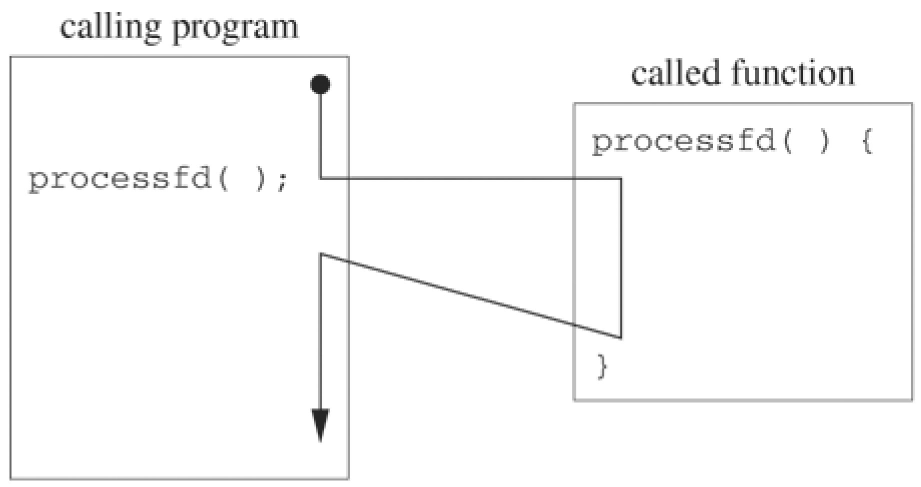
- Thread

Thread

Program that makes an ordinary call to processfd has asingle thread of execution.



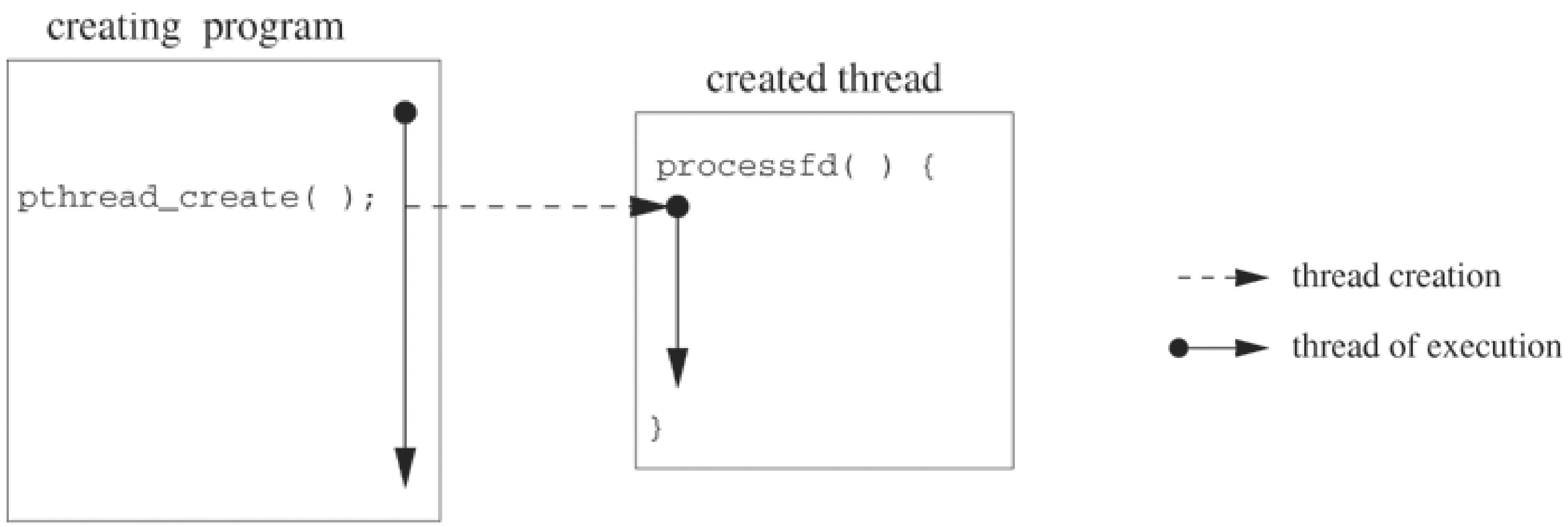
call to the processfd function within the same thread of execution.

