Thread Create Attributes 3

The previous chapter covered the basics of threads creation using default attributes. This chapter discusses setting attributes at thread creation time.

Note that only pthreads uses attributes and cancellation, so the API covered in this chapter is for POSIX threads only. Otherwise, the *functionality* for Solaris threads and pthreads is largely the same. (See Chapter 9, "Programming with Solaris Threads" for more information about similarities and differences.)

3

Attributes

Attributes are a way to specify behavior that is different from the default. When a thread is created with `pthread_create(3T)` or when a synchronization variable is initialized, an attribute object can be specified. The defaults are usually sufficient.

An attribute object is opaque, and cannot be directly modified by assignments. A set of functions is provided to initialize, configure, and destroy each object type.

Once an attribute is initialized and configured, it has process-wide scope. The suggested method for using attributes is to configure all required state specifications at one time in the early stages of program execution. The appropriate attribute object can then be referred to as needed.

Using attribute objects has two primary advantages.

- First, it adds to code portability.

  Even though supported attributes might vary between implementations, you need not modify function calls that create thread entities because the attribute object is hidden from the interface.

  If the target port supports attributes that are not found in the current port, provision must be made to manage the new attributes. This is an easy porting task though, because attribute objects need only be initialized once in a well-defined location.

- Second, state specification in an application is simplified.

  As an example, consider that several sets of threads might exist within a process, each providing a separate service, and each with its own state requirements.

  At some point in the early stages of the application, a thread attribute object can be initialized for each set. All future thread creations will then refer to the attribute object initialized for that type of thread. The initialization phase is simple and localized, and any future modifications can be made quickly and reliably.

Attribute objects require attention at process exit time. When the object is initialized, memory is allocated for it. This memory must be returned to the system. Attribute destroy function calls are provided to do this.

Initialize Attributes

pthread_attr_init(3T)

Use `pthread_attr_init()` to initialize the attributes associated with the object to the default values. The storage is allocated by the thread system during execution.

```
Prototype:
int pthread_attr_init(pthread_attr_t *tattr);
```

```
#include <pthread.h>

pthread_attr_t tattr;
int ret;

/* initialize an attribute to the default value */
ret = pthread_attr_init(&tattr);
```

The default values for attributes (*tattr*) are:

Table 3-1 Default Attribute Values

Attribute	Value	Result
scope	PTHREAD_SCOPE_PROCESS	New thread is unbound – not permanently attached to LWP
detachstate	PTHREAD_CREATE_JOINABLE	Exit status and thread are preserved after the thread terminates.
stackaddr	NULL	New thread has system-allocated stack address
stacksize	1 megabyte	New thread has system-defined stack size

Table 3-1 Default Attribute Values

Attribute	Value	Result
`priority`		New thread inherits parent thread priority
`inheritsched`	`PTHREAD_INHERIT_SCHED`	New thread inherits parent thread scheduling priority
`schedpolicy`	`SCHED_OTHER`	New thread uses Solaris-defined fixed priority scheduling; threads run until preempted by a higher-priority thread or until they block or yield.

Return Values

Returns zero after completing successfully. Any other returned value indicates that an error occurred. If the following condition occurs, the function fails and returns the corresponding value.

`ENOMEM` – Returned when there is not enough memory to initialize the thread attributes object.

Destroy Attributes

`pthread_attr_destroy(3T)`

Use `pthread_attr_destroy()` to remove the storage allocated during initialization. The attribute object becomes invalid.

```
Prototype:
int pthread_attr_destroy(pthread_attr_t *tattr);
```

```
#include <pthread.h>

pthread_attr_t tattr;
int ret;

/* destroy an attribute */
ret = pthread_attr_destroy(&tattr);
```

Return Values

Returns zero after completing successfully. Any other returned value indicates that an error occurred. If the following condition occurs, the function fails and returns the corresponding value.

`EINVAL` – Indicates that the value of *tattr* was not valid.

Set Detach State

pthread_attr_setdetachstate(3T)

When a thread is created detached (`PTHREAD_CREATE_DETACHED`), its thread ID and other resources can be reused as soon as the thread terminates. Use `pthread_attr_setdetachstate()` when you do not want to wait for the thread to terminate.

When a thread is created nondetached (`PTHREAD_CREATE_JOINABLE`), it is assumed that the you will be waiting for it. That is, it is assumed that you will be executing a `pthread_join`(3T) on the thread.

