The PORTS0 Interface

The PORTS Consortium

Version 0.3
January 27, 1995

The PORTS (POrtable RunTime System) group was established to address the problems of constructing a common runtime system to be used as a compiler target for various task- and data-parallel languages. One result of this group's efforts is the definition of an applications programming interface, the PORTS level-zero interface (PORTS0). This interface comprises lightweight thread functions and a core set of reentrant library routines. This report describes the PORTS0 interface.

Contents

1 Introduction **1**

2 Initialization and Shutdown **1**
2.1 `ports0_init()` . . . 1
2.2 `ports0_shutdown()` . . . 2

3 Thread Management **2**
3.1 Thread Control . . . 2
3.1.1 `ports0_threadattr_init()` . . . 3
3.1.2 `ports0_threadattr_destroy()` . . . 3
3.1.3 `ports0_threadattr_setstacksize()` . . . 3
3.1.4 `ports0_threadattr_getstacksize()` . . . 3
3.1.5 `ports0_thread_create()` . . . 4
3.1.6 `ports0_thread_exit()` . . . 4
3.1.7 `ports0_thread_yield()` . . . 4
3.1.8 `ports0_thread_self()` . . . 4
3.1.9 `ports0_thread_equal()` . . . 4
3.1.10 `ports0_thread_once()` . . . 5
3.2 Thread-specific Data . . . 5
3.2.1 `ports0_thread_key_create()` . . . 5
3.2.2 `ports0_thread_key_delete()` . . . 6
3.2.3 `ports0_thread_setspecific()` . . . 6
3.2.4 `ports0_thread_getspecific()` . . . 6
3.3 Mutual Exclusion and Synchronization . . . 6
3.3.1 `ports0_mutexattr_init()` . . . 7
3.3.2 `ports0_mutexattr_destroy()` . . . 7
3.3.3 `ports0_mutex_init()` . . . 7
3.3.4 `ports0_mutex_destroy()` . . . 8
3.3.5 `ports0_mutex_lock()` . . . 8
3.3.6 `ports0_mutex_trylock()` . . . 8
3.3.7 `ports0_mutex_unlock()` . . . 8
3.3.8 `ports0_condattr_init()` . . . 8
3.3.9 `ports0_condattr_destroy()` . . . 8
3.3.10 `ports0_cond_init()` . . . 9
3.3.11 `ports0_cond_destroy()` . . . 9
3.3.12 `ports0_cond_wait()` . . . 9
3.3.13 `ports0_cond_signal()` . . . 9
3.3.14 `ports0_cond_broadcast()` . . . 9

4 Reentrant Library **10**
4.1 ports0_malloc() 10
4.2 ports0_realloc() 10
4.3 ports0_calloc() 10
4.4 ports0_free() 10
4.5 ports0_open() 11
4.6 ports0_close() 11
4.7 ports0_read() 11
4.8 ports0_write() 11
4.9 ports0_lseek() 11
4.10 ports0_fstat() 12
4.11 ports0_reentrant_lock() 12
4.12 ports0_reentrant_unlock() 12

1 Introduction

The PORTS level-zero interface (called PORTS0) comprises a set of functions for lightweight thread management and for reentrant memory management and file I/O. The thread management routines are modeled after a subset of the POSIX thread interface, while the reentrant functions are modeled after standard C library routines. This set of functions is the first set of functions to be agreed upon by the PORTS (POrtable RunTime System) group as necessary parts of a complete runtime system for parallel languages and tools.

This document briefly describes the PORTS0 interface. More information on the PORTS group can be found on the World Wide Web at URL `http://www.cs.uoregon.edu/paracomp/ports`. An implementation of the PORTS0 interface, which was jointly developed at Argonne National Laboratory and California Institute of Technology, is available by anonymous ftp from `ftp://ftp.mcs.anl.gov/pub/ports`.

2 Initialization and Shutdown

PORTS0 provides an initialization and a shutdown routine. These functions should be called before and after any other PORTS0 routines are used, respectively.

