Portable Multithreading

The Signal Stack Trick For User-Space Thread Creation

Ralf S. Engelschall
Technische Universität München (TUM)
rse@engelschall.com, http://www.engelschall.com

Abstract

This paper describes a pragmatic but portable fallback approach for creating and dispatching between the machine contexts of multiple threads of execution on Unix systems that lack a dedicated user-space context switching facility. Such a fallback approach for implementing machine contexts is a vital part of a user-space multithreading environment, if it has to achieve maximum portability across a wide range of Unix flavors. The approach is entirely based on standard Unix system facilities and ANSI-C language features and especially does not require any assembly code or platform specific tricks at all. The most interesting issue is the technique of creating the machine context for threads, which this paper explains in detail. The described approach closely follows the algorithm as implemented by the author for the popular user-space multithreading library *GNU Portable Threads* (*GNU Pth*, [25]) which this way quickly gained the status of one of the most portable user-space multithreading libraries.

Keywords: portability, multithreading, Unix, POSIX, SUSv2, ANSI-C, user-space, context creation, context switching, signal handler, stack, makecontext, switchcontext, sigaltstack, setjmp, longjmp.

Publishing: Early drafts of this paper were distributed with the GNU Pth distribution. The final release version was published on the USENIX Annual Technical Conference, June 18-23, 2000, San Diego, California, USA.

1 Introduction

1.1 Multithreading

The paradigm of programming with multiple threads of execution (aka *multithreading*) is already a very old one and dates back to the decades of programming with *coroutines* [2, 3]. Paradoxically, the use of threads on Unix platforms did not become popular until the early 1990s.

Multithreading Advantages

Multithreading can provide many benefits for applications (good runtime concurrency, parallel programming techniques can be implemented more easily, the popular procedural programming style can be combined with multiple threads of execution, *etc.*) but the most interesting ones are usually performance gains and reduced resource consumption. Because in contrast to multiprocess applications, multithreaded ones usually require less system resources (mainly memory) and their internal communication part can leverage from the shared address space.

Multithreading and Applications

Nevertheless there still exist just a few real applications in the free software world that use multithreading for their benefit, although their application domains are predestined for multithreading. For instance, the popular Apache webserver as of version 1.3 still uses a preforking process model for serving HTTP requests, although two experiments with multithreaded Apache variants in 1996 (with *rsthreads* [27]) and 1998 (with *NSPR* [31]) already showed great performance boosts. The same applies to many similar applications.

The reason for this restraint mainly is that for a long time, multithreading facilities under Unix were rare. The situation became better after some vendors like *Sun* and *DEC* incorporated threading facilities into their Unix flavors and *POSIX* standardized a threading *Application Programming Interface* (API) (aka *Pthreads* [1]). But an API and a few vendor implementations are not enough to fulfill the portability requirements of modern free software packages. Here stand-alone and really portable multithreading environments are needed.

The author collected and evaluated over twenty (mostly user-space) available multithreading facilities for Unix systems (see Table 1), but only a few of them are freely available and showed to be really portable. And even the mostly portable ones suffered from the fact that they partly depend on assembly code or platform specific tricks usually related to the creation and dispatching of the individual threads. This means that the number of platforms they support is limited and applications which are based on these facilities are only portable to those platforms. This situation is not satisfactory, so application authors still avoid the use of multithreading if they want to (or have to) achieve maximum portability for their application.

A pragmatic and mostly portable fallback technique for implementing user-space threads can facilitate wider use of multithreading in free software applications.

Ingredients of a Thread

A Unix process has many ingredients, but the most important ones are its memory mapping table, the signal dispatching table, the signal mask, the set of file descriptors and the machine context. The machine context in turn consists of at least the CPU registers including the program counter and the stack pointer. In addition, there can be light-weight processes (LWP) or threads, which usually share all attributes with the underlying (heavy-weight) process except for the machine context.

Kernel-Space vs. User-Space

Those LWPs or threads, on a Unix platform classically can be implemented either in kernel-space or in user-space. When implemented in kernel-space, one usually calls them LWPs or kernel threads, otherwise (user-space) threads. If threads are implemented by the kernel, the thread context switches are performed by the kernel without notice by the application, similar to the dispatching of processes. If threads are implemented in user-space, the thread context switches are performed usually by an application library without notice by the kernel. Additionally, there exist hybrid threading approaches, where typically a user-space library binds one or more user-space threads to one or more kernel-space LWPs.

Thread Models

The vendor threading facilities under *Sun Solaris*, *IBM AIX*, *DEC Tru64* (formerly *DIGITAL UNIX* or *OSF/1*) and *SGI IRIX* use a **M:N** mapping [21, 30], *i.e.*, **M** user-space threads are mapped onto **N** kernel-space LWPs. On the other hand, *LinuxThreads* [29] under *GNU/Linux* uses a **1:1** mapping and pure user-space implementations like *GNU Pth*, *FSU pthreads* or *MIT pthreads*, *etc.* use a **M:1** mapping [25, 22, 23].

From now on we focus on such **M:1** user space threading approaches, where one or more user space threads are implemented inside a single kernel space process. The exercise is to implement this by using standardized Unix system and ANSI-C language facilities *only*.

1.2 The Exercise

As we have mentioned, a thread shares its state with the underlying process except for the machine context. So the major task for a user-space threading system is to create and dispatch those machine contexts.

In practice, the second major task it has to do is to ensure that no thread by accident blocks the whole process (and thereby all other threads). Instead when an operation would block, the threading library should suspend only the execution of the current thread and in the meantime dispatch the remaining threads. But this task is outside the scope of this paper (see [11] for details about this task). We focus only on the aspect of machine context handling.

1.3 The Curse of Portability

Our goal of real portability for a threading system causes some non-trivial problems which have to be solved. The most obvious one is that dealing with machine contexts usually suffers from portability, because it is a highly CPU dependent task for which not every Unix flavor provides a standardized API. Although such an API would be not too hard for vendors to provide, because in principle it is just a matter of switching a few CPU registers (mainly the program counter and the stack pointer).

