Pthreads Library Interface

Frank Mueller
Department of Computer Science
Florida State University
Tallahassee, Florida 32306-4019
phone: (904) 644-3441, *e-mail: mueller@cs.fsu.edu*

April 18, 1996

Abstract

This document describes the interface of the Pthreads library developed at Florida State University. The implementation is based on the standard proposal "POSIX 1003.4a Threads Extension, Draft 6". The implementation is currently limited to the Sun SPARC architecture and the SunOS 4.1.x or Solaris 2.x operating systems. The package should be portable under other BSD, SVR4, or POSIX compliant UNIX systems.

1 Introduction

This document is a short description of the interface of the Pthreads library distributed by Florida State University. The POSIX 1003.4a Threads standard Draft 6 superseeds this document. Thus, any differences to the POSIX.4a standard should be interpreted in favor of the standard.

This document is meant for programmers who want to use the Pthreads library but do not have access to the POSIX.4a standard, in other words, are too lazy to order the standard document. Even after you read this document and use Pthreads, you should consider bying the standard from:

```
Publication Sales
IEEE Service Center
P.O. Box 1331
445 Hoes Lane
Piscataway, NJ 08854-1331

phone (201) 981-0060
```

If you need more specific information about certain topics (*e.g., signal handling* or some programming examples), you may want to read the papers available by anonymous ftp from `ftp.cs.fsu.edu` (128.186.121.2) in `/pub/PART` [3, 4, 1] or check out the page `http://www.cs.fsu.edu/~mueller` under WWW.

If not specified otherwise, all functions return 0 on success and -1 on error with `errno` set to some meaningful value (see header file `errno.h`).

2 Pthreads Initialization

```
void pthread_init                ();
```

There is an internal initialization routine, which has to execute before any other call to the Pthreads interface is made. It is suggested that this call be made as the first statement in `main`. Under SunOS and Solaris, the initialization is transparent to the user, *i.e.* it takes place of the initialization code executed prior to `main`. Under all other operating systems, the initialization has to be performed excliplitly by the user in `main`.

3 Threads Management

```
int pthread_attr_init             (pthread_attr_t *__attr);
int pthread_attr_destroy          (pthread_attr_t *__attr);
int pthread_attr_setstacksize     (pthread_attr_t *__attr, size_t __stacksize);
int pthread_attr_getstacksize     (pthread_attr_t *__attr, size_t *__stacksize);
int pthread_attr_setdetachstate   (pthread_attr_t *__attr, int *__detachstate);
int pthread_attr_getdetachstate   (pthread_attr_t *__attr);
```

These functions control the attributes that can be attached to a thread upon creation. The functions provide initialization with the default attributes, destruction, and querying (through pointers) and updating of the stack size and the detach state. The detach state determines if a thread will be enter a detached state upon creation (0 for FALSE, 1 for TRUE). If a thread is in a detached state during termination, the memory will be reclaimed.

```
int pthread_create                (pthread_t *__thread,
                                   pthread_attr_t *__attr,
                                   pthread_func_t __func,
                                   any_t __arg);
```

This function creates an independent flow of control (concurrent or parallel thread of execution) with attributes `attr`. The thread may start immediately by executing the specified function `func` with parameters `arg`. The thread identifier is stored in `thread`.

```
int pthread_join                  (pthread_t __thread, any_t *__status);
```

The current thread waits for the specified thread to terminate and detach.m (Since it detaches a thread implicitly, the behavior calling `pthread_detach` after `pthread_join` is unspecified.) The return value of the specified thread is provided in `status`.

```
int pthread_detach                (pthread_t *__thread);
```

The specified `thread` is detached and any memory associated with it may be recovered once the thread terminates.

```
void pthread_exit                 (any_t __status);
```

The current thread is terminated. If cleanup handlers are pending, they are executed in LIFO order (stack). If destructor functions are defined, they are also executed.

```
pthread_t pthread_self            (void);
```

Returns the thread id of the current thread.

```
int pthread_equal                 (pthread_t __t1, pthread_t __t2);
```

Returns true if both threads are the same, false otherwise.

```
int pthread_once                  (pthread_once_t *__once_c,
                                   void (*__func)(any_t __value));
```

Calls function `func` without parameter for the first thread who calls this routine with this function. Any subsequent calls with the same function have no effect.

