UthLib: A Portable Non-Preemptive User-level Threads Package

P. E. Hadjidoukas
IRISA-INRIA, Campus de Beaulieu, 35042 Rennes cedex, France
Tel: +33 (0) 2 99 84 73 90, Fax: +33 (0) 2 99 84 71 71
`Panagiotis.Hadjidoukas@irisa.fr`

Version 1.0

21st March 2004

Abstract

`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. `UthLib` is not a standalone thread package, it does not provide its own synchronization primitives and requires the presence of a POSIX Threads library. Its purpose is to facilitate the implementation of two-level thread models (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 implemented using a minimal and modified version of the State Threads Library. Therefore, it is freely available from `http://www.hpclab.ceid.upatras.gr/ peh/uthlib.html`, under the terms of the Mozilla Public License (MPL) version 1.1 or 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 [6]. 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.

In this work, we present `UthLib` (Underlying Threads Library), a very portable thread package core that provides the primary primitives for managing portable non-preemptive user-level threads (creation and context-switch) 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 [1]. Its purpose is to facilitate the implementation of two-level thread models (libraries), where virtual processors are system scope POSIX Threads. It also exports a well-defined API that can be easily implemented using custom (platform dependant) thread libraries. UthLib has been implemented using a minimal and modified version of the State Threads Library [9]. 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 [7], which however supports multithreading.

We have successfully used UthLib in order to provide a portable and modular OpenMP implementation, through a two-level thread model, on both shared-memory multiprocessors [3] and clusters of SMPs [5].

2 Compiling UthLib

In order to compile UthLib, the first step is to define (in Makefile) the appropriate operating system and the compiler. Currently, UthLib is built only as a static library. Moreover, UthLib has the following compile-time options (defined in uth_opt.h):

- **UTH_MAX_CPUS**: This option determines the maximum number of supported virtual processors (default = 8).
- **UTH_DEFAULT_STACK_SIZE**: This option determines the default size of the user-level stacks. UthLib assumes that all user-level threads have stack of equal size (default = 64K). This simplifies and optimizes the reuse mechanism for stacks.
- **REUSE_THREADS**: This option activates a reuse mechanism for the threads. Thread creation tries to reuse a finished thread that has been recycled before.
- **LOCAL_REUSE_QUEUES**: If the recycling mechanism has been activated, it is performed on a per-virtual processor rather than on a global basis.

- **CTXSW_METHOD**: This option determines the most platform-dependant part of the runtime library, i.e. thread initialization and context-switch. The available methods, which are further discussed later, are **CTXSW_SJLJ** and **CTXSW_MCSC**, based on the `setjmp`/ `longjmp` and `ucontext_t` primitives respectively. Engelschall proposes in [2] a portable trick for user-space thread creation and also refers these two methods.

2.1 Programming Interface

The API of `UthLib` provides 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)**: Initializes the current virtual processor. If uth_init has not been called yet, it is called setting the stack size equal to 128K. 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)**: Reinitializes 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.
- **uth_t *uth_self2(int vp)**: Returns a reference to the thread that is currently executed on virtual processor with rank `vp`.

- **void uth_switchto2(int vp, uth_t old, uth_t new)**: Performs thread context-switching on the virtual processor with rank `vp`. It can be used for cases where the user's runtime library can provide this information on its own.

3 Implementation

In this section, we discuss the most significant implementation parts of `UthLib`.

Self-identification: `UthLib` targets two-level thread models, where non-preemptive user-level threads are executed on top of kernel-level threads (virtual processors, ranked from `0` to `UTH_MAX_CPUS-1`). For this reason, it maintains per-virtual processor global data. Many 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 in a global array. It 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` can optionally reuse a finished thread descriptor. The recycling can be performed globally or on a per-processor basis, by utilizing appropriate thread queues. The queues are protected with POSIX mutexes.

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 for creating a user-level thread and its stack is required.

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 Thread library, the thread descriptor includes a `jmp_buf` data structure, defined in the setjmp.h header file. Two ingredients of the `jmp_buf` 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, we have added support for the QNX 6.0 operating system and integrated the code for Windows platforms as provided in an older version of the State Threads Library for this operating system.