Assembly Code Considered Harmful

Additionally, we disallow the use of any assembly solutions or platform specific tricks, because then the threading system again would be only semi-portable, *i.e.*, it can be ported to **N** platforms but on the **(N+1)**th platform one has to manually adjust or even extend it to work there, too.

This is usually not acceptable, even if it also makes solving the problems harder. At least most of the known free software user-space threading systems [22, 23, 24] do not restrict themself to this and therefore are just semi-portable. But real portability should be a major goal.

2 Problem Analysis

2.1 The Task in Detail

Our task is simple in principle: provide an API and corresponding implementation for creating and dispatching machine contexts on which user-space threads can be implemented.

The Proposed API

In detail we propose the following *Application Programmers Interface* (API) for the machine context handling:

- A data structure of type `mctx_t` which holds the machine context.
- A function "**void** `mctx_create(mctx_t` $*$*mctx*, **void** `(`$*$*sf_addr*`)` (**void ***)`,` **void** $*$*sf_arg*,

Package	Genesis	Latest Version	Implementation Space	Thread Mapping	Active Development	Experimental State	Open Source	Pthread API	Pthread Shared Memory	Native API	Native API $\geq$ Pthread API	Native API is Pthread API	Preemptive Scheduling	Portability	Assembly Code	SysCall Wrap.
gnu-pth	1999	1.3.5	user	n:1	yes	no	yes	yes	no	yes	yes	no	no	full/mcsc+sjlj	no	partly
cmu-lwp	1984	1.4	user	n:1	yes	no	yes	no	-	yes	yes	partly	no	semi/fixed:8	yes	no
fsu-pthread	1992	3.13	user	n:1	no	no	yes	yes	no	no	-	-	yes	semi/fixed:6	yes	yes
mit-pthread	1993	1.8.9	user	n:1	no	no	yes	yes	no	no	-	-	yes	semi/fixed:17	yes	yes
ptl	1997	990622	user	n:1	no	no	yes	yes	no	no	-	-	yes	semi/fixed:10	yes	yes
linuxthreads	1997	2.1.2	user+kernel	1:1	yes	no	yes	yes	no	no	-	-	yes	semi/fixed:5	yes	yes
uthread	1998	3.4	user	n:1	yes	no	yes	yes	no	no	-	-	yes	semi/fixed:2	yes	yes
cthread	1991	991115	user	n:1	no	no	yes	no	-	yes	yes	no	no	semi/fixed:8	yes	yes
openthreads/qt	1996	2.0	user	n:1	no	no	yes	no	-	yes	no	no	no	semi/fixed:9	yes	no
rt++/qt	1996	1.0	user	n:1	no	no	yes	no	-	yes	yes	no	no	semi/fixed:9	yes	no
rsthreads	1996	980331	user	n:1	no	yes	yes	no	-	yes	no	no	no	semi/fixed:9	yes	no
pcthread	1996	1.0	user	n:1	no	yes	yes	yes	no	no	-	-	yes	semi/fixed:1	yes	no
bbthreads	1996	0.3	kernel	1:1	no	yes	yes	no	-	yes	no	-	yes	semi/fixed:1	yes	no
jkthreads	1998	1.2	kernel	1:1	no	yes	yes	no	-	yes	no	-	yes	semi/fixed:1	yes	no
nthreads	1997	970604	user	n:1	no	yes	yes	no	-	yes	no	-	no	semi/fixed:9	yes	partly
rexthreads	1993	930614	user	n:1	no	yes	yes	no	-	yes	no	-	no	semi/fixed:4	yes	no
coro	1999	1.0.3	user	n:1	no	yes	yes	no	-	yes	no	-	no	semi/fixed:1	yes	no
greenthreads	1995	1.2	user	n:1	no	no	no	no	-	yes	yes	-	yes	full/mcsc	no	no
solaris-pthread	NN	2.7	user+kernel	n:m	yes	no	no	yes	yes	yes	yes	no	yes	NN	NN	yes
tru64-pthread	NN	5.0	user+kernel	n:m	yes	no	no	yes	yes	no	no	no	yes	NN	NN	yes
aix-pthread	NN	4.3	user+kernel	1:1	yes	no	no	yes	yes	no	no	no	yes	NN	NN	yes

Table 1: Summary of evaluated multithreading packages and some of their determined characteristics. Notice that mostly all packages contain assembly code and are just semi-portable, *i.e.*, they support only a fixed set of platforms and do not automatically adjust for new ones.

void **sk_addr*, **size_t** *sk_size*)" which creates and initializes a machine context structure in *mctx* with a start function *sf_addr*, a start function argument *sf_arg*, and a stack starting at *sk_addr*, which is *sk_size* bytes in size.

- A function "**void** `mctx_save(mctx_t` **mctx*`)`" which saves the current machine context into the machine context structure *mctx*.

- A function "**void** `mctx_restore(mctx_t` **mctx*`)`" which restores the new machine context from the machine context structure *mctx*. This function does not return to the caller. Instead it does return at the location stored in *mctx* (which is either *sf_addr* from a previous `mctx_create` call or the location of a previous `mctx_save` call).

- A function "**void** `mctx_switch(mctx_t` **mctx_old*, `mctx_t` **mctx_new*`)`" which switches from the current machine context (saved to *mctx_old* for later use) to a new context (restored from *mctx_new*). This function returns only to the caller if `mctx_restore` or `mctx_switch` is again used on *mctx_old*.

2.2 Technical Possibilities

Poking around in the references of the ANSI-C language reference and the Unix standards show the following functions on which an implementation can be based:

- There is the `ucontext`(3) facility with the functions `getcontext`(3), `makecontext`(3), `swapcontext`(3) and `setcontext`(3) which conform to the *Single Unix Specification*, Version 2 (*SUSv2* [20], aka *Unix95/98*). Unfortunately these are available on modern Unix platforms only.

- There are the `jmp_buf` based functions `setjmp`(3) and `longjmp`(3) which conform to ISO 9899:1990 (ISO-C) and the `sigjmp_buf` based `sigsetjmp`(3) and `siglongjmp`(3) functions which conform to IEEE Std1003.1-1988 (*POSIX*), and *Single Unix Specification*, Version 2 (*SUSv2* [20], aka *Unix95/98*). The first two functions are available really on all Unix platforms, the last two are available only on some of them.