4 Synchronization

```
int pthread_mutexattr_init        (pthread_mutexattr_t *__attr);
int pthread_mutexattr_destroy     (pthread_mutexattr_t *__attr);
int pthread_mutexattr_getpshared (pthread_mutexattr_t *__attr, int *__pshared);
int pthread_mutexattr_setpshared (pthread_mutexattr_t *__attr, int __pshared);
```

Mutex attribute initialization (w/ default attributes) and destruction, and set/query the shared state of a mutex across processes. Mutexes shared across processes are currently not supported.

```
int pthread_mutex_init            (pthread_mutex_t *__mutex,
                                   pthread_mutexattr_t *__attr);
int pthread_mutex_destroy         (pthread_mutex_t *__mutex);
```

Mutex initialization (w/ default attributes if `attr` is NULL) and destruction.

```
int pthread_mutex_lock            (pthread_mutex_t *__mutex);
int pthread_mutex_trylock         (pthread_mutex_t *__mutex);
int pthread_mutex_unlock          (pthread_mutex_t *__mutex);
```

These primitives allow for mutual exclusion by locking, trying to lock, and unlocking a mutex. A mutex can only be locked (owned) by one thread at a time. Any subsequent attempt to lock the same mutex by a second thread causes the second thread to suspend in the priority-ordered wait queue of the mutex. When a thread unlocks a mutex it owns, the thread on the head of the wait queue of this mutex may resume execution and lock the mutex. If a mutex is already locked, trylock returns -1 and provides the errno EBUSY.

```
int pthread_condattr_init        (pthread_condattr_t *__attr);
int pthread_condattr_destroy     (pthread_condattr_t *__attr);
int pthread_condattr_getpshared  (pthread_condattr_t *__attr, int *__pshared);
int pthread_condattr_setpshared  (pthread_condattr_t *__attr, int __pshared);
int pthread_cond_init            (pthread_cond_t *__cond,
                                  pthread_condattr_t *__attr);
int pthread_cond_destroy         (pthread_cond_t *__cond);
```

Same as mutex attributes, but for condition variables.

```
int pthread_cond_signal          (pthread_cond_t *__cond);
int pthread_cond_broadcast       (pthread_cond_t *__cond);
```

Signals the thread at the head of the priority-ordered wait queue for condition variables to wake up. Broadcast signals all threads in the queue.

```
int pthread_cond_wait            (pthread_cond_t *__cond,
                                  pthread_mutex_t *__mutex);
int pthread_cond_timedwait       (pthread_cond_t *__cond,
                                  pthread_mutex_t *__mutex,
                                  struct timespec *__timeout);
```

Condition variables should be used for indefinite suspension. In contrast, a mutex should be used for short code sequences to guarantee mutual exclusion, but a thread should never suspend while locking a mutex since all other threads blocked on that mutex remain suspended as well.

These primitives suspend the current thread on the condition variable associated with a mutex until signaled (see above). An optional timeout value may limit the time of suspension. The timeout value is an absolute time value (see gettimeofday). A timeout is indicated by returning -1 and the errno EAGAIN.

The mutex must be locked by the thread before calling any of the two primitives. The thread then atomically suspends and unlocks the mutex. Upon interruption or signaling the condition variable, the thread(s) first atomically wake(s) up and reacquires the mutex, then return from the conditional wait. (Notice that the mutex is thereby locked inside a signal handler if the suspension is interrupted!)

A return from a conditional wait is no guarantee that the condition variable was signaled (due to interrupts and spurious wakeups allowed by the semantics of signaling a condition variable). Thus, after waking up, it becomes necessary to check some condition indicating if the wake-up had been intended. This is typically accomplished by checking condition based a global variable shared by the threads repeatedly until the condition is satisfied.