The calling mechanism creates an activation record (usually on the stack) that contains the return address.

The thread of execution jumps to processfd when the calling mechanism writes the starting address of processfd in the processor's program counter.

Thread

Program that creates a new thread to execute processfd has two threads of execution



creation of a separate thread to execute the processfd function.

The pthread_create call creates a new "schedulable entity" with its own value of the program counter, its own stack and its own scheduling parameters.

The "schedulable entity" (i.e.,thread) executes an independent stream of instructions, never returning to the point of the call.

The calling program continues to execute concurrently.

Thread

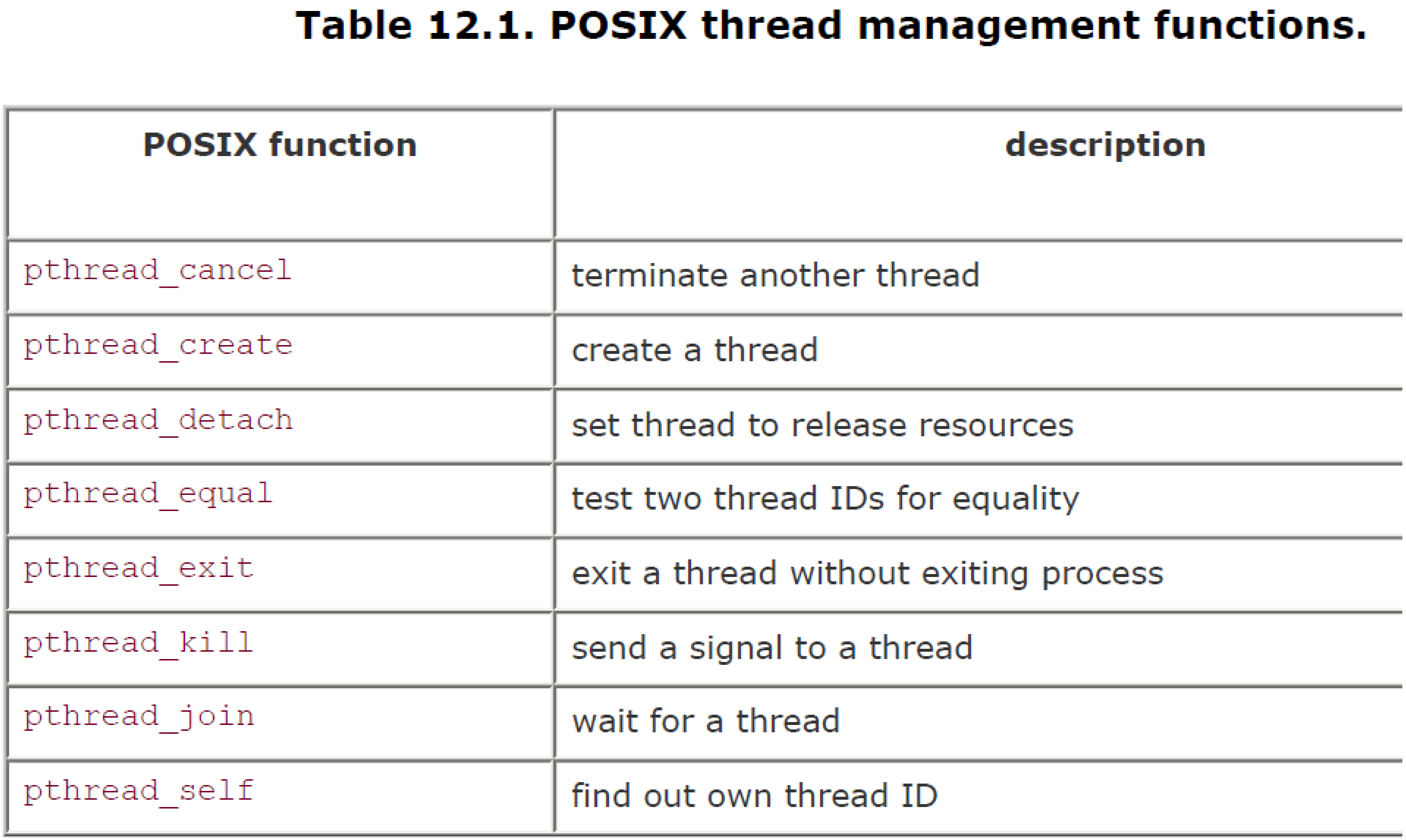


Table 12.1. POSIX thread management functions.

POSIX function	description
`pthread_cancel`	terminate another thread
`pthread_create`	create a thread
`pthread_detach`	set thread to release resources
`pthread_equal`	test two thread IDs for equality
`pthread_exit`	exit a thread without exiting process
`pthread_kill`	send a signal to a thread
`pthread_join`	wait for a thread
`pthread_self`	find out own thread ID

Thread

returns the thread ID of the calling thread.

```
SYNOPSIS
#include <pthread.h>
pthread_t pthread_self(void);
```

compare thread IDs for equality.

If t1 = t2, returns a nonzero value.

If the thread IDs are not equal, returns 0

```
SYNOPSIS
#include <pthread.h>
pthread_t pthread_equal(thread_t t1, pthread_t t2);
```

Creating a thread

pthread_create function creates a thread

```
SYNOPSIS
#include <pthread.h>
int pthread_create(pthread_t *restrict thread, const pthread_attr_t *restrict attr,
void *(*start_routine)(void *), void *restrict arg);
```

thread parameter is the ID of the newly created thread

attr parameter represents an attribute object that contains the attributes of a thread. If attr is NULL, the new thread has the default attributes.

start_routine, is the name of a function that the thread calls when it begins execution.

arg parameter, is a single argument

Detaching and joining