  On some platforms `setjmp`(3) and `longjmp`(3) save and restore also the signal mask (if one does not want this semantics, one has to call `_setjmp`(3) and `_longjmp`(3) there) while on others one has to explicitly use the superset functions sigsetjmp(3) and siglongjmp(3) for this. In our discussion we can assume that setjmp(3) and longjmp(3) save and restore the signal mask, because if this is not the case in practice, one easily can replace them with sigsetjmp(3) and siglongjmp(3) calls (if available) or (if not available) emulate the missing functionality manually with additional sigprocmask(2) calls (see pth_mctx.c in *GNU Pth* [25]).

- There is the function sigaltstack(2) which conforms to the *Single Unix Specification*, Version 2 (*SUSv2* [20], aka *Unix95/98*) and its ancestor function sigstack(2) from *4.2BSD*. The last one exists only on *BSD*-derived platforms, but the first function already exists on all current Unix platforms.

2.3 Maximum Portability Solution

The maximum portable solution obviously is to use the standardized makecontext(3) function to create threads and switchcontext(3) or getcontext(3)/setcontext(3) to dispatch them. And actually these are the preferred functions modern user-space multithreading systems are using. We could easily implement our proposed API as following (all error checks omitted for better readability):

```
/* machine context data structure */
typedef struct mctx_st {
    ucontext_t uc;
} mctx_t;

/* save machine context */
#define mctx_save(mctx) \
    (void)getcontext(&(mctx)->uc)

/* restore machine context */
#define mctx_restore(mctx) \
    (void)setcontext(&(mctx)->uc)

/* switch machine context */
#define mctx_switch(mctx_old,mctx_new) \
    (void)swapcontext(&((mctx_old)->uc), \
                      &((mctx_new)->uc))

/* create machine context */
void mctx_create(
    mctx_t *mctx,
    void (*sf_addr)(void *), void *sf_arg,
    void *sk_addr, size_t sk_size)
{
    /* fetch current context */
    getcontext(&(mctx->uc));

    /* adjust to new context */
    mctx->uc.uc_link           = NULL;
    mctx->uc.uc_stack.ss_sp    = sk_addr;
    mctx->uc.uc_stack.ss_size  = sk_size;
    mctx->uc.uc_stack.ss_flags = 0;

    /* make new context */
    makecontext(&(mctx->uc),
                sf_addr, 1, sf_arg);
    return;
}
```

Unfortunately there are still lots of Unix platforms where this approach cannot be used, because the standardized ucontext(3) API is not provided by the vendor. Actually the platform test results for *GNU Pth* (see Table 2 below) showed that only 7 of 21 successfully tested Unix flavors provided the standardized API (makecontext(3), *etc.*). On all other platforms, *GNU Pth* was forced to use the fallback approach of implementing the machine context as we will describe in the following. Obviously this fallback approach has to use the remaining technical possibilities (sigsetjmp(3), *etc.*).

Operating System	Architecture(s)	mcsc	sjlj
FreeBSD 2.x/3.x	Intel	no	yes
FreeBSD 3.x	Intel, Alpha	no	yes
NetBSD 1.3/1.4	Intel, PPC, M68K	no	yes
OpenBSD 2.5/2.6	Intel, SPARC	no	yes
BSDI 4.0	Intel	no	yes
Linux 2.0.x glibc 1.x/2.0	Intel, SPARC, PPC	no	yes
Linux 2.2.x glibc 2.0/2.1	Intel, Alpha, ARM	no	yes
Sun SunOS 4.1.x	SPARC	no	yes
Sun Solaris 2.5/2.6/2.7	SPARC	yes	yes
SCO UnixWare 2.x/7.x	Intel	yes	yes
SCO OpenServer 5.0.x	Intel	no	yes
IBM AIX 4.1/4.2/4.3	RS6000, PPC	yes	yes
HP HPUX 9.10/10.20	HPPA	no	yes
HP HPUX 11.0	HPPA	yes	yes
SGI IRIX 5.3	MIPS 32/64	no	yes
SGI IRIX 6.2/6.5	MIPS 32/64	yes	yes
ISC 4.0	Intel	no	yes
Apple MacOS X	PPC	no	yes
DEC OSF1/Tru64 4.0/5.0	Alpha	yes	yes
SNI ReliantUNIX	MIPS	yes	yes
AmigaOS	M68K	no	yes

Table 2: Summary of operating system support. The level and type of support found on each tested operating system. mcsc: functional makecontext(3)/switchcontext(3), sjlj: functional setjmp(3)/longjmp(3) or sigsetjmp(3)/siglongjmp(3). See file PORTING in *GNU Pth* [25] for more details.

2.4 Remaining Possibilities

Our problem can be divided into two parts, an easy one and a difficult one.

The Easy Part

That setjmp(3) and longjmp(3) can be used to implement user-space threads is commonly known [24, 27, 28]. Mostly all older portable user-space threading libraries are based on them, although some problems are known with these facilities (see below). So it becomes clear that we also have to use these functions and base our machine context (mctx_t) on their jmp_buf data structure.

We immediately recognize that this way we have at least solved the dispatching problem, because our mctx_save, mctx_restore and mctx_switch functions can be easily implemented with setjmp(3) and longjmp(3).

The Difficult Part

Nevertheless, the difficult problem of how to create the machine context remains. Even knowing that our machine context is jmp_buf based is no advantage to us. A jmp_buf has to be treated by us as an opaque data structure — for portability reasons. The only operations we can perform on it are setjmp(3) and longjmp(3) calls, of course. Additionally, we are forced to use sigaltstack(3) for our stack manipulations, because it is the only portable function which actually deals with stacks.

So it is clear that our implementation for mctx_create has to play a few tricks to use a jmp_buf for passing execution control to an arbitrary startup routine. And our approach has to be careful to ensure that it does not suffer from unexpected side-effects. It should be also obvious that we cannot again expect to find an easy solution (as for mctx_save, mctx_restore and mctx_switch), because setjmp(3) and sigaltstack(3) cannot be trivially combined to form mctx_create.