Example:

```
Thread 1:                            | Thread 2:
                                     |
pthread_mutex_lock(&mutex);          | pthread_mutex_lock(&mutex);
...                                  | ready = TRUE;
while (!ready)                       | pthread_mutex_unlock(&mutex);
  pthread_cond_wait(&cond, &mutex);  | pthread_cond_signal(&cond);
pthread_mutex_unlock(&mutex);        |
```

5 Thread-Specific Data

```
int pthread_key_create          (pthread_key_t *__key,
                                 void (*__func)(any_t __value));
```

Returns a new `key` with which thread-specific data may be associated later. The key is shared among threads while the data is not. Optionally, a destructor function can be specified that will be called with the thread-specific data as parameter upon termination of the thread if a non-NULL data value is associated with this thread at this time.

```
int pthread_setspecific         (pthread_key_t __key, any_t __value);
int pthread_getspecific         (pthread_key_t __key, any_t *__value);
```

Sets/queries the thread-specific data value associated with the key and the current thread.

6 Thread Priority Scheduling

```
int pthread_attr_setscope       (pthread_attr_t *__attr,
                                 int __contentionscope);
int pthread_attr_getscope       (pthread_attr_t __attr);
int pthread_attr_setinheritsched (pthread_attr_t *__attr, int __inherit);
int pthread_attr_getinheritsched (pthread_attr_t *__attr);
int pthread_attr_setsched       (pthread_attr_t *__attr, int __scheduler);
int pthread_attr_getsched       (pthread_attr_t *__attr);
int pthread_attr_setprio        (pthread_attr_t *__attr, int __priority);
int pthread_attr_getprio        (pthread_attr_t *__attr);
```

Set/query thread attribute for contention scope, inherit scheduling, scheduling, and priority. These attributes are used upon thread creation.

The contention scope PTHREAD_SCOPE_GLOBAL specifies that a thread contends with all threads in the system (across processes and processors), PTHREAD_SCOPE_LOCAL specified thread contention contention only within the current process. Only the latter is supported by this implementation.

The values PTHREAD_INHERIT_SCHED may be used to inherit the scheduling parameters during thread creation while PTHREAD_DEFAULT_SCHED indicates that the default attributes are used.

The scheduler options are SCHED_FIFO for strict (preemptive) priority scheduling. Under this policy, a priority-ordered ready queue is maintained. Within one priority range, threads are ordered FIFO. The thread at the head of the ready queue is always executing (highest-priority longest waiting thread). This policy is enforced when interrupts occur or when priorities are changed dynamically with immediate effect, i.e., a high-priority thread awakened by an interrupt preempts a lower priority thread currently executing. An alternative scheduling policy is SCHED_RR, which adds a round-robin time-slicing to the SCHED_FIFO policy. Under SCHED_RR, the current thread is preempted after his time slice expires (if he is still executing). The preempted thread is then requeued at the tail of his priority level in the ready queue (also see [4]). The third policy, SCHED_OTHER, is implementation defined and not supported by this implementation.

The last primitives allow the manipulation of the priorities of a thread attribute.

```
int pthread_getschedattr        (pthread_t __thread, pthread_attr_t *__attrs);
int pthread_setschedattr        (pthread_t __thread, pthread_attr_t __attrs);
```

Sets/queries the attributes of an existing thread.

```
void pthread_yield              (any_t __arg);
```

Causes the current thread to suspend execution and requeue itself at the tail of the its priority level in the ready queue. The argument is always NULL.

```
int setprio                        (pid_t __pid, int __prio);
int getprio                        (pid_t __pid);
int setscheduler                   (pid_t __pid, int __sched, int __prio);
int getscheduler                   (pid_t __pid);
```

Same as same primitives named `pthread_xxx` but applies to the process `pid`. Not supported by this implementation.

```
int pthread_mutexattr_setprotocol     (pthread_mutexattr_t *__attr,
                                       pthread_protocol_t __protocol);
pthread_protocol_t
    pthread_mutexattr_getprotocol     (pthread_mutexattr_t __attr);
int pthread_mutexattr_setprio_ceiling (pthread_mutexattr_t *__attr,
                                       int __prio_ceiling);
int pthread_mutexattr_getprio_ceiling (pthread_mutexattr_t __attr);
```

The attribute of a mutex can be manipulated w.r.t. the protocol, which can be NO_PRIO_INHERIT (default), PRIO_INHERIT (priority inheritance), or PRIO_PROTECT (priority ceiling). The priority ceiling level of a mutex may be set/queried with the latter primitives. The current implementation supports priority ceilings only.

