A Runtime Library for Lightweight Process-Scope Threads

P.E. Hadjidoukas V.V. Dimakopoulos
Parallel Processing Group
Department of Computer Science
University of Ioannina, Ioannina 45110, Greece
{phadjido,dimako}@cs.uoi.gr
http://paragroup.cs.uoi.gr

Technical Report PPG-CS-UOI-280907

Abstract

This work presents an open-source software package that implements a two-level thread model. It consists of two thread libraries, UthLib and PSthreads. UthLib (Underlying Threads Library) is a very portable thread package core that provides the primary primitives for managing non-preemptive user-level threads (creation and context-switch) on UNIX and Windows platforms. The PSthreads (process scope threads) library takes advantage of UthLib and implements a hybrid thread model. The software package is freely available from the webpage: http://www.cs.uoi.gr/~ompi. It is distributed under the terms of the GNU General Public License (GPL) version 2 or later.

1. Introduction

It is common knowledge that the performance of kernel threads, although an order of magnitude better than that of traditional processes, has been typically an order of magnitude worse than the best-case performance of user-level threads [2]. In an application that utilizes user threads, the threads of the application are managed by the application itself. In this way, functionality and scheduling policy can be chosen according to the application. These user threads are much more efficient than kernel threads in carrying out operations such as context switching, since no kernel intervention is necessary to manipulate threads.

This work presents an open-source software package that implements a two-level thread model. It consists of two thread libraries, UthLib and PSthreads. UthLib is a very portable thread package core that provides the primary primitives for managing non-preemptive user-level threads (creation and context-switch) on UNIX and Windows platforms. The purpose of UthLib is to facilitate the implementation of two-level thread models, where virtual processors are system scope POSIX Threads. The PSthreads library takes advantage of UthLib and implements a two-level user-level thread model. PSthreads provide efficient support of nested parallelism to the OMPi OpenMP compiler. The software package is freely available from the webpage: http://www.cs.uoi.gr/~ompi. It is distributed under the terms of the GNU General Public License (GPL) version 2 or later.

2. Design

The configuration and installation process of both PSthreads and UthLib is performed with a common *configure* script that includes the whole software package. There are configuration options related either to Uthlib or to PSthreads. Moreover, some configuration options apply to both software libraries and allow the user to determine the maximum number of virtual processors, the synchronization mechanism used, the thread recycling approach and the cache line size.

Figure 1 presents the general design of the software package. At the lower level, UthLib implements the necessary primitives for thread management (creation and context-switch). As we will show, these primitives may constitute the machine dependent (md) part of the library. They are based on the management routines of jmpbuf or ucontext_t structures, or exclusively on POSIX threads. UthLib utilizes a queue-based recycling mechanism for the underlying threads. The necessary routines for multiprocessor synchronization and queue management are implemented in two separate header files (locks.h and queues.h). These routines and the exported application programming interface of UthLib are both utilized by the PSthreads runtime library, which implements the two-level thread model. It also exports a Pthreads-like application programming interface (API), which can be used by multithreaded applications. In the case of the OMPi OpenMP C compiler, its OpenMP runtime library can take advantage of PSthreads, providing thus efficient support of nested parallelism to applications that have been parallelized according to the OpenMP programming model.