Prototype:

```
int pthread_attr_setdetachstate(pthread_attr_t *tattr,int detachstate);
```

```
#include <pthread.h>

pthread_attr_t tattr;
int ret;

/* set the thread detach state */
ret = pthread_attr_setdetachstate(&tattr,PTHREAD_CREATE_DETACHED);
```

Note – When there is no explicit synchronization to prevent it, a newly created, detached thread can die and have its thread ID reassigned to another new thread before its creator returns from `pthread_create`().

For nondetached (PTHREAD_CREATE_JOINABLE) threads, it is very important that some thread join with it after it terminates—otherwise the resources of that thread are not released for use by new threads. This commonly results in a memory leak. So when you do not want a thread to be joined, create it as a detached thread.

Code Example 3-1 Creating a Detached Thread

```
#include <pthread.h>

pthread_attr_t tattr;
pthread_t tid;
void *start_routine;
void arg
int ret;

/* initialized with default attributes */
ret = pthread_attr_init(&tattr);
ret = pthread_attr_setdetachstate(&tattr,PTHREAD_CREATE_DETACHED);
ret = pthread_create(&tid, &tattr, start_routine, arg);
```

Return Values

Returns zero after completing successfully. Any other returned value indicates that an error occurred. If the following condition occurs, the function fails and returns the corresponding value.

EINVAL – Indicates that the value of *detachstate* or *tattr* was not valid.

Get Detach State

pthread_attr_getdetachstate(3T)

Use pthread_attr_getdetachstate() to retrieve the thread create state, which can be either detached or joined.

```
Prototype:
int pthread_attr_getdetachstate(const pthread_attr_t *tattr,
    int *detachstate;
```

```
#include <pthread.h>

pthread_attr_t tattr;
int detachstate;
int ret;

/* get detachstate of thread */
ret = pthread_attr_getdetachstate (&tattr, &detachstate);
```

Return Values

Returns zero after completing successfully. Any other returned value indicates that an error occurred. If the following condition occurs, the function fails and returns the corresponding value.

EINVAL – Indicates that the value of *detachstate* is NULL or *tattr* is invalid.

Set Scope

pthread_attr_setscope(3T)

Use pthread_attr_setscope() to create a bound thread (PTHREAD_SCOPE_SYSTEM) or an unbound thread (PTHREAD_SCOPE_PROCESS).

```
Prototype:
int pthread_attr_setscope(pthread_attr_t *tattr,int scope);
```

```
#include <pthread.h>

pthread_attr_t tattr;
int ret;

/* bound thread */
ret = pthread_attr_setscope(&tattr, PTHREAD_SCOPE_SYSTEM);

/* unbound thread */
ret = pthread_attr_setscope(&tattr, PTHREAD_SCOPE_PROCESS);
```

Notice that there are three function calls in this example: one to initialize the attributes, one to set any variations from the default attributes, and one to create the pthreads.

Table 3-2 Creating a Bound Thread

```
#include <pthread.h>

pthread_attr_t tattr;
pthread_t tid;
void start_routine;
void arg;
int ret;

/* initialized with default attributes */
ret = pthread_attr_init(&tattr);

/* BOUND behavior */
ret = pthread_attr_setscope(&tattr, PTHREAD_SCOPE_SYSTEM);
ret = pthread_create(&tid, &tattr, start_routine, arg);
```

Return Values

Returns zero after completing *successfully*. Any other returned value indicates that an error occurred. If the following conditions occur, the function fails and returns the corresponding value.

`EINVAL` – An attempt was made to set *tattr* to a value that is not valid.

Get Scope

pthread_attr_getscope(3T)

Use this routine to retrieve the thread scope, which can be process or system.

```
Prototype:
int pthread_attr_getscope(pthread_attr_t *tattr, int scope);
```

```
#include <pthread.h>

pthread_attr_t tattr;
int scope;
int ret;

/* get scope of thread */
ret = pthread_attr_getscope(&tattr, &scope);
```

Return Values

Returns zero after completing successfully. Any other returned value indicates that an error occurred. If the following condition occurs, the function fails and returns the corresponding value.

`EINVAL` – The value of *scope* is NULL or *tattr* is invalid.

Set Scheduling Policy

pthread_attr_setschedpolicy(3T)

Use `pthread_attr_setschedpolicy()` to set the scheduling policy. The POSIX draft standard specifies scheduling policy attributes of `SCHED_FIFO` (first-in-first-out), `SCHED_RR` (round-robin), or `SCHED_OTHER` (an implementation-defined method).