The priority inheritance protocol specifies that the effective priority of a thread holding a mutex shall be the level of the highest priority of all threads contending (waiting) for the mutex. When the current thread releases the mutex, its priority will be lowered again. (Notice that mutex locking may be nested and priority inheritance is transitive in this case, i.e. after unlocking the inner mutex, the priority of the current thread is lowered according to the above rule but with respect to the outer mutex. Notice further that this policy adjusts dynamically by boosting the priority of a thread holding the mutex whenever a new thread contends for the mutex with a higher effective priority.)

The priority ceiling protocol specifies that the effective priority of a thread locking this mutex shall be raised to the maximum of his current priority and the priority ceiling of the mutex. Upon unlocking, the priority level shall be lowered to the level before the locking primitive took effect. The priority ceiling of a mutex shall be (at least) the maximum of the base priorities of all threads contenting for the mutex at any time. (Notice that this policy requires static determination of the priority ceiling level and assumes that the base priorities of the threads contenting as well as the set of threads ever locking the mutex are known a priori. Furthermore, implementations can use stacks and should be more efficient that priority inheritance.) A more detailed description of the two protocols is given in [4].

The current implementation of the priority ceiling protocol is based on the stack resource policy [2]. Under this policy, mutexes have to unlocked in the opposite order in which they were locked (LIFO like a stack).

```
int pthread_mutex_setprio_ceiling (pthread_mutex_t *__mutex,
                                   int __prio_ceiling);
int pthread_mutex_getprio_ceiling (pthread_mutex_t __mutex);
```

These primitives allow the direct manipulation of the priority ceiling level of a mutex. Not supported by this implementation.

7 Process Control

```
pid_t fork (void);
```

Create a new process with only the current thread. Not supported by this implementation.

```
int execl                          (char *__path, char *__arg0, char *__arg1, ...,
                                    char *__argn, (char *) 0);
```

```
int execv                (char *__path, char *__argv[]);
int execle               (char *__path, char *__arg0, char *__arg1, ...,
                          char *__argn, (char *) 0);
int execve               (char *__path, char *__argv[], chat *__envp[]);
int execlp               (char *__file, char *__path,  char *__arg0,
                          char *__arg1, ..., char *__argn, (char *) 0);
int execve               (char *__file, char *__argv[]);
```

Same as in POSIX.1 except that all threads are terminated before the new process is executed. Not supplied by the threads implementation but supported via the regular operating system.

```
void _exit               (int __status);
```

Terminates process and threads, does not call cleanup handlers or destructor functions. Not supplied by the threads implementation but supported via the regular operating system.

```
int wait                 (int *__stat_loc);
int waitpid              (pid_t *__pid, int *__stat_loc, int __options);
```

Same as in POSIX.1 except only the calling thread is suspended. Not supported by this implementation.

8 Signals

Each thread has his own signal mask. Signal handlers are shared among threads of a process. If more than one thread is eligible to receive a certain signal, the recipient may be chosen arbitrarily. Signals may pend on a thread (when the thread has the signal masked) or on the process (when all threads have the signal masked). Default actions (if applicable) are performed with respect to the process (also see [4]).

```
int sigwait              (sigset_t *__set);
```