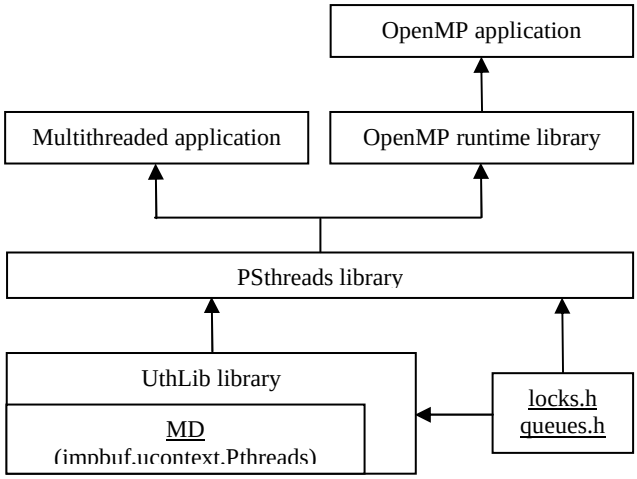

Figure 1. General design

3. UthLib

UthLib (Underlying Threads Library) is a very portable thread package core that provides the primary primitives (creation and context-switch) for managing portable non-preemptive user-level threads on UNIX and Windows platforms. UthLib is not a standalone thread package; it does not provide its own synchronization primitives and requires/assumes the presence of a POSIX Threads library [5]. Its purpose is to facilitate the implementation of two-level thread libraries, where virtual processors are system scope POSIX Threads. It also exports a well-defined API that can be easily implemented using custom (platform specific) thread libraries. UthLib has been partially implemented using a minimal and modified version of the State Threads Library [7]. Therefore, it is distributed under the terms of the Mozilla Public License (MPL) version 1.1 or the GNU General Public License (GPL) version 2 or later.

State Threads is an application library that provides a foundation for writing fast and highly scalable Internet Applications on UNIX-like platforms. It combines the simplicity of the multithreaded programming paradigm, in which one thread supports each simultaneous connection, with the performance and scalability of event-driven state machine architecture. It is a very portable user-level threads package based on the setjmp-longjmp primitives, but supports a multi-process rather than a multithreaded environment. It can be combined with traditional threading or multiple process parallelism to take advantage of multiple processors. It has been derived from the Netscape's Portable RunTime Library [5], which however supports multithreading.

3.1 Implementation Issues

In this section, we discuss the most important implementation issues of UthLib.

- *Self-identification*: UthLib targets two-level thread models, where non-preemptive user-level threads are executed on top of kernel-level threads that act as virtual processors, ranked from 0 to MAX_VPS-1). For this reason, it maintains per-virtual processor global data. Several operations require a self-identification method of the current virtual processor. A portable way to perform this is to use the self-identification mechanism provided by the POSIX Threads API: pthread_self. When a virtual processor is initialized, it stores its pthread_t identifier into a global array and can find its rank by locating the position of its identifier in this array.
- *Stack size*: In the current implementation, all threads have stacks of equal size, set with the uth_init call. This design decision is not mandatory and has been adopted because it simplifies the recycling of threads.
- *Synchronization*: UthLib optionally reuses finished thread descriptors. The recycling can be performed globally or on a per-processor basis, by utilizing appropriate thread queues.
- *Internal data structures*: The data structures that describe a user-level thread and its stack (thread and stack descriptors) are similar with those defined in the State Thread library. However, we have

encapsulated the stack descriptor in the thread descriptor and thus a single memory allocation operation is required for creating a user-level thread and its stack.

- *Thread context*: The only platform-dependant part of the library resides in the thread context management (initialization and context-switch). The state information of a user-level thread is manipulated using an appropriate structure that is stored in its descriptor. We support two methods:
    - *SJLJ (setjmp/longjmp)*: According to this method, which is utilized by the State Threads library, the thread descriptor includes a jmpbuf data structure, defined in the setjmp.h header file. Two ingredients of the jmpbuf data structure (the program counter and the stack pointer) have to be manually set in the thread creation routine. The data structure differs from platform to platform. Usually the program counter is a structure member with PC in the name and the stack pointer is a structure member with SP in the name. One can also look in the Netscape's NSPR library source, which already has this code for many UNIX-like platforms (mozilla/nsprpub/pr/include/md/*.h files). Furthermore, the State Threads library provides an assembly-based built-in implementation of the setjmp-longjmp operations on some platforms.
    - *MCSC (makecontext/swapcontext)*: Most modern UNIX environments provide one more option for user-level context-switching between multiple threads of control within a process: the ucontext data structure defined in ucontext.h and the four functions: getcontext, setcontext, makecontext and swapcontext. For more information on the usage of these functions, you can look at [6]. Although the Microsoft C Runtime Library does not provide these functions, we have implemented the UNIX ucontext operations on Windows platforms by using the Win32 API GetThreadContext and SetThreadContext functions [3].

3.2 UthLib Configuration Options

Some configuration options are related to both UthLib and PSthreads libraries. Since we discuss UthLib first, we report them here:

- *Maximum number of virtual processors*: This option determines the maximum number of supported virtual processors (default = 16).
- *Default stack size*: This option determines the default size of the user-level stacks. As already mentioned, all user-level threads have stack of equal size (default = 4MB).
- *Thread recycling*: This option activates a recycling mechanism for the threads. Thread creation tries to reuse a finished thread that has been recycled before. The use can specify whether the recycling mechanism will be performed on a per-virtual processor or global basis.

