptive* approach of multithreading (which provides better portability and performance) with an API similar to the popular one found in **Pthread** libraries (which provides easy programming).

So the essential idea of the non-preemptive approach was taken over from *Peter Simons* scheduler. The priority based scheduling algorithm was suggested by *Martin Kraemer*. Some code inspiration also came from an experimental threading library (**rsthreads**) written by *Robert S. Thau* for an ancient internal test version of the Apache webserver. The concept and API of message ports was borrowed from AmigaOS' **Exec** subsystem. The concept and idea for the flexible event mechanism came from *Paul Vixie*'s **eventlib** (which can be found as a part of **BIND** v8).

## BUG REPORTS AND SUPPORT

If you think you have found a bug in **Pth**, you should send a report as complete as possible to *bug−pth@gnu.org*. If you can, please try to fix the problem and include a patch, made with '`diff -u3`', in your report. Always, at least, include a reasonable amount of description in your report to allow the author to deterministically reproduce the bug.

For further support you additionally can subscribe to the *pth−users@gnu.org* mailing list by sending an Email to *pth−users−request@gnu.org* with '`subscribe pth-users`' (or '`subscribe pth-users` *address*' if you want to subscribe from a particular Email *address*) in the body. Then you can discuss your issues with other **Pth** users by sending messages to *pth−users@gnu.org*. Currently (as of August 2000) you can reach about 110 Pth users on this mailing list. Old postings you can find at *http://www.mail−archive.com/pth−users@gnu.org/*.

## SEE ALSO

### Related Web Locations

'comp.programming.threads Newsgroup Archive', http://www.deja.com/topics_if.xp?search=topic&group=comp.programming.threads

'comp.programming.threads Frequently Asked Questions (F.A.Q.)', http://www.lambdacs.com/newsgroup/FAQ.html

'*Multithreading − Definitions and Guidelines*', Numeric Quest Inc 1998; http://www.numeric−quest.com/lang/multi−frame.html

'*The Single UNIX Specification, Version 2 − Threads*', The Open Group 1997; http://www.opengroup.org/onlinepubs /007908799/xsh/threads.html

SMI Thread Resources, Sun Microsystems Inc; http://www.sun.com/workshop/threads/

Bibliography on threads and multithreading, Torsten Amundsen; http://liinwww.ira.uka.de/bibliography/Os/threads.html

### Related Books

B. Nichols, D. Buttlar, J.P. Farrel: '*Pthreads Programming − A POSIX Standard for Better Multiprocessing*', O'Reilly 1996; ISBN 1−56592−115−1

B. Lewis, D. J. Berg: '*Multithreaded Programming with Pthreads*', Sun Microsystems Press, Prentice Hall 1998; ISBN 0−13−680729−1

B. Lewis, D. J. Berg: '*Threads Primer − A Guide To Multithreaded Programming*', Prentice Hall 1996; ISBN 0−13−443698−9

S. J. Norton, M. D. Dipasquale: '*Thread Time − The Multithreaded Programming Guide*', Prentice Hall 1997; ISBN 0−13−190067−6

D. R. Butenhof: '*Programming with POSIX Threads*', Addison Wesley 1997; ISBN 0−201−63392−2

### Related Manpages

*pth−config* (1), *pthread* (3).

*getcontext* (2), *setcontext* (2), *makecontext* (2), *swapcontext* (2), *sigstack* (2), *sigaltstack* (2), *sigaction* (2), *sigemptyset* (2), *sigaddset* (2), *sigprocmask* (2), *sigsuspend* (2), *sigsetjmp* (3), *siglongjmp* (3), *setjmp* (3), *longjmp* (3), *select* (2), *gettimeofday* (2).

## AUTHOR

```
Ralf S. Engelschall
rse@engelschall.com
www.engelschall.com
```