Atomically clears the signals defined in `set` from the thread's signal mask. If one of the signals in `set` is pending, sigwait return immediately. Otherwise the current thread suspends until a signal in `set` is received. Upon return, the signals in `set` are added to the thread's signal mask again, and the received signal number is returned. (Notice that use of sigwait and sigaction on the same signal is undefined.)

```
int sigprocmask          (int __how, const sigset_t *__set, sigset_t *__oset);
int sigpending           (sigset_t *__set);
int sigsuspend           (const sigset_t *__set);
int pause                (void);
int sigaction            (int __sig,
                          const struct sigaction *__act,
                          struct sigaction *__oact);
```

Same as in POSIX.1 except thread masks are maintained for each thread. Thus, primitives manipulating the signal mask affect the current thread only. Suspending primitives only suspend the current thread. Interrupt handlers[1] are shared by all threads. An interrupt handler receives one/three arguments (if `SA_SIGINFO` is cleared/set in the `sa_flags` field of the installed signal action, as defined in POSIX.4):

[1] The common UNIX term is signal handler. But this may be confusing in the context of signalling condition variables. Thus, UNIX signal handlers will be referred to as interrupt handlers.

```
void interrupt_handler          (int signo, siginfo_t *info, void *context);

struct siginfo {
  int si_signo;
  int si_code;
  union sigval si_value;
};

struct context_t {
  int sc_onstack; /* ignored */
  int sc_mask;    /* per-thread signal mask to be restored */
  int sc_sp;      /* stack pointer to be restored */
  int sc_pc;      /* program counter to be restored */
  int sc_npc;     /* next pc, only used if _sigtramp present on thread's stack,
                   * ignored o.w.; should usually be sc_pc+4 */
};
```

where siginfo contains the signal number, the code of the cause for the signal (as defined in the header file `sys/signal.h`), and an implementation defined signal value that is unused as of now. The context parameter, also implementation-defined, is a pointer to the above context structure that should be self-explanatory.

```
void longjmp                    (struct jmp_buf *__env, int __val);
void siglongjmp                 (struct jmp_buf *__env, int __val);
int setjmp                      (struct jmp_buf *__env);
int sigsetjmp                   (struct jmp_buf *__env, int __savemask);
```

Same as in POSIX.1 but only for the current thread. Notice that it is implementation defined whether cancellation handlers (installed after the `setjmp`) are executed when a long jump is performed. This implementation does not call any cancellation handlers on a long jump. This implementation handles long jumps out of signal handlers unless the signal handler executes on a separate signal stack. The execution of user signal handlers on a separate stack in strongly discouraged since it results in defined behavior. The global signal stack state of the process is not restored properly over a long jump out of a signal handler on a signal stack (BSD UNIX versions only).

```
unsigned int alarm              (unsigned int __seconds);
```

Sends the signal SIGALRM to the current thread after `seconds`. Not supported by this implementation.

```
unsigned int sleep              (unsigned int __seconds);
```

Suspends the current thread for `seconds`.

```
int raise                       (int __sig);
```

Send the signal `sig` to the current process.

```
int pthread_kill                (pthread_t __thread, int __sig);
```

Send the signal `sig` to the specified thread.

9 Thread Cancellation

```
int pthread_cancel              (pthread_t __thread);
```

Terminate the specified thread, i.e. the current execution is aborted and the cleanup handlers and destructor functions are executed before the thread seizes stops to exist. The effect of cancellation is determined by the interruptibility state (see below).

```
int pthread_setintr             (int __state);
int pthread_setintrtype         (int __type);
void pthread_testintr           (void);
```

Set interruptibility and type or create an interruption point.

Interruptibility states are PTHREAD_INTR_ENABLE to allow cancellation and PTHREAD_INTR_-DISABLE to prevent cancellation.

Interruptibility types are PTHREAD_INTR_CONTROLLED and PTHREAD_INTR_ASYNCHRONOUS. In the former case, cancellation can only take effect at interruption points, in the latter case at any time. Interruption points are any place where are thread would suspend (except for pthread_mutex_lock) and whenever pthread_testintr is called.

```
int pthread_cleanup_push        (void (*__func)(any_t __value), any_t __arg);
int pthread_cleanup_pop         (int execute);
```

Push and pop cleanup routines. When a thread is cancelled or terminates, all pending cleanup routines are executed in LIFO order with the argument specified in the push primitive. The pop primitive removes the last cleanup function from the stack and executes it iff `execute` is non-zero.