The following configuration options are exclusively related to UthLib:

- *Context switch method*: This option determines the most platform-dependant part of the runtime library, i.e. thread initialization and context-switch. The available methods are SJLJ and MCSC, based on the setjmp-longjmp and ucontext primitives respectively. Engelschall proposes in [2] a portable trick for user-level thread creation and also references these two methods.
- *Stack alignment*: The user-level stack can be aligned on either 64-byte or page boundary.
- *User-level thread emulation*: UthLib provides the primary primitives for non-preemptive user-level threads. Optionally, UthLib can emulate these user-level threads with POSIX threads. This emulation is completely transparent to the user. Context-switch between threads is implemented using condition variables.

3.3 Programming Interface

The API of UthLib includes the following definitions and calls (exported to the user through uth.h):

- **uth_t**: Type of the underlying thread
- **void uth_init (int stacksize)**: Initializes the library and sets the stack-size of the user-level threads. It is called only once.
- **int uth_vp_init (int vp, void *arg)**: Initializes the current virtual processor. Each kernel thread must call this once, associating itself with an id and an argument. If uth_init has not been called yet, it is called setting the stack size to its default size. Returns 0 on success, -1 on error.

- **int uth_get_vpid(void)**: Returns the rank (id) of the current virtual processor (0...UTH_MAX_CPUS-1). On error, terminates the application.
- **uth_t uth_create (void (*fn)(void *), void *arg)**: Creates a user-level thread that will execute fn function, which receives a single argument (arg). If the recycling mechanism is active, the routine tries to reuse a finished thread. Upon successful completion, a (new) thread descriptor is returned. Otherwise, it returns NULL.
- **void uth_reinit (uth_t thread, void (*func)(void *), void *arg)**: Re-initializes an underlying thread.
- **void uth_delete(uth_t thread)**: Deletes (or recycles) and underlying thread.
- **void uth_switchto(uth_t old, uth_t new)**: Performs thread context-switching on the current virtual processor, saving the context of thread old (if this is not NULL) and restoring the context new.
- **uth_t *uth_self(void)**: Returns a reference to the current thread.
- **void *uth_getarg(uth_t thread)**: Returns the function argument of a thread.
- **void *uth_setarg(uth_t thread, void *arg)**: Sets the function argument of a thread.
- **double uth_gettime(void)**: Returns current time.
- **void uth_vp_sleep(int ms)**: Suspends the kernel thread for ms milliseconds.

For performance reasons, the following routines are also provided:

- **void uth_switchto_ex(int vp, uth_t old, uth_t new)**: Performs thread context-switching on the virtual processor with rank vp. It can be used in cases where the user's runtime library provides this information on its own.
- **uth_t *uth_self_ex(int vp)**: Returns a reference to the thread that is currently executed on the virtual processor with rank vp.
- **void *uth_getarg_ex(int vp)**: Returns the function argument of the thread that is currently running on virtual processor with rank vp.

3.4 Validation

The successful execution of the program in Figure 2 validates the feasibility of a two-level thread model implementation on top of POSIX Threads. The main kernel thread (vp 0 - virtualprocessorA) creates NumThreads user-level threads. Next, it passes the control of execution to the first user-level thread, the first to the second and so on (First Round), until the control of execution returns to the main thread. Finally, it creates a kernel thread (vp 1 - virtualprocessorB) that repeats the previous pass of execution on the same threads (Second Round).

The output of the program should be similar to the following:

```
[0x312c48] master thread A starts    [pthread_t = 0x2f44a0]
[0x45fffc] round one: arg (0 / 0)    [local = 1 global = 1]
[0x480034] round one: arg (1 / 1)    [local = 1 global = 2]
[0x312c48] master thread A continues [global = 2]
[0x312d28] master thread B starts    [pthread_t = 0x2f4528]
[0x45fffc] round two: arg (0 / 0)    [local = 2 global = 3]
[0x480034] round two: arg (1 / 1)    [local = 2 global = 4]
[0x312d28] master thread B exits     [global = 4]
[0x312c48] master thread A exits
```

UthLib, and particularly the test application, has been tested successfully on the hardware/software configurations presented in Table 1.