- **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_t` 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 [8]. Although the Microsoft C Runtime Library does not provide these functions, we have implemented the Unix `ucontext_t` operations on Windows platforms by using the Win32 API `GetThreadContext` and `SetThreadContext` functions.

4 Test Application

The successful execution of the following program (Figure 1 means the ability to implement a two-level thread model 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 getpid = 816]
[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 getpid = 816]
[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.

5 Evaluation

In this section, we measure the overhead for the primary operations of `UthLib`: context-switch and creation/re-initialization of user-level threads. In all experiments, we use the default stack size (64KB) for the threads. The experiments were performed on the following machines:

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

#define NumThreads 2

uth_t worker[NumThreads];
uth_t virtualprocessorA;
uth_t virtualprocessorB;

int global_var = 0;

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

   global_var++;
   local_var++;

   printf("[0x%lx] round one: arg (%d / %d)\t [local = %d global = %d]\n",
       uth_self(), id, (int) uth_getarg(uth_self()), local_var, global_var);

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

   global_var++;
   local_var++;

   printf("[0x%lx] round two: arg (%d / %d)\t [local = %d global = %d]\n",
       uth_self(), id, (int) uth_getarg(uth_self()), local_var, global_var);

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

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

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

   uth_switchto(virtualprocessorB, worker[0]);
   printf("[0x%lx] master thread B exits\t [global = %d]\n",
       uth_self(), global_var);

   return 0;
}

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

   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);
   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 getpid = %ld]\n",
       uth_self(), pthread_self(), getpid());
   virtualprocessorA = uth_self();
   uth_switchto(virtualprocessorA, worker[0]);
   printf("[0x%lx] master thread A continues\t [global = %d]\n",
       uth_self(), global_var);
   pthread_create(&pth, NULL, kernelthreadfunc, NULL);
   pthread_join(pth, NULL);