Any program that uses PORTS0 functions must include the file "`ports0.h`". This file contains functions prototypes and symbol definitions for the PORTS0 interface.

2.1 `ports0_init()`

```
int ports0_init(int *argc,
                char **argv[],
                char *package_id)
```

Initialize the PORTS0 library. This function must be called before any other PORTS0 functions are called.

The value zero is returned if initialization is successful, and nonzero is returned if initialization fails.

The arguments in `argc` and `argv` are scanned by `ports_init()`, and any arguments that are recognized by PORTS0 will be removed by modifying `argc` and `argv`.

The `package_id` string is used as a prefix to all PORTS0 arguments. For example, if `package_id` is "foo", then each PORTS arguments either will begin with "-foo_" or will be positioned between "-foo" and "-foo_end" arguments.

Hence, an argument to specify the default stack size can be specified either as "-foo_stack 1024" or "-foo -stack 1024 -foo_end".

While all PORTS arguments will have a `package_id` prefix, this does not imply that all arguments prefixed by `package_id` are PORTS0 arguments. Some of them may be for the system that is using PORTS0. Therefore, not all arguments with a `package_id` prefix will necessarily be removed from `argc` and `argv` when `ports0_init()` returns. Only those arguments that are actually recognized by PORTS0 will be removed. The `package_id` simply allows PORTS0 and the system using it to have consistent argument naming.

2.2 `ports0_shutdown()`

```
int ports0_start()
```

Shut down the PORTS0 library. This function must be called before the program using PORTS0 terminates. No other PORTS0 functions may be called after `ports0_shutdown()`.

Return zero if shutdown is successful, otherwise nonzero.

3 Thread Management

PORTS0 threads are modeled after a subset of POSIX threads (IEEE standard P1003.4a draft 8). The semantics of the PORTS0 routines are identical to the corresponding POSIX thread routines, unless otherwise noted. This section briefly describes each of the PORTS0 thread routines. Please refer to the POSIX thread standard for more details.

3.1 Thread Control

PORTS0 provides the following functions for basic thread control:

- `ports0_threadattr_init`: initialize a thread attribute
- `ports0_threadattr_destory`: destroy a thread attribute
- `ports0_threadattr_setstacksize`: set the stack size in a thread attribute
- `ports0_threadattr_getstacksize`: get the stack size from a thread attribute
- `ports0_thread_create`: create a thread
- `ports0_thread_exit`: terminate the current thread

- `ports0_thread_yield`: yield the processor to another thread
- `ports0_thread_self`: return the thread ID of the calling thread
- `ports0_thread_equal`: compare two thread IDs
- `ports0_thread_once`: for dynamic module initialization

3.1.1 `ports0_threadattr_init()`

```
int ports0_threadattr_init(ports0_threadattr_t *attr)
```

Initialize `attr` to have the default thread attributes.
Return zero upon successful completion, otherwise nonzero.

3.1.2 `ports0_threadattr_destroy()`

```
int ports0_threadattr_destroy(ports0_threadattr_t *attr)
```

Destroy the thread attributes object, `attr`.
Return zero upon successful completion, otherwise nonzero.

3.1.3 `ports0_threadattr_setstacksize()`

```
int ports0_threadattr_setstacksize(ports0_threadattr_t *attr,
                                   size_t stacksize)
```

Set the stack size value in the thread attributes object, `attr`.
Return zero upon successful completion, otherwise nonzero.

3.1.4 `ports0_threadattr_getstacksize()`

```
int ports0_threadattr_getstacksize(ports0_threadattr_t *attr,
                                   size_t *stacksize)
```

Get the stack size value from the thread attributes object, `attr`, and place it into the address pointed to by `stacksize`.
Return zero upon successful completion, otherwise nonzero.