Operating System	Architecture	Compiler	Context-switch method
GYGWIN (WIN32)	X86	GCC	SJLJ, MCSC
MINGW32	X86	GCC	SJLJ, MCSC
LINUX 2.6	X86, X86_64	GCC, ICC	SJLJ, MCSC
SOLARIS 9	SPARCv9	GCC, CC	SJLJ, MCSC
IRIX 6.5	MIPS	GCC, CC	SJLJ, MCSC
FREEBSD 6.2	X86	GCC	SJLJ, MCSC

Table 1. Tested Platforms

```
#include <pthread.h>
#include <uth.h>
#include <stdio.h>

#define NumThreads 2

uth_t worker[NumThreads], virtualprocessorA, virtualprocessorB;
int gvar = 0;

void workerFunc(void* arg) {
   long id = (long)arg;
   int lvar = 0;

   gvar++; lvar++;

   printf("[0x%lx] round one: arg (%ld / %ld)\t [local = %d global = %d]\n",
        (unsigned long) uth_self(), id, (long) uth_getarg(uth_self()), lvar, gvar);

   if (id == NumThreads-1)    uth_switchto_ex(0, worker[id], virtualprocessorA);
   else                       uth_switchto_ex(0, worker[id], worker[id+1]);

   gvar++; lvar++;

   printf("[0x%lx] round two: arg (%ld / %ld)\t [local = %d global = %d]\n",
        (unsigned long) uth_self(), id, (long) uth_getarg(uth_self()), lvar, gvar);

   if (id == NumThreads-1)    uth_switchto_ex(1, NULL, virtualprocessorB);
   else                       uth_switchto_ex(1, NULL, worker[id+1]);
}

void *kernelthreadfunc(void *arg) {
   uth_vp_init(1, NULL);
   virtualprocessorB = uth_self();

   printf("[0x%lx] master thread B starts\t [pthread_t = 0x%lx]\n",
        (unsigned long) uth_self(), pthread_self());

   uth_switchto_ex(1, virtualprocessorB, worker[0]);
   printf("[0x%lx] master thread B exits\t [global = %d]\n",
        (unsigned long) uth_self(), gvar);
   return 0;
}

int main(void) {
   long i;
   pthread_t pth;
   pthread_attr_t attr;
   long status;

   pthread_attr_init(&attr);
   pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);
   pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);

   uth_init(0);
   uth_vp_init(0, NULL);
   for(i=0; i<NumThreads; ++i) worker[i] = uth_create(workerFunc,(void *)i);

   printf("[0x%lx] master thread A starts\t [pthread_t = 0x%lx]\n",
               (unsigned long) uth_self(), pthread_self());
   virtualprocessorA = uth_self();
   uth_switchto_ex(0, virtualprocessorA, worker[0]);
   printf("[0x%lx] master thread A continues\t [global = %d]\n",
        (unsigned long) uth_self(), gvar);

   pthread_create(&pth, &attr, kernelthreadfunc, NULL);
   pthread_join(pth, (void *)&status);

   for(i=0; i<NumThreads; ++i)       uth_delete(worker[i]);

   printf("[0x%lx] master thread A exits\n", (unsigned long) uth_self());
   return (gvar == (2*NumThreads));
}
```

Figure 2. Check application

3.5 Evaluation

In this section, we measure the overhead for user-level context-switch in UthLib. The experiments were performed on the following machines:

- LAPTOP: Pentium M 1600 MHz, 512MB RAM
- ATLANTIS: 4xPentium III 701MHz, 1520MB RAM
- IRO: Pentium 4 2.40GHZ, 512MB RAM
- GRID: 2xAMD Opteron 248 2210MHz, 4 GB RAM
- ARTHUR: 4xSparcv9 480MHz, 3072 MB RAM
- ZEUS: 2xSparcv9 1200MHz, 8192 MB RAM
- SOCRATES:4x MIPS R12000 360MHz, 1536 MB RAM

We measure the pure context-switch overhead using a ping-pong benchmark between two threads. The results are depicted in Table 2. We observe that the SJLJ method provides faster lightweight context-switch, since it saves fewer registers than the MCSC method. The latter overhead, however, is balanced by the portability of the MCSC method.