`SCHED_FIFO` and `SCHED_RR` are optional in POSIX, and are supported for Realtime bound threads, only.

Currently, only the Solaris-based SCHED_OTHER is supported in pthreads. For a discussion of scheduling, see the section “Scheduling” on page 7.

Prototype:
```
int pthread_attr_setschedpolicy(pthread_attr_t *tattr, int policy);
```

```
#include <pthread.h>

pthread_attr_t tattr;
int policy;
int ret;

/* set the scheduling policy to SCHED_OTHER */
ret = pthread_attr_setschedpolicy(&tattr, SCHED_OTHER);
```

Return Values

Returns zero after completing successfully. Any other returned value indicates that an error occurred. When either of the following conditions occurs, the function fails and returns the corresponding value.

EINVAL – An attempt was made to set *tattr* to a value that is not valid.

ENOTSUP – An attempt was made to set the attribute to an unsupported value.

Get Scheduling Policy

pthread_attr_getschedpolicy(3T)

Use `pthread_attr_getschedpolicy()` to retrieve the scheduling policy.

```
Prototype:
int pthread_attr_getschedpolicy(pthread_attr_t *tattr, int policy);
```

```
#include <pthread.h>

pthread_attr_t tattr;
int policy;
int ret;

/* get scheduling policy of thread */
ret = pthread_attr_getschedpolicy (&tattr, &policy);
```

Return Values

Returns zero after completing successfully. Any other returned value indicates that an error occurred. If the following condition occurs, the function fails and returns the corresponding value.

`EINVAL` – The parameter *policy* is NULL or *tattr* is invalid.

Set Inherited Scheduling Policy

pthread_attr_setinheritsched(3T)

An *inherit* value of PTHREAD_INHERIT_SCHED (the default) means that the scheduling policies defined in the creating thread are to be used, and any scheduling attributes defined in the pthread_create() call are to be ignored. If PTHREAD_EXPLICIT_SCHED is used, the attributes from the pthread_create() call are to be used.

```
Prototype:
int pthread_attr_setinheritsched(pthread_attr_t *tattr, int inherit);
```

```
#include <pthread.h>

pthread_attr_t tattr;
int inherit;
int ret;

/* use the current scheduling policy */
ret = pthread_attr_setinheritsched(&tattr, PTHREAD_EXPLICIT_SCHED);
```

Return Values

Returns zero after completing successfully. Any other returned value indicates that an error occurred. When either of the following conditions occurs, the function fails and returns the corresponding value.

EINVAL – An attempt was made to set *tattr* to a value that is not valid.

ENOTSUP – An attempt was made to set the attribute to an unsupported value.

Get Inherited Scheduling Policy

pthread_attr_getinheritsched(3T)

Prototype:

```
int pthread_attr_getinheritsched(pthread_attr_t *tattr, int inherit);
```

```
#include <pthread.h>

pthread_attr_t tattr;
int inherit;
int ret;

/* get scheduling policies of the creating thread */
ret = pthread_attr_getinheritsched (&tattr, &inherit);
```

Return Values

Returns zero after completing successfully. Any other returned value indicates that an error occurred. If the following condition occurs, the function fails and returns the corresponding value.

`EINVAL` – The parameter *inherit* is NULL or *tattr* is invalid.

Set Scheduling Parameters

pthread_attr_setschedparam(3T)

Scheduling parameters are defined in the `param` structure; only priority is supported. Newly created threads run with this priority.

```
Prototype:
int pthread_attr_setschedparam(pthread_attr_t *tattr,
    const struct sched_param *param);
```

```
#include <pthread.h>

pthread_attr_t tattr;
int newprio;
sched_param param;
newprio = 30;

/* set the priority; others are unchanged */
param.sched_priority = newprio;

/* set the new scheduling param */
ret = pthread_attr_setschedparam (&tattr, &param);
```

Return Values

Returns zero after completing successfully. Any other returned value indicates that an error occurred. If the following conditions occur, the function fails and returns the corresponding value.

`EINVAL` – The value of *param* is NULL or *tattr* is invalid.

You can manage pthreads priority two ways. You can set the priority attribute before creating a child thread, or you can change the priority of the parent thread and then change it back.