3 Implementation

As we have already discussed, our implementation contains an easy part (mctx_save, mctx_restore and mctx_switch) and a difficult part (mctx_create). Let us start with the easy part, whose implementation is obvious (all error checks again omitted for better readability):

```
/* machine context data structure */
typedef struct mctx_st {
    jmp_buf jb;
} mctx_t;

/* save machine context */
#define mctx_save(mctx) \
    (void)setjmp((mctx)->jb)

/* restore machine context */
#define mctx_restore(mctx) \
    longjmp((mctx)->jb, 1)

/* switch machine context */
#define mctx_switch(mctx_old,mctx_new) \
    if (setjmp((mctx_old)->jb) == 0) \
        longjmp((mctx_new)->jb, 1)

/* create machine context */
void mctx_create(
    mctx_t *mctx,
    void (*sf_addr)(void *), void *sf_arg,
    void *sk_addr, size_t sk_size)
{
    ...initialization of mctx to be filled in...
}
```

There is one subtle but important point we should mention: The use of the C pre-processor #define directive to implement mctx_save, mctx_restore and mctx_switch is intentional. For technical reasons related to setjmp(3) semantics and return related stack behavior (which we will explain later in detail) we *cannot* implement these three functions (at least not mctx_switch) as C functions if we want to achieve maximum portability across all platforms. Instead they have to be implemented as pre-processor macros.

3.1 Algorithm Overview

The general idea for mctx_create is to configure the given stack as a signal stack via sigaltstack(2), send the current process a signal to transfer execution control onto this stack, save the machine context there via setjmp(3), get rid of the signal handler scope and bootstrap into the startup routine.

The real problem in this approach comes from the signal handler scope which implies various restrictions on Unix platforms (the signal handler scope often is just a flag in the process control block (PCB) which various system calls, like sigaltstack(2), check before allowing the operation – but because it is part of the process state the kernel manages, the process cannot change it itself). As we will see, we have to perform a few tricks to get rid of it. The second main problem is: how do we prepare the calling of the start routine without immediately entering it?

3.2 Algorithm

The input to the mctx_create function is the machine context structure *mctx* which should be initialized, the thread startup function address *sf_addr*, the thread startup function argument *sf_arg* and a chunk of memory starting at *sk_addr* and *sk_size* bytes in size, which should become the threads stack.

The following algorithm for mctx_create is directly modeled after the implemented algorithm one can find in *GNU Pth* [25], which in turn was derived from techniques originally found in *rsthreads* [27]:

1. Preserve the current signal mask and block an arbitrary worker signal (we use SIGUSR1, but any signal can be used for this – even an already used one). This worker signal is later temporarily required for the trampoline step.

2. Preserve a possibly existing signal action for the worker signal and configure a trampoline function as the new temporary signal action. The signal delivery is configured to occur on an alternate signal stack (see next step).

3. Preserve a possibly active alternate signal stack and configure the memory chunk starting at *sk_addr* as the new temporary alternate signal stack of length *sk_size*.

4. Save parameters for the trampoline step (*mctx*, *sf_addr*, *sf_arg*, *etc.*) in global variables, send the current process the worker signal, temporarily unblock it and this way allow it to be delivered on the signal stack in order to transfer execution control to the trampoline function.

5. After the trampoline function asynchronously entered, save its machine context in the *mctx* structure and immediately return from it to terminate the signal handler scope.

6. Restore the preserved alternate signal stack, preserved signal action and preserved signal mask for worker signal. This way an existing application configuration for the worker signal is restored.

7. Save the current machine context of `mctx_create`. This allows us to return to this point after the next trampoline step.

8. Restore the previously saved machine context of the trampoline function (*mctx*) to again transfer execution control onto the alternate stack, but this time without(!) signal handler scope.

9. After reaching the trampoline function (*mctx*) again, immediately bootstrap into a clean stack frame by just calling a second function.

10. Set the new signal mask to be the same as the original signal mask which was active when `mctx_create` was called. This is required because in the first trampoline step we usually had at least the worker signal blocked.

11. Load the passed startup information (*sf_addr*, *sf_arg*) from `mctx_create` into local (stack-based) variables. This is important because their values have to be preserved in machine context dependent memory until the created machine context is the first time restored by the application.

12. Save the current machine context for later restoring by the calling application.

13. Restore the previously saved machine context of `mctx_create` to transfer execution control back to it.

14. Return to the calling application.

When the calling application now again switches into the established machine context *mctx*, the thread starts running at routine *sf_addr* with argument *sf_arg*. Figure 1 illustrates the algorithm (the numbers refer to the algorithm steps listed above).