PLATFORM	OPERATING SYSTEM	ABI	COMPILER	SJLJ	MCSC	PTH
LAPTOP	CYGWIN	32	GCC 3.4.4	0.07	1.74	6.17
LAPTOP	MINGW32[1]	32	GCC 3.4.2	0.07	1.74	3.67
LAPTOP	LINUX 2.4.20	32	GCC 4.2.0	0.07	0.46	4.27
ATLANTIS	LINUX 2.6.18	32	GCC 4.2.0	0.18	0.85	12.72
IRO	FREEBSD 6.2	32	GCC 3.4.6	0.15	1.65	23.03
GRID	LINUX 2.6.9	64	GCC 3.4.5	0.04	0.27	4.63
ARTHUR	SOLARIS 2.8	64	GCC 3.0.2	0.78	17.51	26.09
ZEUS	SOLARIS 2.9	64	GCC 3.4.2	0.42	8.95	16.04
		32		0.44	9.23	14.83
SOCRATES	IRIX 6.5	32	GCC 3.3[2]	0.46	16.02	5.02
		64		0.43	21.03	5.38
		32	CC 7.30	0.41	16.65	4.87
		64		0.42	16.35	5.43

Table 2. Context-Switch Overhead (microseconds)

3.6 Possible Optimizations

- The context-switch mechanism can be based on assembly instructions. For instance, the user can implement platform-specific versions of setjmp-longjmp, which might result in lower overhead. In fact, this is the case the following configurations: CYGWIN, MINGW, LINUX/IA64, LINUX/I386, LINUX/X86_64 and LINUX/AMD64. On both CYGWIN and MINGW, we use the same assembly based version of setjmp-longjmp to avoid some complications of the system provided implementation. On LINUX platforms, the assembly code is provided by the State Threads library.
- In order to achieve maximum portability, UthLib has been built on top of the POSIX Threads API. However, it can be also built on top of the native kernel threads that operating systems provide. Furthermore, platform-specific mechanisms for mutual exclusion can replace the POSIX Threads based ones in use, while non-blocking/lock-free algorithms can be utilized for the reuse queues.
- Another possible POSIX Threads compliant self-identification mechanism is the use of thread-specific data (pthread_{set,get}_specific). Moreover, a stack-based implementation for thread self identification can be used. This can be performed by allocating stacks on appropriate page boundaries (memalign).
- User-level threads are executed through a driver routine (_uth_main). This routine identifies the currently executed thread and calls the user specified function for this thread. This self-identification can be avoided by passing an argument (a pointer to the thread descriptor) to the driver routine.

[1] The Pthreads-win32 and MinGW32 MSys development kit were used. More information can be found at http://sources.redhat.com/pthreads-win32/.

[2] Software package was configured with CFLAGS=-D__SGI_LIBC_NAMESPACE_QUALIFIER=

4. Process Scope Threads (PSthreads) Library

4.1 Introduction

Figure 3 presents a general overview of the design and functionality of user-level thread libraries. Most modern operating systems provide support for creating kernel level threads. In this case, the kernel distinguishes processes, which include all the necessary resources for process execution (e.g. virtual memory, file descriptors), from threads, which represent the execution state of processes. A running program consists of a process and one or more kernel level threads that execute in the context of this process. Operating systems offer an additional system call for kernel thread creation. Kernel threads are entities the operating system is aware of, and the OS scheduler assigns these threads to physical processors. An application running on a computer system with *p* processors does not usually create more than p kernel threads, since this is the maximum number of threads that can run in parallel.

As both creation and management of kernel threads is performed through system calls, the corresponding runtime overhead is high due to the transition of processor execution from user to kernel mode. This overhead prevents many applications from exploiting their fine-grained parallelism, since the thread creation overhead overwhelms the benefits from the additional created parallelism. User-level threads were introduced to overcome this problem. They consist of a descriptor, where hardware register contents are stored, a private memory (stack), and some additional fields that are used by the thread library for their management.