Cleanup handlers are only meaningful in the context of controlled interruptibility to guarantee, for example, that a mutex does not remain locked by a cancelled task. The cleanup routine may be used in this case to unlock the mutex. This is especially useful when the cleanup handler is pushed before a condition wait and popped right afterwards such that the mutex is release in the cleanup handler if the thread is cancelled while suspended on the condition wait. (Notice that, by definition of the condition wait, the mutex has to be reacquired before cancellation can take place. This explains the necessity of these primitives.)

Example:

```
Thread 1:                                       | Thread 2:
                                                |
pthread_mutex_lock(&mutex);                     | ...
...                                             | pthread_mutex_lock(&mutex);
pthread_cleanup_push(m_clean, &mutex);          | ready = TRUE;
while (!ready)                                  | pthread_mutex_unlock(&mutex);
  pthread_cond_wait(&cond, &mutex);             | pthread_cancel(thread1);
pthread_cleanup_pop(TRUE); /* pop & unlock */   |
...

void m_clean(mutex)
pthread_mutex_t *mutex;
{
  pthread_mutex_unlock(mutex);
}
```

10 Reentrant Functions

Notice that any libraries may not be thread-safe, e.g. some C library function (stdio, malloc, free, etc.) use global state, which is not protected by mutexes and may therefore become inconsistent in a multi-threaded environment, in particular as a result of preemption. Such functions are non-reentrant, i.e. only one thread may execute them at a time. This implementation does not provide thread-safe libraries. The user is strongly advised to provide explicit mutual exclusion around known non-reentrant functions or to rewrite other library routines in a reentrant fashion.

11 POSIX.4 Primitives

These primitives are adjusted to work with threads:

```
int sched_get_priority_max      (int __policy);
int sched_get_priority_min      (int __policy);
```

Returns the minimum/maximum priorities applicable for the specified scheduling policy.

```
int nanosleep                   (const struct timespec *__rqtp,
                                 struct timespec *__rmtp);
```

Same as `sleep` but with precision of nanoseconds.

```
int clock_gettime               (int __clock_id, struct timespec *__tp);
```

Reads the time of a clock. This implementation only supports CLOCK_REALTIME as a `clock_id`.

12 POSIX.1 Primitives

These primitives are adjusted to work with threads:

```
#ifdef IO
int read                        (int __fd, int __nbytes, char *__buf);
int write                       (int __fd, int __nbytes, char *__buf);
#endif
```

If the compile flag IO is defined, these functions are the same as in POSIX.1 except that they only block the current thread, not any other thread. The blocked thread becomes unblocked when the signal corresponding to the I/O request is received. If the compile flag IO is not defined, these functions act exactly as defined in POSIX.1 and block the whole process.

13 Implementation-Defined Primitives

```
int pthread_attr_getstarttime_np(pthread_attr_t *__attr, struct timespec *__tp);
int pthread_attr_setstarttime_np(pthread_attr_t *__attr, struct timespec *__tp);
int pthread_attr_getdeadline_np (pthread_attr_t *__attr, struct timespec *__tp);
int pthread_attr_setdeadline_np (pthread_attr_t *__attr, struct timespec *__tp,
                                 void *(*__user_handler)(void *));
int pthread_attr_getperiod_np   (pthread_attr_t *__attr, struct timespec *__tp);
int pthread_attr_setperiod_np   (pthread_attr_t *__attr, struct timespec *__tp,
                                 void *(*__user_handler)(void *));
```

The scheduling attributes of a thread are extended to allow the specification of an absolute start time, an absolute deadline, and a relative period. These attributes have to be set before passed as an argument for thread creation. They cannot be changed dynamically while the thread is already running but the value can be inquired at any time.