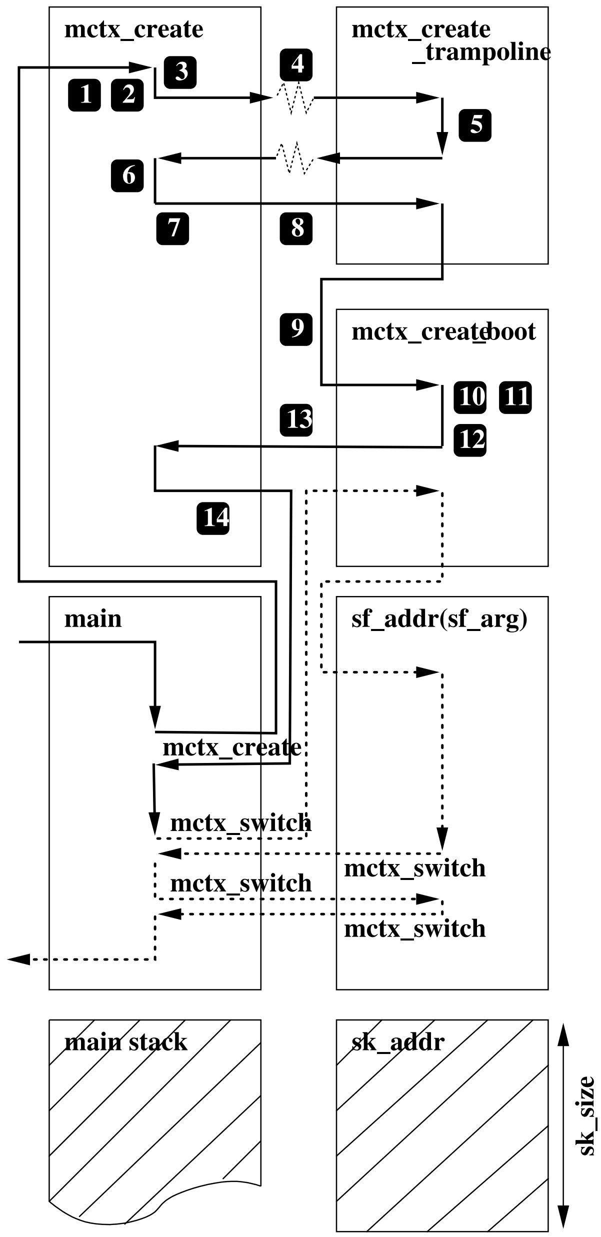

Figure 1: Illustration of the machine context creation procedure. The thick solid lines and numeric marks correspond to the algorithm steps as described in section 3.2. The thick dotted lines show a possible further processing where a few context switches are performed to dispatch between the main thread and the new created thread.

3.3 Source Code

The corresponding ANSI-C code, which implements `mctx_create`, is a little bit more complicated. But with the presented algorithm in mind, it is now straightforward.

```
static mctx_t       mctx_caller;
static sig_atomic_t mctx_called;

static mctx_t      *mctx_creat;
static void       (*mctx_creat_func)(void *);
static void        *mctx_creat_arg;
static sigset_t     mctx_creat_sigs;

void mctx_create(
    mctx_t *mctx,
    void (*sf_addr)(void *), void *sf_arg,
    void *sk_addr, size_t sk_size)
{
    struct sigaction sa;
    struct sigaction osa;
    struct sigaltstack ss;
    struct sigaltstack oss;
    sigset_t osigs;
    sigset_t sigs;

    /* Step 1: */
    sigemptyset(&sigs);
    sigaddset(&sigs, SIGUSR1);
    sigprocmask(SIG_BLOCK, &sigs, &osigs);

    /* Step 2: */
    memset((void *)&sa, 0,
           sizeof(struct sigaction));
    sa.sa_handler = mctx_create_trampoline;
    sa.sa_flags = SA_ONSTACK;
    sigemptyset(&sa.sa_mask);
    sigaction(SIGUSR1, &sa, &osa);

    /* Step 3: */
    ss.ss_sp    = sk_addr;
    ss.ss_size  = sk_size;
    ss.ss_flags = 0;
    sigaltstack(&ss, &oss);

    /* Step 4: */
    mctx_creat      = mctx;
    mctx_creat_func = sf_addr;
    mctx_creat_arg  = sf_arg;
    mctx_creat_sigs = osigs;
    mctx_called     = FALSE;
    kill(getpid(), SIGUSR1);
    sigfillset(&sigs);
    sigdelset(&sigs, SIGUSR1);
    while (!mctx_called)
        sigsuspend(&sigs);

    /* Step 6: */
    sigaltstack(NULL, &ss);
    ss.ss_flags = SS_DISABLE;
    sigaltstack(&ss, NULL);
    if (!(oss.ss_flags & SS_DISABLE))
        sigaltstack(&oss, NULL);
    sigaction(SIGUSR1, &osa, NULL);
    sigprocmask(SIG_SETMASK,
                &osigs, NULL);

    /* Step 7 & Step 8: */
    mctx_switch(&mctx_caller, mctx);

    /* Step 14: */
    return;
}

void mctx_create_trampoline(int sig)
{
    /* Step 5: */
    if (mctx_save(mctx_creat) == 0) {
        mctx_called = TRUE;
        return;
    }

    /* Step 9: */
    mctx_create_boot();
}

void mctx_create_boot(void)
{
    void (*mctx_start_func)(void *);
    void *mctx_start_arg;

    /* Step 10: */
    sigprocmask(SIG_SETMASK,
                &mctx_creat_sigs, NULL);

    /* Step 11: */
    mctx_start_func = mctx_creat_func;
    mctx_start_arg  = mctx_creat_arg;

    /* Step 12 & Step 13: */
    mctx_switch(mctx_creat, &mctx_caller);

    /* The thread ``magically'' starts... */
    mctx_start_func(mctx_start_arg);

    /* NOTREACHED */
    abort();
}
```

3.4 Run-time Penalty

After this discussion of the implementation details, an obviously occuring question now is what the expected run-time penalty is. That is, what does our presented machine context implementation cost compared to a `ucontext`(3) based solution. From the already discussed details we can easily guess that our complex machine context creation procedure (`mctx_create`) will be certainly noticeably slower than a solution based on a `ucontext`(3) facility.

But a wild guess is not sufficing for a reasonable statement. So we have written a *Simple Machine Context Benchmark* (SMCB [32]) which was used to compare run-time costs of the `mctx_create` and `mctx_switch` functions if once implemented through the *POSIX* `makecontext`(3)/`swapcontext`(3) functions (as shown in section 2.3), and once implemented with our based fallback implementation (for convenience reasons we directly used `sigjmp_buf`, `sigsetjmp`(3) and `siglongjmp`(3) in the benchmark, because all tested platforms provided this). The results are shown Table 3 below.

As one can derive from these evaluations, our signal stack trick to implement mctx_create in practice is approximately 15 times slower than the makecontext(3) based variant. This cost should not be neglected. On the other hand, the sigsetjmp(3)/ siglongjmp(3) based mctx_switch performs about as good as the swapcontext(3) based variant (the reason why on most of the tested platforms it is even slightly faster is not known – but we guess it is related to a greater management overhead in the ucontext(3) facility, which is a superset of the functionality we require). Or in short: our presented fallback approach costs noticeable extra CPU cycles on thread creation time, but is as fast as the standardized solution under thread dispatching time.