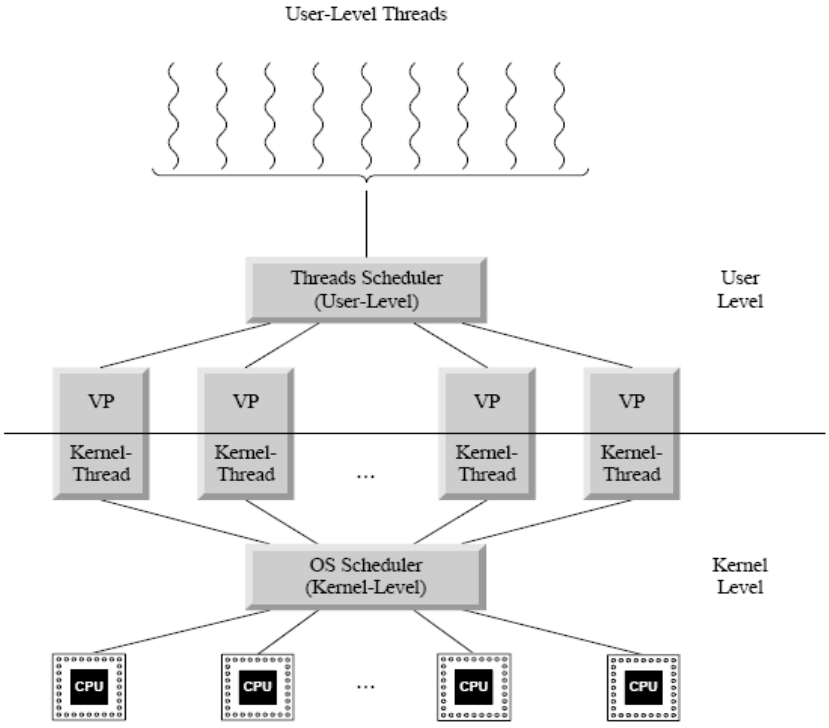

Figure 3. Two-level thread model

User-level threads are unknown to the operating system. Their creation, management, scheduling and destruction are performed exclusively by the user-level thread library, in the context of a kernel thread. For every task that can be executed in parallel, a user-level thread is created and executed by a kernel thread. This means that kernel threads act at virtual processors, which execute user-level threads, similarly to physical processors which execute kernel threads. Therefore, exploitation of multiprocessor hardware cannot be achieved by creating multiple user-level threads and having a single kernel thread. The OS scheduler assigns a single processor to that kernel thread and consequently, only a single user-level thread is executed at a given time.

The hybrid thread model manages to exploit multiple processors without losing the low overheads of user-level threads. A parallel program creates a specific number of kernel threads, equal to the degree of parallelism that needs to achieve. These kernel threads are maintained throughout the application execution. Meanwhile, the program creates as many user-level threads as required for instantiating the existing parallelism.

4.2 Implementation Issues

The PSthreads library implements a two-level thread model, as previously discussed. There are local ready queues, one per virtual processor, and a global one. Each virtual processor runs a dispatch loop, selecting the next to run user-level thread from this set of ready queues, where threads are submitted for execution. An idle virtual processor extracts threads from the front of its local ready queue but steals from the end of remote queues. The global queue is used for coarse grain tasks. The queue architecture allows the runtime library to represent the layout of physical processors.

PSthreads are non-preemptive, therefore a thread cannot start its execution before the previous one terminates or voluntarily releases its processor. The user-level thread scheduler is activated at well defined entry points of the runtime library and not periodically using an appropriate timer.

Despite the user-level multithreading, the PSthreads library is fully portable because its implementation is based on the POSIX standard. The primary user-level thread operations are provided by UthLib. An underlying thread is actually the stack that a psthread uses during its execution. An important feature of PSthreads is the utilization of a lazy stack allocation policy. According to this policy, the stack (uth_t) of a psthread is allocated just before its execution. This results in minimal memory consumption and thread migrations between processors.

4.3 Application Programming Interface

The PSthreads library exports an application programming interface (API) that is similar to that of POSIX threads. To take advantage of this interface, the user must include psthread.h.

4.3.1 Initialization

- **int psthread_init(int nvps, unsigned long stacksize)**: Initializes the runtime library. It must be called before any other library routine. It initializes the internal data structures of the library and creates the user specified number of virtual processors (nvps parameter). It also sets the stack size of user-level threads. On success it returns 0, on failure EAGAIN.