3.1.5 ports0_thread_create()

```
typedef void *(*ports0_thread_func_t)(void *user_arg);

int ports0_thread_create(ports0_thread_t *thread,
                         ports0_threadattr_t *attr,
                         void *(*func)(void *),
                         void *user_arg)
```

Create a new thread that invokes the supplied function `func` with one argument `user_arg`. The thread ID for the newly created thread is placed in `thread`. The `attr` argument specifies the attributes for the thread. Default attributes will be used if `attr` is `NULL`.

Return zero if successful, otherwise nonzero.

Note: There is no equivalent to `pthread_join()` in PORTS0. All PORTS0 threads are automatically detached when they are created. This is a departure from POSIX thread semantics.

3.1.6 ports0_thread_exit()

```
void ports0_thread_exit(void *status)
```

Terminate the calling thread. Returning from the user thread function will implicitly terminate the thread.

Note: The `status` argument is not used, since PORTS0 does not support an equivalent to `pthread_join()`.

3.1.7 ports0_thread_yield()

```
void ports0_thread_yield()
```

Yield the processor to another thread.

3.1.8 ports0_thread_self()

```
ports0_thread_t ports0_thread_self()
```

Return the thread ID of the calling thread.

3.1.9 ports0_thread_equal()

```
int ports0_thread_equal(ports0_thread_t t1,
                        ports0_thread_t t2)
```

Compare the two thread IDs `t1` and `t2`.

Return nonzero if the threads are the same, otherwise zero.

3.1.10 ports0_thread_once()

```
ports0_thread_once_t once_control = PORTS0_THREAD_ONCE_INIT;

int ports0_thread_once(ports0_thread_once_t *once_control,
                       void (*init_routine)() )
```

The first call to `ports0_thread_once()` by any thread in a process, with a given `once_control`, will result in a call to the supplied `init_routine()` with no arguments. Subsequent calls to `ports0_thread_once()` will not call the `init_routine()`. On return of `ports0_thread_once()` it is guaranteed that `init_routine()` has completed. The `once_control` parameter is used to determine whether the associated initialization routine has been called.

Return zero upon successful completion, otherwise nonzero.

3.2 Thread-specific Data

PORTS0 provides the following functions for thread-specific data:

- `ports0_thread_key_create`: create a thread-specific data key
- `ports0_thread_key_delete`: delete a thread-specific data key
- `ports0_thread_setspecific`: associate a value with a thread-specific data key
- `ports0_thread_getspecific`: retrieve the value associated with a thread-specific data key

3.2.1 ports0_thread_key_create()

```
typedef void (*ports0_thread_key_destructor_func_t)(void *value);

int ports0_thread_key_create(
               ports0_thread_key_t *key,
               void (*destructor_func)(void*))
```

Create a thread-specific data key that is visible to all threads in the process, and place that key in the `key` argument.

Although the same key may be used by different threads, the values bound to the key by `ports0_thread_setspecific()` are maintained on a per-thread basis. The value associated with a new key is `NULL` in all active threads and will be initialized to `NULL` in all threads that are subsequently created. If `destructor_func` is not `NULL`, then upon termination of the thread if the

value for this key is not NULL, the function pointed to by destructor_func is called with the current value for the key as its argument.

Return zero upon successful completion, otherwise nonzero. A return of EAGAIN indicates that the key name space is exhausted.

3.2.2 ports0_thread_key_delete()

```
int ports0_thread_key_delete(ports0_thread_key_t key)
```

Delete the thread-specific data key.

The destructor fuction associated with this key is *not* called. Subsequent use of this key will result in undefined behavior.

Return zero upon successful completion, otherwise nonzero.

3.2.3 ports0_thread_setspecific()

```
int ports0_thread_setspecific(ports0_thread_key_t key,
                              void *value)
```

Set the value associated with the thread-specific data key to value.

Different threads may bind different values to the same key.

Return zero upon successful completion, otherwise nonzero.

3.2.4 ports0_thread_getspecific()

```
int ports0_thread_getspecific(ports0_thread_key_t key,
                              void **value)
```

Get the thread-specific data value associated with key, and return it in the value argument.

Return zero upon successful completion, otherwise nonzero.