Get Scheduling Parameters

pthread_attr_getschedparam(3T)

```
Prototype:
int pthread_attr_getschedparam(pthread_attr_t *tattr,
    const struct sched_param *param);
```

```
#include <pthread.h>

pthread_attr_t attr;
sched_param param;
int ret;

/* get the existing scheduling param */
ret = pthread_attr_getschedparam (&tattr, &param);
```

Return Values

Returns zero after completing successfully. Any other returned value indicates that an error occurred. If the following condition occurs, the function fails and returns the corresponding value.

`EINVAL` – The value of *param* is NULL or *tattr* is invalid.

Creating a Thread With a Specified Priority

You can set the priority attribute before creating the thread. The child thread is created with the new priority that is specified in the `sched_param` structure (this structure also contains other scheduling information).

It is always a good idea to get the existing parameters, change the priority, and then set it. Code Example 3-2 shows an example of this.

Code Example 3-2 Creating a Prioritized Thread

```
#include <pthread.h>
#include <sched.h>

pthread_attr_t tattr;
pthread_t tid;
int ret;
int newprio = 20;
sched_param param;

/* initialized with default attributes */
ret = pthread_attr_init (&tattr);

/* safe to get existing scheduling param */
ret = pthread_attr_getschedparam (&tattr, &param);

/* set the priority; others are unchanged */
param.sched_priority = newprio;

/* setting the new scheduling param */
ret = pthread_attr_setschedparam (&tattr, &param);

/* with new priority specified */
ret = pthread_create (&tid, &tattr, func, arg);
```

Set Stack Size

pthread_attr_setstacksize(3T)

The stacksize attribute defines the size of the stack (in bytes) that the system will allocate. The size should not be less than the system-defined minimum stack size. See "About Stacks" on page 61 for more information.

```
Prototype:
int pthread_attr_setstacksize(pthread_attr_t *tattr, int size);
```

```
#include <pthread.h>

pthread_attr_t tattr;
int size;
int ret;

size = (PTHREAD_STACK_MIN + 0x4000);

/* setting a new size */
ret = pthread_attr_setstacksize(&tattr, size);
```

In the example above, *size* contains the size, in number of bytes, for the stack that the new thread uses. If *size* is zero, a default size is used. In most cases, a zero value works best.

`PTHREAD_STACK_MIN` is the amount of stack space required to start a thread. This does not take into consideration the threads routine requirements that are needed to execute application code.

Return Values

Returns zero after completing successfully. Any other returned value indicates that an error occurred. If the following condition occurs, the function fails and returns the corresponding value.

`EINVAL` – The value returned is less than the value of `PTHREAD_STACK_MIN`, or exceeds a system-imposed limit, or *tattr* is not valid.

Get Stack Size

pthread_attr_getstacksize(3T)

```
Prototype:
int pthread_attr_getstacksize(pthread_attr_t *tattr, int size);
```

```
#include <pthread.h>

pthread_attr_t tattr;
int size;
int ret;

size = (PTHREAD_STACK_MIN + 0x1000);

/* getting the stack size */
ret = pthread_attr_getstacksize(&tattr, &size);
```

Return Values

Returns zero after completing successfully. Any other returned value indicates that an error occurred. If the following condition occurs, the function fails and returns the corresponding value.

`EINVAL` – The value returned is less than the value of `PTHREAD_STACK_MIN`, or exceeds a system-imposed limit.

About Stacks

Typically, thread stacks begin on page boundaries and any specified size is rounded up to the next page boundary. A page with no access permission is appended to the top of the stack so that most stack overflows result in sending a `SIGSEGV` signal to the offending thread. Thread stacks allocated by the caller are used as is.

When a stack is specified, the thread should also be created `PTHREAD_CREATE_JOINABLE`. That stack cannot be freed until the `pthread_join(3T)` call for that thread has returned, because the thread's stack cannot be freed until the thread has terminated. The only reliable way to know if a thread has terminated is through `pthread_join(3T)`.

Generally, you do not need to allocate stack space for threads. The threads library allocates one megabyte of virtual memory for each thread's stack with no swap space reserved. (The library uses the `MAP_NORESERVE` option of `mmap(2)` to make the allocations.)

Each thread stack created by the threads library has a red zone. The library creates the red zone by appending a page to the top of a stack to catch stack overflows. This page is invalid and causes a memory fault if it is accessed. Red zones are appended to all automatically allocated stacks whether the size is specified by the application or the default size is used.

Note – Because runtime stack requirements vary, you should be absolutely certain that the specified stack will satisfy the runtime requirements needed for library calls and dynamic linking.