10000 × mctx_create (in seconds):

Platform	mcsc	sjlj	overhead
Sun Solaris 2.6 (SPARC)	0.076	1.268	16.7
DEC Tru64 5.0 (Alpha)	0.019	0.235	12.4
SGI IRIX 6.5 (MIPS)	0.105	1.523	14.5
SCO UnixWare 7.0 (Intel)	0.204	3.827	18.8
HP HP/UX 11.0 (HPPA)	0.057	0.667	11.8
		Average:	**14.8**

10000 × mctx_switch (in seconds):

Platform	mcsc	sjlj	overhead
Sun Solaris 2.6 (SPARC)	0.137	0.210	1.5
DEC Tru64 5.0 (Alpha)	0.034	0.022	0.6
SGI IRIX 6.5 (MIPS)	0.235	0.190	0.8
SCO UnixWare 7.0 (Intel)	0.440	0.398	0.9
HP HP/UX 11.0 (HPPA)	0.106	0.065	0.6
		Average:	**0.9**

Table 3: Summary of *Simple Machine Context Benchmark* (SMCB, [32]). The speed of machine context creation and switching found on each tested operating system. **mcsc**: functional makecontext(3) / switchcontext(3), **sjlj**: functional sigsetjmp(3)/siglongjmp(3). **overhead**: the overhead of using **sjlj** instead of **mcsc**.

3.5 Remaining Issues

The presented algorithm and source code can be directly used in practice for implementing a minimal threading system or the concept of co-routines. Its big advantage is that if the operating system provides the required standardized primitives, we do not need to know anything at all about the machine we are running on — everything just works. Nevertheless, there remain a few special issues we still have to discuss.

The Waggly longjmp(3) after Return

On some platforms, longjmp(3) may not be called after the function which called the setjmp(3) returned. When this is done, the stack frame situation is not guaranteed to be in a clean and consistent state. But this is exactly the mechanism we use in order to get rid of the signal handler scope in step 5.

The only alternative would be to leave the signal handler via longjmp(3), but then we would have another problem, as experience showed. For instance, ROBERT S. THAU's *Really Simple Threads* (*rsthreads*) [27] was ported to several platforms and was used to run an experimental multithreaded version of the Apache webserver. THAU's approach was similar to ours, but differed significantly in the way the signal handler is left. In particular, in an attempt to avoid the unsafe stack frame, it used a longjmp(3) call to leave the signal handler, rather than returning from it. But this approach does not work on some *SysV*-derived kernels, as we already mentioned.

The problem is that these kernels do not "believe" that the code is out of the signal-handling context, until the signal handler has returned — and accordingly, refuse to allow readjustment of the signal stack until it has. But with the *rsthreads* approach, the signal handler that created the first thread never returns, and when *rsthreads* wants to create the second thread, these kernels refuse to readjust the signal stack, and we are stuck. So with portability in mind, we decided that it is better to get rid of the signal handler scope with the straight-forward "return" and instead fight the mentioned (simpler) problem of an unsafe stack frame.

Fortunately, in practice this is not as problematic as it seems, because evaluations (for *GNU Pth*) on a wide range of current Unix platforms showed that one can reach a safe stack frame again by just calling a function. That is the reason why our algorithm enters the second trampoline function in step 9.

The Uncooperative longjmp(3)

Even on operating systems which have working *POSIX* functions, our approach may theoretically still not work, because longjmp(3) does not cooperate. For instance, on some platforms the standard *libc* longjmp(3) branches to error-handling code if it detects that the caller tries to jump *up* the stack, *i.e.*, into a stack frame that has already returned.

This is usually implemented by comparing the current stack pointer to the one in the jmp_buf structure. That is why it is important for our algorithm to return from the signal handler and this way enter the (different) stack of the parent thread. In practice, the implementation in *GNU Pth* showed that then one no longer suffers from those uncooperative longjmp(3) implementations, but one should keep this point in mind when reaching even more uncooperative variants on esoteric Unix platforms. If it still occurs, one can only try to resume the operation by using a possibly existing platform-specific error handling hook.

Garbage at Bottom of Stacks

There is a subtle side-effect of our implementation: there remains some garbage at the bottom of each thread stack. The reason is that if a signal is delivered, the operating system pushes some state onto the stack, which is restored later, when the signal handler returns. But although we return from the signal handler, we jump in again, and this time we enter not directly at the bottom of the stack, because of the `setjmp`(3) in the trampoline function.

Since the operating system has to capture all CPU registers (including those that are ordinarily scratch registers or caller-save registers), there can be a fair amount of memory at the bottom of the established thread stack. For some systems this can be even up to 1 KB of garbage [27]. But except for the additional memory consumption it does not hurt.

We just have to keep in mind this additional stack consumption when deciding the stack size (*sk_size*). A reasonable stack size usually is between 16 and 32 KB. Less is neither reasonable nor always allowed (current Unix platforms usually require a stack to be at least 16 KB in size).

Stack Overflows

There is a noticeable difference between the initial `main`() thread and the explicitly spawned threads: the initial thread runs on the standard process stack. This stack automatically can grow under Unix, while the stacks of the spawned threads are fixed in size. So stack overflows can occur for the spawned threads. This implies that the parent has to make a reasonable guess of the threads stack space requirement already at spawning time.

And there is no really portable solution to this problem, because even if the thread library's scheduler can detect the stack overflow, it cannot easily resize the stack. The reason is simply that the stack initialization goes hand in hand with the initialization of the start routine, as we discussed before. And this start routine has to be a real C function in order to *call*. But once the thread is running, there no longer exists such an entry point. So, even if the scheduler would be able to give the thread a new enlarged stack, there is no chance to restart the thread on this new stack.

Or more correct, there is no *portable* way to achieve it. As with the previous problems, there is a non-portable solution. That is why our implementation did not deal with this issue. Instead in practice one usually lets the scheduler just detect the stack overflow and terminate the thread. This is done by using a red zone at the top of the stack which is marked with a magic value the scheduler checks between thread dispatching operations.