3.3 Mutual Exclusion and Synchronization

Mutual exclusion and synchronization between threads are provided by the following operations:

- ports0_mutexattr_init: initialize a mutex attribute
- ports0_mutexattr_destroy: destroy a mutex attribute
- ports0_mutex_init: initialize a mutual exclusion lock
- ports0_mutex_destroy: destroy a lock
- ports0_mutex_lock: obtain a mutually exclusive access to lock

- `ports0_mutex_trylock`: attempt to obtain a mutually exclusive access to lock
- `ports0_mutex_unlock`: release a lock
- `ports0_condattr_init`: initialize a condition attribute
- `ports0_condattr_destroy`: destroy a condition attribute
- `ports0_cond_init`: initialize a condition variable
- `ports0_cond_destroy`: destroy a condition variable
- `ports0_cond_wait`: wait for a condition
- `ports0_cond_signal`: signal a condition
- `ports0_cond_broadcast`: signal to all waiting for a condition

3.3.1 `ports0_mutexattr_init()`

```
int ports0_mutexattr_init(ports0_mutexattr_t *attr)
```

Initialize `attr` to have the default mutex attributes.
Return zero upon successful completion, otherwise nonzero.

3.3.2 `ports0_mutexattr_destroy()`

```
int ports0_mutexattr_destroy(ports0_mutexattr_t *attr)
```

Destroy the mutex attributes object, `attr`.
Return zero upon successful completion, otherwise nonzero.

3.3.3 `ports0_mutex_init()`

```
int ports0_mutex_init(ports0_mutex_t *mutex,
                      ports0_mutexattr_t *attr)
```

Initialize the mutual exclusion lock, `mutex`.

The attributes for the mutex are specified by `attr`. Default attributes will be used if `attr` is `NULL`. The result of calling `ports0_mutex_lock()` or `ports0_mutex_unlock()` on a mutex that has not been initialized is undefined.

Return zero upon successful completion, otherwise nonzero.

3.3.4 ports0_mutex_destroy()

```
int ports0_mutex_destroy(ports0_mutex_t *mutex)
```

Destroy the `mutex` that was initialized with `ports0_mutex_init()`.

The result of calling `ports0_mutex_lock()` or `ports0_mutex_unlock()` on a mutex that has been destroyed is undefined.

Return zero upon successful completion, otherwise nonzero.

3.3.5 ports0_mutex_lock()

```
int ports0_mutex_lock(ports0_mutex_t *mutex)
```

Block until the mutual exclusion lock, `mutex`, is acquired.

Return zero upon successful completion, otherwise nonzero.

3.3.6 ports0_mutex_trylock()

```
int ports0_mutex_trylock(ports0_mutex_t *mutex)
```

Attempt to acquire the mutual exclusion lock, `mutex`.

Returen 0 if successful. If `mutex` has already been acquired, then do not acquire the lock, and return EBUSY.

3.3.7 ports0_mutex_unlock()

```
int ports0_mutex_unlock(ports0_mutex_t *mutex)
```

Unlock the mutual exclusion lock, `mutex`, enabling another thread to acquire the mutex.

Fairness in locking is not guaranteed; that is, a thread is not guaranteed to acquire a lock if other threads are also attempting to acquire the same lock.

Return zero upon successful completion, otherwise nonzero.

3.3.8 ports0_condattr_init()

```
int ports0_condattr_init(ports0_condattr_t *attr)
```

Initialize `attr` to have the default condition attributes.

Return zero upon successful completion, otherwise nonzero.

3.3.9 ports0_condattr_destroy()

```
int ports0_condattr_destroy(ports0_condattr_t *attr)
```

Destroy the condition attributes object, `attr`.

Return zero upon successful completion, otherwise nonzero.

3.3.10 ports0_cond_init()

```
int ports0_cond_init(ports0_cond_t *cond,
                     ports0_condattr_t *attr)
```

Initialize the condition variable, `cond`.

The attributes for the condition are specified by `attr`. Default attributes will be used if `attr` is `NULL`. The result of calling any other `ports0_cond_*()` function on a condition that has not been initialized is undefined.