There are very few occasions when it is sensible to specify a stack, its size, or both. It is difficult even for an expert to know if the right size was specified. This is because even an ABI-compliant program can't determine its stack size statically. Its size is dependent on the needs of the particular runtime environment in which it executes.

Building Your Own Stack

When you specify the size of a thread stack, be sure to account for the allocations needed by the invoked function and by each function called. The accounting should include calling sequence needs, local variables, and information structures.

Occasionally you want a stack that is a bit different from the default stack. An obvious situation is when the thread needs more than one megabyte of stack space. A less obvious situation is when the default stack is too large. You might be creating thousands of threads and not have enough virtual memory to handle the gigabytes of stack space that this many default stacks require.

The limits on the maximum size of a stack are often obvious, but what about the limits on its minimum size? There must be enough stack space to handle all of the stack frames that are pushed onto the stack, along with their local variables and so on.

You can get the absolute minimum limit on stack size by calling the macro PTHREAD_STACK_MIN(), which returns the amount of stack space required for a thread that executes a null procedure. Useful threads need more than this, so be very careful when reducing the stack size.

When you allocate your own stack, be sure to append a red zone to its end by calling mprotect(2).

```
#include <pthread.h>

pthread_attr_t tattr;
pthread_t tid;
int ret;

int size = PTHREAD_STACK_MIN + 0x4000;

/* initialized with default attributes */
ret = pthread_attr_init(&tattr);

/* setting the size of the stack also */
ret = pthread_attr_setstacksize(&tattr, size);

/* only size specified in tattr*/
ret = pthread_create(&tid, &tattr, start_routine, arg);
```

Set Stack Address

pthread_attr_setstackaddr(3T)

The stackaddr attribute defines the base of the thread's stack. If this is set to non-null (NULL is the default) the system initializes the stack at that address.

```
Prototype:
int pthread_attr_setstackaddr(pthread_attr_t *tattr,void **stackaddr);
```

```
#include <pthread.h>

pthread_attr_t tattr;
void *base;
int ret;

base = (void *) malloc(PTHREAD_STACK_MIN + 0x4000);

/* setting a new address */
ret = pthread_attr_setstackaddr(&tattr, base);
```

In the example above, *base* contains the address for the stack that the new thread uses. If *base* is NULL, then pthread_create(3T) allocates a stack for the new thread with at least PTHREAD_STACK_MIN bytes.

Return Values

Returns zero after completing successfully. Any other returned value indicates that an error occurred. If the following condition occurs, the function fails and returns the corresponding value.

EINVAL – The value or *base* or *tattr* is incorrect.

This example shows how to create a thread with a custom stack address.

```
#include <pthread.h>

pthread_attr_t tattr;
pthread_t tid;
int ret;
void *stackbase;

stackbase = (void *) malloc(size);

/* initialized with default attributes */
ret = pthread_attr_init(&tattr);

/* setting the base address in the attribute */
ret = pthread_attr_setstackaddr(&tattr, stackbase);

/* only address specified in attribute tattr */
ret = pthread_create(&tid, &tattr, func, arg);
```

This example shows how to create a thread with both a custom stack address and a custom stack size.

```
#include <pthread.h>

pthread_attr_t tattr;
pthread_t tid;
int ret;
void *stackbase;

int size = PTHREAD_STACK_MIN + 0x4000;
stackbase = (void *) malloc(size);

/* initialized with default attributes */
ret = pthread_attr_init(&tattr);

/* setting the size of the stack also */
ret = pthread_attr_setstacksize(&tattr, size);

/* setting the base address in the attribute */
ret = pthread_attr_setstackaddr(&tattr, stackbase);

/*address and size specified */
ret = pthread_create(&tid, &tattr, func, arg);
```

Get Stack Address

pthread_attr_getstackaddr(3T)

```
Prototype:
int pthread_attr_getstackaddr(pthread_attr_t *tattr,void **stackaddr);
```

```
#include <pthread.h>

pthread_attr_t tattr;
void *base;
int ret;

base = (void *) malloc(PTHREAD_STACK_MIN + 0x1000);

/* getting a new address */
ret = pthread_attr_getstackaddr (&tattr, base);
```

Return Values

Returns zero after completing successfully. Any other returned value indicates that an error occurred. If the following condition occurs, the function fails and returns the corresponding value.

EINVAL – The value or *base* or *tattr* is incorrect.