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

   printf("[0x%lx] master thread A exits\n", uth_self());
   return 0;
}
```

Figure 1: Test application

OPERATING SYSTEM	ARCH	COMPILER	METHOD
WINDOWS 2000	X86	MSVC, GCC, ICC	SJLJ, MCSC
LINUX 2.4.18	X86	GCC, ICC	SJLJ, MCSC
SOLARIS 8	X86, SPARC	GCC, UCBCC	SJLJ, MCSC
IRIX 6.5 - n32	MIPS	GCC, CC	SJLJ, MCSC
IRIX 6.5 - n32	MIPS	CC	SJLJ, MCSC
AIX 5.3 - 32	POWER	GCC, XLC	SJLJ, MCSC
AIX 5.3 - 64	POWER	XLC	MCSC
QNX 6.0	X86	GCC	SJLJ
FREEBSD 5.0	X86	GCC	SJLJ, MCSC

Table 1: Tested platforms

MACHINE	OPERATING SYSTEM	COMPILERS	SJLJ	MCSC
ALKAIOS	WINDOWS 2000	MSVC 6.0	201	3012
ALKAIOS	LINUX 2.4.18	GCC 3.2	129	644
GALOIS	SOLARIS 8	UCBCC 5.0	336	6823
KARNAK	IRIX 6.5 - n32	CC 7.30	101	4846
KARNAK	IRIX 6.5 - n32	CC 7.30	118	5030
KADESH	AIX 5.3 - 32	XLC	163	3196
KADESH	AIX 5.3 - 64	XLC	N/A	4343

Table 2: Context-Switch Overhead (processor cycles)

ALKAIOS	`Pentium III 866MHz, 256MB RAM`
GALOIS	`UltraSparc 296MHz, 512 MB RAM`
KARNAK	`MIPS R10000 250MHz, 512 MB RAM`
KADESH	`RS-6000 POWER3 375MHz, 64 GB RAM`

5.1 Context-switch overhead

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

5.2 Creation and Re-Initialization Overhead

In this benchmark, we measure the average time required for creation, reuse and re-initialization of a user-level thread. The benchmark measures the required time for the creation (`uth_create`) of 100 threads (allocation of memory and initialization). These threads are recycled (`uth_delete`) explicitly and the creation of the same number of (recycled) threads is measured again. Finally, we measure the time to re-initialize all these threads (`uth_reinit`). Tables 3 and 4

MACHINE	OPERATING SYSTEM	COMPILERS	SJLJ	MCSC
ALKAIOS	WINDOWS 2000	MSVC 6.0	9.1	15.1
ALKAIOS	LINUX 2.4.18	GCC 3.2	12.7	13.2
GALOIS	SOLARIS 8	UCBCC 5.0	165.5	183.7
KARNAK	IRIX 6.5 - n32	CC 7.30	148.6	195.8
KARNAK	IRIX 6.5 - n32	CC 7.30	171.7	200.8
KADESH	AIX 5.3 - 32	XLC	30.7	33.1
KADESH	AIX 5.3 - 64	XLC	N/A	36.9

Table 3: Creation Overhead (usec)

MACHINE	OPERATING SYSTEM	COMPILERS	SJLJ	MCSC
ALKAIOS	WINDOWS 2000	MSVC 6.0	0.5	2.6
ALKAIOS	LINUX 2.4.18	GCC 3.2	0.5	0.7
GALOIS	SOLARIS 8	UCBCC 5.0	0.9	13.2
KARNAK	IRIX 6.5 - n32	CC 7.30	2.1	13.2
KARNAK	IRIX 6.5 - n32	CC 7.30	2.6	14.3
KADESH	AIX 5.3 - 32	XLC	1.2	3.5
KADESH	AIX 5.3 - 64	XLC	N/A	4.9

Table 4: Recycling Overhead (usec)

present our measurements for the creation and reuse overheads of `UthLib`. The creation overhead is dominated by the memory allocation call (`malloc`). We observe that having pre-allocated the stacks decreases the overhead of thread creation an order of magnitude.

Finally, Table 5 presents the overhead (in processor cycles) for re-initializing (function and argument) an already created user-level thread.

MACHINE	OPERATING SYSTEM	COMPILERS	SJLJ	MCSC
ALKAIOS	WINDOWS 2000	MSVC 6.0	117	1731
ALKAIOS	LINUX 2.4.18	GCC 3.2	69	577
GALOIS	SOLARIS 8	UCBCC 5.0	118	3448
KARNAK	IRIX 6.5 - n32	CC 7.30	233	3165
KARNAK	IRIX 6.5 - n32	CC 7.30	280	3320
KADESH	AIX 5.3 - 32	XLC	158	983
KADESH	AIX 5.3 - 64	XLC	N/A	1538

Table 5: Re-initialization Overhead (processor cycles)

6 Possible Optimizations

1. 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 POSIX Threads mutexes while non-blocking algorithms can be utilized for the reuse queues.

2. Another possible POSIX Threads compliant self identification mechanism is the use of thread-specific data (`pthread_set,get_specific`).

3. Alternatively, a stack-based implementation for thread self identification can be used. This can be performed by allocating stacks on appropriate page boundaries (`memalign`).

4. 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.

5. The context-switch mechanism can be based on assembly. For example, the user can implement platform-specific versions of `setjmp-longjmp`, which might result in faster code.

An optimized version of `UthLib` will be included in a future software distribution of our work in the European FET-IST POP (Performance Portability of OpenMP) program. For more information on this project you can visit: `www.cepba.upc.es/pop`.

Acknowledgments

We would like to thank the European Center for Parallelism of Barcelona (CEPBA) for providing us access to the Origin and SP3 parallel machines.

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., Polychronopoulos, E. D., Papatheodorou, T. S.: Implementing the Nano-Threads Programming Model on top of POSIX Threads. In Proc. of the 20th IASTED Applied Informatics Conference, Innsburg, Austria, February 2002.

[4] 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.

[5] Hadjidoukas, P. E., Polychronopoulos, E. D., Papatheodorou, T. S.: OpenMP Runtime Support for Clusters of Multiprocessors. In Proc. Intl. Workshop on OpenMP Applications and Tools (WOMPAT '03), Toronto, Canada, June 2003.

[6] 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.

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

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

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