Resizing solutions are only possible in semi-portable ways. One approach is to place the thread stacks into a memory mapped area (see `mmap`(2)) of the process address space and let the scheduler catch `SIGSEGV` signals. When such a signal occurs, because of a stack overflow in this area, the scheduler explicitly resizes the memory mapped area. This resizing can be done either by copying the stack contents into a new larger area which is then re-mapped to the old address or via an even more elegant way, as the vendor threading libraries of *Sun Solaris*, *FreeBSD* and *DEC Tru64* do it: the thread stacks are allocated inside memory mapped areas which are already initially a few MB in (virtual) size and then one just relies on the virtual memory system's feature that only the actually consumed memory space is mapped.

Startup Routine Termination

There is a cruel `abort`(3) call at the end of our `mctx_create_boot` function. This means, if the startup routine would return, the process is aborted. That is obviously not reasonable, so why have we written it this way?

If the thread returns from the startup routine, it should be cleanly terminated. But it cannot terminate itself (for instance, because it cannot free its own stack while running on it, *etc.*). So the termination handling actually is the task of the thread library scheduler. As a consequence, the thread spawning function of a thread library should be not directly `mctx_create`.

Instead the thread spawning function should use an additional trampoline function as the higher-level startup routine. And this trampoline function performs a context switch back into the thread library scheduler before the lower-level startup routine would return. The scheduler then can safely remove the thread and its machine context. That is why the `abort`(3) call is never reached in practice (more details can be found in the implementations of `pth_spawn` and `pth_exit` in `pth_lib.c` of *GNU Pth* [25])

The sigstack(2) Fallback Situation

Not all platforms provide the standardized `sigaltstack`(2). Instead they at least provide the *4.2BSD* ancestor function `sigstack`(2). But one cannot trivially replace `sigaltstack`(2) by `sigstack`(2) in this situation, because in contrast to `sigaltstack`(2), the old `sigstack`(2) does not automatically handle the machine dependent direction of stack growth.

Instead, the caller has to know the direction and always call `sigstack`(2) with the address of the bottom of the stack. So, in a real-world implementation one first has to determine the direction of stack growth in order to use `sigstack`(2) as a replacement for `sigaltstack`(2). Fortunately this is easier than it seems on the first look (for details see the macros `AC_CHECK_STACKGROWTH` and `AC_CHECK_STACKSETUP` in file `aclocal.m4` from *GNU Pth* [25]). Alternatively if one can afford to waste memory, one can use an elegant trick: to set up a stack of size N, one allocates a chunk of memory (starting at address A) of size $N \times 2$ and then calls `sigstack`(2) with the parameters *sk_addr*$=A+N$ and *sk_size*$=N$, *i.e.*, one specifies the middle of the memory chunk as the stack base.

The Blind Alley of Brain-Dead Platforms

The world would not be as funny as it is, if really all Unix platforms would be fair to us. Instead, currently at least one platform exists which plays unfair: unfortunately, ancient versions of the popular *GNU/Linux*. Although we will discover that it both provides `sigaltstack`(2) and `sigstack`(2), our approach won't work on *Linux* kernels prior to version 2.2 and *glibc* prior to version 2.1.

Why? Because its *libc* provides only stubs of these functions which always return just `-1` with `errno` set to `ENOSYS`. So, this definitely means that our nifty algorithm is useless there, because its central point *is* `sigaltstack`(2)/`sigstack`(2). Nevertheless we do not need to give up. At least not, if we, for a single brain-dead platform, accept to break our general goal of not using any platform dependent code.

So, what can we actually do here? All we have to do, is to fiddle around a little bit with the machine-dependent `jmp_buf` ingredients (by poking around in `setjmp.h` or by disassembling `longjmp`(3) in the debugger). Usually one just has to do a `setjmp`(3) to get an initial state in the `jmp_buf` structure and then manually adjust two of its fields: the program counter (usually a structure member with "`pc`" in the name) and the stack pointer (usually a structure member with "`sp`" in the name).

That is all and can be acceptable for a real-world implementation which really wants to cover mostly *all* platforms – at least as long as the special treatment is needed just for one or two platforms. But one has to keep in mind that it at least breaks one of the initial goals and has to be treated as a last chance solution.

Functions sigsetjmp(3) and siglongjmp(3)

One certainly wants the *POSIX* thread semantics where a thread has its own signal mask. As already mentioned, on some platforms `setjmp`(3) and `longjmp`(3) do not provide this and instead one has to explicitly call `sigsetjmp`(3) and `siglongjmp`(3) instead. There is only one snare: on some platforms `sigsetjmp`(3)/`siglongjmp`(3) save also information about the alternate signals stack. So here one has to make sure that although the thread dispatching later uses `sigsetjmp`(3)/`siglongjmp`(3), the thread creation step in `mctx_create` still uses plain `setjmp`(3)/`longjmp`(3) calls for the trampoline trick. One just has to be careful because the `jmp_buf` and `sigjmp_buf` structures cannot be mixed between calls to the `sigsetjmp`(3)/`siglongjmp`(3) and `setjmp`(3)/`longjmp`(3).

More Machine Context Ingredients

Finally, for a real-world threading implementation one usually wants to put more state into the machine context structure `mctx_t`. For instance to fulfill more *POSIX* threading semantics, it is reasonable to also save and restore the global `errno` variable. All this can be easily achieved by extending the `mctx_t` structure with additional fields and by making the `mctx_save`, `mctx_restore` and `mctx_switch` functions to be aware of them.

3.6 Related Work

Beside *GNU Pth* [25], there are other multithreading libraries which use variants of the presented approach for implementing machine contexts in user-space. Most notably there are ROBERT S. THAU's *Really Simple Threads* (*rsthreads*, [27]) package which uses `sigaltstack`(2) in a very similar way for thread creation, and KOTA ABE's *Portable Thread Library* (*PTL*, [24]) which uses a `sigstack`(2) approach. But because their approaches handle the signal handler scope differently, they are not able to achieve the same amount of portability and this way do not work for instance on some System-V-derived platforms.

3.7 Summary & Availability

We have presented a pragmatic and mostly portable fallback approach for implementing the machine context for user-space threads, based entirely on Unix system and ANSI-C language facilities. The approach was successfully tested in practice on a wide range of Unix flavors by *GNU Pth* and should also adapt to the remaining Unix platforms as long as they adhere to the relevant standards.