4.3.2 Thread Management

- **psthread_t**: Type of a process scope thread.
- **psthread_attr_t**: Data structure where attributes of PSthreads are set. The thread attribute allows the user to define whether a thread will be detached or joinable (detachstate field) and also to explicitly set the parent for this thread (parent field). The default values for the two thread attributes are PSTHREAD_JOINABLE and NULL.
- **int psthread_create(psthread_t *psthr, psthread_attr_t *attr, void (func)(void *), void *arg)**: Creates a new user-level thread. When scheduled, the thread executes the func routine, which takes a single argument (arg) and does not return anything. A thread terminates its execution when its routine returns or it calls psthread_exit(). The created thread is not automatically dispatched for execution. The user has to explicitly insert it in a ready queue. On success, psthread_create stores the descriptor of the new thread in the address pointed by the first argument and returns 0. Otherwise, it returns EAGAIN.
- **int psthread_enqueue(psthread_t psthr, int queue_id)**: Inserts a user-level thread in the specified ready queue. If queue_id equals -1, the thread is inserted in the global ready queue. Returns 0 on success, EINVAL on failure (e.g. invalid queue_id).
- **int psthread_enqueue_head(psthread_t thread, int queue)**: As above, but it puts the thread at the front of a given queue.
- **int psthread_enqueue_tail(psthread_t thread, int queue)**: Alias of psthread_enqueue.
- **void psthread_exit(void)**: Terminates the execution of the current thread. The memory of a finished thread is freed (recycled). If it is joinable, the thread notifies its parent (a counter is decreased).
- **int psthread_waitall(void)**: The current thread suspends its execution waiting all its joinable children threads to finish. If there are not pending threads, the current thread continues its execution. The function returns 1 if the thread is suspended and 0 if there are not children threads running.

- **void psthread_yield(void)**: The current thread releases it virtual processor, allowing the execution of another user-level thread. The suspended thread will be restarted when selected by the thread scheduler.
- **psthread_t psthread_self(void)**: Returns the descriptor of a current thread.

4.3.3 Execution environment

- **int psthread_nvps(void)**: Returns the number of virtual processors.
- **int psthread_current_vp(void)**: Returns the number of the virtual processor where the current user-level thread executes on.

4.3.4 Synchronization

The synchronization primitives of PSthreads are based on the POSIX threads API. They are mapped to POSIX mutexes or spinlocks, according to the configuration process of the library during its installation. These primitives are not user-level thread aware.

- **psthread_lock_t**: Type of synchronization variable.
- **void psthread_lock_init(psthread_lock_t *lock_var)**: Initializes a lock.
- **void psthread_lock_acquire(psthread_lock_t *lock_var)**: Acquires a lock.
- **void psthread_lock_try_acquire(psthread_lock_t *lock_var)**: Attempts to acquire a lock.
- **void psthread_lock_release(psthread_lock_t *lock_var)**: Releases a lock.
- **void psthread_lock_destroy (psthread_lock_t *lock_var)**: Destroys a lock.

4.3.5 Condition Variables

The implementation of condition variables is based on the queues.

- **psthread_cond_t**: Type of condition variable.
- **void psthread_cond_init(psthread_cond_t *cond)**: Initializes a condition variable.
- **void psthread_cond_wait (psthread_cond_t *cond, psthread_lock_t *lock)**: The current thread waits on a condition variable. The thread has to acquire the lock before calling psthread_cond_wait. Before its suspension, the thread releases the lock. When the thread is resumed, the routine returns and the thread implicitly acquires the lock.
- **void psthread_cond_signal (psthread_cond_t *cond)**: Signals a condition variable. A single user-level thread that waits on that condition variable will become ready for execution.
- **void psthread_cond_broadcast (psthread_cond_t *cond)**: Broadcasts a condition variable. All threads blocked on that condition variable will be awaked.
- **void psthread_cond_destroy (psthread_cond_t *cond)**: Destroys a condition variable.

4.3.6 Barriers

The barriers of PSthreads are based on D. Butenhof's [1] proposed implementation that makes use of condition variables.

- **psthread_barrier_t**: Type of barrier between user-level threads.
- **int psthread_barrier_init (psthread_barrier_t *barrier, int count)**: Initializes a barrier.
- **int psthread_barrier_wait (psthread_barrier_t *barrier)**: The current thread blocks until the required number of threads has reached the barrier.
- **int psthread_barrier_destroy (psthread_barrier_t *barrier)**: Destroys a barrier.