When thread exits, it does not release its resources unless it is detached.

pthread_detach function sets a thread's internal options to specify that storage for the thread can be reclaimed when the thread exits

The pthread_join function causes the caller to wait for the specified thread to exit, similar to waitpid at the process level.

Threads that are not detached are joinable and do not release all their resources until another thread calls pthread_join for them or the entire process exits.

SYNOPSIS
#include <pthread.h>
int pthread_detach(pthread_t thread);
	SYNOPSIS
	#include <pthread.h>
	int pthread_join(pthread_t thread, void **value_ptr);

The pthread_join function suspends the calling thread until the target thread, specified by the first parameter, terminates.

The value_ptr parameter provides a location for a pointer to the return status that the target thread passes to pthread_exit or return.

If value_ptr is NULL, the caller does not retrieve the target thread return status.

Exiting and cancellation

A process can terminate

- by calling exit directly,
- by executing return from main, or
- by having one of the other process threads call exit.

In any of these cases, all threads terminate.

If the main thread has no work to do after creating other threads, it should either block until all threads have completed or call pthread_exit(NULL).

A call to exit causes the entire process to terminate; A call to pthread_exit causes only the calling thread to terminate.

A thread that executes return from its top level implicitly calls pthread_exit with the return value (a pointer) serving as the parameter to pthread_exit.

A process will exit with a return status of 0 if its last thread calls pthread_exit.

```
SYNOPSIS
#include <pthread.h>
void pthread_exit(void *value_ptr);
```

Thread Synchronization

Thread synchronization is defined as a mechanism which ensures that two or more concurrent processes or threads do not simultaneously execute some particular program segment known as critical section.

Processes' access to critical section is controlled by using synchronization techniques.