The *GNU Pth* package is distributed under the GNU Library General Public License (LGPL 2.1) and freely available from *http://www.gnu.org/software/pth/* and *ftp://ftp.gnu.org/gnu/pth/*.

3.8 Acknowledgements

I would like to thank ROBERT S. THAU, DAVID BUTENHOF, MARTIN KRAEMER, ERIC NEWTON and BRUNO HAIBLE for their comments which helped to write the initial version of this paper. Additionally, credit has to be given to CHRISTOPHER SMALL and the USENIX reviewers for their invaluable feedback which allowed this paper to be extended, cleaned up and finally published at the USENIX Annual Technical Conference 2000. Finally, thanks go to all users of *GNU Pth* for their feedback on the implementation, which helped in fine-tuning the presented approach. [rse]

References

[1] *POSIX 1003.1c Threading*, IEEE POSIX 1003.1c-1995, ISO/IEC 9945-1:1996

[2] M.E. CONWAY: *Design of a separable transition-diagram compiler.*, Comm. ACM 6:7, 1963, p.396-408

[3] E.W. DIJKSTRA: *Co-operating sequential processes*, in F. Genuys (Ed.), *Programming Languages*, NATO Advanced Study Institute, Academic Press, London, 1965, p.42-112.

[4] B. NICHOLS, D. BUTTLAR, J.P. FARREL: *Pthreads Programming - A POSIX Standard for Better Multiprocessing*, O'Reilly, 1996; ISBN 1-56592-115-1

[5] B. LEWIS, D. J. BERG: *Threads Primer - A Guide To Multithreaded Programming*, Prentice Hall, 1996; ISBN 0-13-443698-9

[6] S. J. NORTON, M. D. DIPASQUALE: *Thread Time - The Multithreaded Programming Guide*, Prentice Hall, 1997; ISBN 0-13-190067-6

[7] D. R. BUTENHOF: *Programming with POSIX Threads*, Addison Wesley, 1997; ISBN 0-201-63392-2

[8] S. PRASAD: *Multithreading Programming Techniques*, McGraw-Hill, 1996; ISBN 0-079-12250-7

[9] S. KLEINMAN, B. SMALDERS, D. SHAH: *Programming with Threads*, Prentice Hall, 1995; ISBN 0-131-72389-8

[10] C.J. NORTHRUP: *Programming With Unix Threads*, John Wiley & Sons, 1996; ISBN 0-471-13751-0

[11] P. BARTON-DAVIS, D. MCNAMEE, R. VASWANI, E. LAZOWSKA: *Adding Scheduler Activations to Mach 3.0*, University of Washington, 1992; Technical Report 92-08-03

[12] D. STEIN, D. SHAH: *Implementing Lightwight Threads*, SunSoft Inc., 1992 (published at USENIX'92).

[13] W.R.STEVENS: *Advanced Programming in the Unix Environment*, Addison-Wesley, 1992; ISBN 0-201-56317-7

[14] D. LEWINE: *POSIX Programmer's Guide: Writing Portable Unix Programs*, O'Reilly & Associates,Inc., 1994; ISBN 0-937175-73-0

[15] BRYAN O'SULLIVAN: *Frequently asked questions for comp.os.research*, 1995; http://www.serpentine.com/~bos/os-faq/, ftp://rtfm.mit.edu/pub/usenet/comp.os.research/

[16] SUN MICROSYSTEMS, INC: *Threads Frequently Asked Questions*, 1995, http://www.sun.com/workshop/-threads/faq.html

[17] BRYAN O'SULLIVAN: *Frequently asked questions for comp.programming.threads*, 1997; http://www.serpentine.com/~bos/threads-faq/.

[18] BIL LEWIS: *Frequently asked questions for comp.programming.threads*, 1999; http://-www.lambdacs.com/newsgroup/FAQ.html

[19] NUMERIC QUEST INC: *Multithreading - Definitions and Guidelines*; 1998; http://www.numeric-quest.com/lang/multi-frame.html

[20] THE OPEN GROUP: *The Single Unix Specification, Version 2 - Threads*; 1997; http://www.opengroup-.org/onlinepubs/007908799/xsh/threads.html

[21] SUN MICROSYSTEMS INC: *SMI Thread Resources*; http://www.sun.com/workshop/threads

[22] FRANK MUELLER: *FSU pthreads*; 1997; http://www-.cs.fsu.edu/~mueller/pthreads/

[23] CHRIS PROVENZANO: *MIT pthreads*; 1993; http://www.mit.edu/people/proven/pthreads.html (old), http://www.humanfactor.com/pthreads/mit-pthreads.html (updated)

[24] KOTA ABE: *Portable Threading Library* (PTL); 1999; http://www.media.osaka-cu.ac.jp/~k-abe/PTL/

[25] RALF S. ENGELSCHALL: *GNU Portable Threads* (Pth); 1999; http://www.gnu.org/software/pth/, ftp://ftp.gnu-.org/gnu/pth/

[26] MICHAEL T. PETERSON: *POSIX and DCE Threads For Linux* (PCThreads); 1995; http://members.aa-.net/~mtp/PCthreads.html

[27] ROBERT S. THAU: *Really Simple Threads* (rsthreads); 1996; ftp://ftp.ai.mit.edu/pub/rst/

[28] JOHN BIRRELL: *FreeBSD uthreads*; 1998; ftp://ftp.freebsd.org/pub/FreeBSD/FreeBSD-current/src/lib/libc_r/uthread/

[29] XAVIER LEROY: *The LinuxThreads library*; 1999; http://pauillac.inria.fr/~xleroy/linuxthreads/

[30] IBM: *AIX Version 4.3 General Programming Concepts: Writing and Debugging Programs; Understanding Threads*; 1998; http://www.rs6000.ibm.com/doc_link/-en_US/a_doc_lib/aixprggd/genprogc/understanding-_threads.htm

[31] *Netscape Portable Runtime* (NSPR); http://-www.mozilla.org/docs/refList/refNSPR/, http://-lxr.mozilla.org/seamonkey/source/nsprpub/

[32] RALF S. ENGELSCHALL: *Simple Machine Context Benchmark*; 2000; http://www.gnu.org/software/pth-/smcb.tar.gz