4.3.7 Thread Local Storage (TLS)

The PSthreads library supports a limited form of thread local storage. It allows up to MAX_TLS_KEYS (= 64) keys of storage and there is not support for key destruction.

- **psthread_key_t**: Type of key for thread specific data.
- **int psthread_key_create(psthread_key_t *key, void *dummy)**: Allocates a key for TLS data.
- **int psthread_setspecific(psthread_key_t key, const void *value)**: Stores a value at position *key* of thread local storage.

- **int psthread_setspecific2(psthread_t thr, psthread_key_t key, const void *value)**: Optimized version of the previous routine.
- **void *psthread_getspecific(psthread_key_t key)**: Retrieves the value stored at position *key* of thread local storage.
- **void psthread_key_destroy(psthread_key_t key)**: Releases a specified position in the thread local storage array.

4.3.8 Auxiliary Routines

- **double psthread_gettime(void)**: Returns the current time in seconds.
- **void psthread_vp_sleep (int msec)**: Suspends the current virtual processor (kernel thread) for msec milliseconds. This call is not user-level thread aware and thus the running user-level thread also remains suspended.

4.4 Psthread Specific Configuration Options

- *Ready queues architecture*: The user can determine whether a single global queue will be only available or local ready queues will be also present.
- *Thread stealing mechanism*: If a single global queue is used, all virtual processors try to dispatch work from there. When local ready queues have been activated, an idle virtual processor will try to find work in remote queues only if the thread stealing mechanism of the library is enabled.
- *Runtime behaviour of idle virtual processors*: When there is enough work available, a virtual processor may not find any thread to execute and thus becomes idle. The user can determine whether an idle virtual processor executes the scheduling loop continuously or yields the physical processor periodically.

5. Configure

Common (general) options

- [--with-maxvps=num]: sets the maximum supported number of virtual processors (default: 16).
- [--with-sync=method]: sets the Pthreads synchronization mechanism that will be used. Possible options: mutex (default), mutex_try, spin, spin_try.
- [--with-recycling=method]: sets the recycling mechanism for threads in both libraries. Possible options: local, global, none.
- [--with-cachelinesize=value]: the user can give the cache line size, which is used in the alignment of internal data structures. The configuration script tries to find out the exact value. If this not possible, the default value is set equal to 128.

Uthlib options

- [--with-csm=method]: sets the context switch method (sjlj, mcsc).
- [--enable-asmsjlj]: enables the built-in implementation of the sjlj context-switch method. This option is ignored if there is not such implementation on the working platform.
- [--enable-uth2pth]: enables the emulation of user-level threads on top of POSIX threads. It is disabled by default. If enabled, the previous option is meaningless.
- [--enable-stackalign64]: if set, the stack is aligned on 64-byte boundaries. By default, stacks are page-aligned.

PSthreads options

- [--enable-singlequeue]: a single ready queue will be used for thread dispatching. It is disabled by default, which means that local queues are in use.
- [--enable-yieldvp]: an idle virtual processor yields the underlying physical processor by calling *sched_yield()* (default:disabled).
- [--enable-stealing]: enables thread stealing for PSthreads (default:disabled).

References

[1] Butenhof, D. R.: Programming with POSIX Threads. Professional Computing Series, Addison-Wesley, ISBN 0-201-63392-2, May 1997.

[2] Engelschall, R.: Portable Multithreading: the Signal Stack Trick for User-Space Thread Creation, In Proc. of the USENIX Annual Technical Conference, 2000.

[3] Hadjidoukas, P. E: Implementing Unix ucontext_t operations on Windows Platforms. The Code Project. Threads, Processes and IPC. May 2003. Available at http://www.codeproject.com/threads/context.asp.

[4] Keppel, D.: Tools and Techniques for Building Fast Portable Thread Packages, University of Washington at Seattle, Technical Report UW-CSE-93-05-06, June 1993.

[5] Netscape Portable Runtime Library. Available at http://www.mozilla.org/docs/refList/refNSPR.

[6] The Open Group Base Specifications Issue 6, IEEE Std. 1003.1, 2003 Edition, http://www.opengroup.org/onlinepubs/007904975/.

[7] State Threads Library for Internet Applications. IBM Open Source Projects, http://oss.sgi.com/projects/state-threads/.