The synchronization functions that are available in the POSIX:THR Extension are

1. mutex locks
2. Condition variable
3. Read-write locks

The mutex locks and condition variables allow static initialization.

Mutex Locks

- A mutex is a special variable.
- It can be either in the locked state or the unlocked state.
- If the mutex is locked, it has a distinguished thread that holds or owns the mutex.
- If no thread holds the mutex, we say the mutex is unlocked, free or available.
- The mutex also has a queue for the threads that are waiting to hold the mutex.
- The order in which the threads in the mutex queue obtain the mutex is determined by the thread-scheduling policy.
- It is simplest and most efficient thread synchronization mechanism.
- Programs use mutex locks to preserve critical sections and to obtain exclusive access to resources.

Declaration and Initialization:

Declaration and Initialization:

```
SYNOPSIS
#include <pthread.h>
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

For statically allocated pthread_mutex_t variables, assign PTHREAD_MUTEX_INITIALIZER to the mutex variable.

Creating and destroy a mutex

Creation:

```
SYNOPSIS
#include <pthread.h>
int pthread_mutex_init(pthread_mutex_t *restrict mutex, const pthread_mutexattr_t *restrict attr);
```

1st parameter -- is a pointer to the mutex to be initialized.
2nd parameter – is mutex attribute, pass NULL to initialize a mutex with the default attributes

It destroys the mutex referenced by its parameter.

The mutex parameter is a pointer to the mutex to be destroyed.

A pthread_mutex_t variable that has been destroyed with pthread_mutex_destroy can be reinitialized with pthread_mutex_init.

```
SYNOPSIS
#include <pthread.h>
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

Locking and unlocking a mutex

For acquiring a mutex 2 functions:

pthread_mutex_lock and pthread_mutex_trylock

.The pthread_mutex_lock function blocks until the mutex is available, while the pthread_mutex_trylock always returns immediately.

The pthread_mutex_unlock function releases the specified mutex.

All three functions take a single parameter, mutex, a pointer to a mutex.

```
SYNOPSIS
#include <pthread.h>
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

Condition Variables

Ordering of threads:

Assume that two variables, x and y, are shared by multiple threads.
We want a thread to wait until x and y are equal

One incorrect busy-waiting solution is

```
while (x != y) ;
```

Correct strategy for non-busy waiting for the predicate x==y to become true.

1. Lock a mutex.
2. Test the condition x==y.
3. If true, unlock the mutex and exit the loop.
4. If false, suspend the thread and unlock the mutex.

POSIX condition variables provide an atomic waiting mechanism

Declaration and Initialization:

For statically allocated pthread_cond_t variables, assign PTHREAD_COND_INITIALIZER to the cond variable.

```
SYNOPSIS
#include <pthread.h>
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```

For dynamically allocated pthread_cond_t variables, use init function

Creating and destroy

Dynamic initialization:

```
SYNOPSIS
#include <pthread.h>
int pthread_cond_init(pthread_cond_t *restrict cond, const pthread_condattr_t *restrict attr);
```

1st parameter -- is a pointer to the cond to be initialized.
2nd parameter – is attribute, pass NULL to initialize with the default attributes

destroys the condition variable referenced by its cond parameter.

A pthread_cond_t variable that has been destroyed with pthread_cond_destroy can be reinitialized with pthread_cond_init.

```
SYNOPSIS
#include <pthread.h>
int pthread_cond_destroy(pthread_cond_t *cond);
```

Waiting and signaling operations

Wait :

```
SYNOPSIS
#include <pthread.h>
int pthread_cond_wait(pthread_cond_t *restrict cond, pthread_mutex_t *restrict mutex);
```

1st parameter -- is a pointer to the cond to be initialized.
2nd parameter -- is a pointer to the mutex for which it wait.

Signal:

it unblocks at least one of the threads that are blocked on the condition variable

```
SYNOPSIS
#include <pthread.h>
int pthread_cond_signal(pthread_cond_t *cond);
```