Return zero upon successful completion, otherwise nonzero.

3.3.11 ports0_cond_destroy()

```
int ports0_cond_destroy(ports0_cond_t *cond)
```

Destroy the specified condition.

The result of calling any other `ports0_cond_*()` function on a condition that has been destroyed is undefined.

Return zero upon successful completion, otherwise nonzero.

3.3.12 ports0_cond_wait()

```
int ports0_cond_wait(ports0_cond_t *cond,
                     ports0_mutex_t *mutex)
```

Atomically release `mutex` and wait on `cond`. When the function returns, `mutex` has been reacquired.

If the thread executing the function has not acquired `mutex`, the result is undefined.

Return zero upon successful completion, otherwise nonzero.

3.3.13 ports0_cond_signal()

```
int ports0_cond_signal(ports0_cond_t *cond)
```

Signal the specified condition, waking up one thread that is suspended on this condition.

If no threads are suspended on this condition, this call will have no effect.

Return zero upon successful completion, otherwise nonzero.

3.3.14 ports0_cond_broadcast()

```
void ports0_cond_broadcast(ports0_cond_t *cond)
```

Unsuspend all threads suspended on the specified condition.

Return zero upon successful completion, otherwise nonzero.

4 Reentrant Library

PORTS0 provides a set of functions that mirror standard C library routines. These routines guarantee the following:

- *Reentrancy*: Multiple threads can call these routines without interfering with each other.
- *Nonblocking*: A thread that calls one of these routines is guaranteed to not block other threads from executing.

The following sections briefly describe routines that are provided by PORTS0. Unless otherwise stated, each is identical to the underlying C library routine after which the PORTS0 routine is named.

In addition, PORTS0 exports a lock that may be used to control reentrancy in non-PORTS0 routines. This lock is manipulated by the `ports0_reentrant_lock()` and `ports0_reentrant_unlock()` routines.

4.1 ports0_malloc()

```
void *ports0_malloc(size_t bytes)
```

Allocate memory, like `malloc()`.

4.2 ports0_realloc()

```
void *ports0_reallco(void *ptr, size_t bytes)
```

Reallocate memory, like `realloc()`.

4.3 ports0_calloc()

```
void *ports0_callco(size_t nobj, size_t bytes)
```

Allocate memory, like `calloc()`.

4.4 ports0_free()

```
void ports0_free(void *ptr)
```

Free memory, like `free()`. The memory pointed to by `ptr` must have been previously allocated by `ports0_malloc()`, `ports0_realloc()`, or `ports0_calloc()`.

4.5 ports0_open()

```
int ports0_open(char *path,
                int flags,
                int mode)
```

Open a file descriptor, like `open()`. All file descriptors that are passed to other PORTS0 library routines must be opened using `ports0_open()`.

4.6 ports0_close()

```
int ports0_close(int fd)
```

Close a file, like `close()`.

4.7 ports0_read()

```
int ports0_read(int fd,
                char *buf,
                int nbytes)
```

Read from a file, like `read()`.

Note: Instead of blocking indefinitely, this routine may return an error with `errno` set to EINTR.

4.8 ports0_write()

```
int ports0_write(int fd,
                 char *buf,
                 int nbytes)
```

Write to a file, like `write()`.

Note: Instead of blocking indefinitely, this routine may return an error with `errno` set to EINTR.

4.9 ports0_lseek()

```
int ports0_lseek(int fd,
                 off_t offset,
                 int whence)
```

Move the offset within a file for subsequent reads and writes, like `lseek()`.

4.10 ports0_fstat()

```
int ports0_fstat(int fd,
                 struct stat *buf)
```

Get information about a file, like `fstat()`.

4.11 ports0_reentrant_lock()

```
int ports0_reentrant_lock()
```

Acquire the PORTS0 reentrancy lock.

4.12 ports0_reentrant_unlock()

```
int ports0_reentrant_unlock()
```

Release the PORTS0 reentrancy lock.