The semantics for these deadline-scheduled threads is as follows: A thread with a start time begins to execute no earlier than its start time. A thread with a period (and a required start time) is restarted when when its prior start time plus its period is reached. If the thread does not complete its execution within its period, a user-defined handler is invoked at the end of its period before it is restarted. For a thread with a deadline (and a required start time) this user handler is invoked when its deadline is reached if it has not yet completed its execution. The user handler executes as a signal handler. It is commonly used to lower the priority of a thread or to terminate the thread via a call to `pthread_exit()`.

```
int pthread_setprio_np            (pthread_t __thread, int prio);
```

Sets the priority of an existing thread. This interface is NOT part of draft 6 anymore but is supported by the implementation if the compilation flag DRAFT5 is set. It provides a fast interface to set the priority of a thread.

```
extern void pthread_setsigcontext_np(struct context_t *__scp, jmp_buf __env, int __val);
```

Modifies the context structure of an interrupt handler to simulate a `longjmp(env, val)` out of the handler, *i.e.* control is transferred after the handler returns to the last `setjmp(env)` call with a return value `val` instead of continuing the execution at the interruption point. This facility should be used instead of a `longjmp(env, val)` out of an interrupt handler to ensure portability. An example is given below.

```
int pthread_lock_stack_np        (pthread_t __p);
```

Reenables the signal-driven stack check after a stack overflow has occurred for thread `p`. Thus routine must not be called within the user-defined signal handler that handles stack overflow.

If the compile flags STACK_CHECK and SIGNAL_STACK are defined, a stack overflow causes the delivery of either a SIGILL signal with code ILL_STACK or a SIGBUS signal with code FC_OBJERR to a user-defined signal handler routine. The user signal handler is limited to use no more stack space than one page (4Kb). It should perform some cleanup and transfer control out of the user signal handler by manipulating the context structure (NOT by a `longjmp`). After the transfer of control, the signal-driven stack check has to be reenabled by calling the above routine (at the point where `setjmp` returns a non-zero value).

Example:

```
jmp_buf env;

void new_thread(arg)
  int *arg;
{
  ...
  if (setjmp(env)) /* return here after stack overflow */
    pthread_lock_stack_np(pthread_self());
  ...
}

/*
 * stack overflow handler, transfers control to corresponding setjmp()
 * by mimicing a longjmp(env, TRUE) except that we return through the
 * signal catching frames, thus unwinding the stack properly.
 * Under SunOS 4.1.x, longjmp(env, TRUE) works but under Solaris 2.x,
 * it does not since the process is made to believe it is still executing
 * on the alternate signal stack. The code below works for both SunOS 4.1.x
 * and Solaris 2.x.
 */
void stack_overflow_handler(sig, sip, scp)
     struct siginfo *sip;
     int sig;
     struct context_t *scp;
{
  pthread_setsigcontext_np(scp, env, TRUE); /* TRUE is return value for setjmp() */
}

main()
{
  struct sigaction act;

  pthread_init();

  act.sa_handler = stack_overflow_handler;
  sigemptyset(&act.sa_mask);
  act.sa_flags = SA_SIGINFO;
  if (sigaction(SIGBUS, &act, (struct sigaction *) NULL))
    printf("ERROR: sigaction error\n");
  if (sigaction(SIGILL, &act, (struct sigaction *) NULL))
    printf("ERROR: sigaction error\n");
  if (sigaction(SIGSEGV, &act, (struct sigaction *) NULL))
    printf("ERROR: sigaction error\n");
  ...
}
```

References

[1] T. P. Baker, F. Mueller, and Viresh Rustagi. Experience with a prototype of the POSIX "minimal realtime system profile". In *IEEE Workshop on Real-Time Operating Systems and Software*, pages 12–16, 1994.

[2] T.P. Baker. Stack-based scheduling of realtime processes. *Real-Time Systems*, 3(1):67–99, March 1991.

[3] Frank Mueller. Implementing POSIX threads under UNIX: Description of work in progress. In *Proceedings of the Second Software Engineering Research Forum*, pages 253–261, November 1992.

[4] Frank Mueller. A library implementation of POSIX threads under UNIX. In *Proceedings of the USENIX Conference*, pages 29–41, January 1993.