Subsections

Further Threads Programming:Thread Attributes (POSIX)

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.

Attributes

Attributes are a way to specify behavior that is different from the default. When a thread is created with `pthread_create()` or when a synchronization variable is initialized, an attribute object can be specified. **Note:** however that the default atributes are usually sufficient for most applications.

Impottant Note: Attributes are specified ***only at thread creation time***; they **cannot** be altered while the thread is **being used**.

Thus three functions are usually called in tandem

- Thread attibute intialisation -- `pthread_attr_init()` create a default `pthread_attr_t tattr`
- Thread attribute value change (unless defaults appropriate) -- a variety of `pthread_attr_*()` functions are available to set individual attribute values for the `pthread_attr_t tattr` structure. (see below).
- Thread creation -- a call to `pthread_create()` with approriate attribute values set in a `pthread_attr_t tattr` structure.

The following code fragment should make this point clearer:

```
#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);

/* call an appropriate functions to alter a default value */
```

```
ret = pthread_attr_*(&tattr,SOME_ATRIBUTE_VALUE_PARAMETER);

/* create the thread */
ret = pthread_create(&tid, &tattr, start_routine, arg);
```

In order to save space, code examples mainly focus on the attribute setting functions and the intializing and creation functions are ommitted. These **must** of course be present in all actual code fragtments.

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. The pthreads standard provides function calls to destroy attribute objects.

Initializing Thread Attributes

The function `pthread_attr_init()` is used to initialize object attributes to their default values. The storage is allocated by the thread system during execution.

The function is prototyped by:

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

An example call to this function is:

```
#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:

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.
		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.

This function zero after completing successfully. Any other returned value indicates that an error occurred. If the following condition occurs, the function fails and returns an error value (to `errno`).

Destroying Thread Attributes

The function `pthread_attr_destroy()` is used to remove the storage allocated during initialization. The attribute object becomes invalid. It is prototyped by:

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

A sample call to this functions is:

```
#include <pthread.h>
pthread_attr_t tattr;
int ret;
/* destroy an attribute */
ret = pthread_attr_destroy(&tattr);
```

Attribites are declared as for `pthread_attr_init()` above.

`pthread_attr_destroy()` returns zero after completing successfully. Any other returned value indicates that an error occurred.

Thread's Detach State

When a thread is created detached (`PTHREAD_CREATE_DETACHED`), its thread ID and other resources can be reused as soon as the thread terminates.

If you do not want the calling thread to wait for the thread to terminate then call the function `pthread_attr_setdetachstate()`.

When a thread is created nondetached (`PTHREAD_CREATE_JOINABLE`), it is assumed that you will be waiting for it. That is, it is assumed that you will be executing a `pthread_join()` on the thread. Whether a thread is created detached or nondetached, the process does not exit until all threads have exited.

`pthread_attr_setdetachstate()` is prototyped by:

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

`pthread_attr_setdetachstate()` 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.

An example call to detatch a thread with this function is:

```
#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.

It is quite common that you will wish to create a thread which is detatched from creation. The

following code illustrates how this may be achieved with the standard calls to initialise and set and then create a 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);
```

The function `pthread_attr_getdetachstate()` may be used to retrieve the thread create state, which can be either detached or joined. It is prototyped by:

```
int pthread_attr_getdetachstate(const pthread_attr_t *tattr, int *detachstate);
```

`pthread_attr_getdetachstate()` returns zero after completing successfully. Any other returned value indicates that an error occurred.

An example call to this fuction is:

```
#include <pthread.h>
pthread_attr_t tattr;
int detachstate;
int ret;

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

Thread's Set Scope

A thread may be bound (`PTHREAD_SCOPE_SYSTEM`) or an unbound (`PTHREAD_SCOPE_PROCESS`). Both these types of types are accessible **only** within a given process.

The function `pthread_attr_setscope()` to create a bound or unbound thread. It is prototyped by:

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

Scope takes on the value of either `PTHREAD_SCOP_SYSTEM` or `PTHREAD_SCOPE_PROCESS`.

`pthread_attr_setscope()` returns zero after completing successfully. Any other returned value indicates that an error occurred and an appropriate value is returned.

So to set a bound thread at thread creation on would do the following function calls:

```
#include <pthread.h>

pthread_attr_t attr;
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);
```

If the following conditions occur, the function fails and returns the corresponding value.

The function `pthread_attr_getscope()` is used to retrieve the thread scope, which indicates

whether the thread is bound or unbound. It is prototyped by:

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

An example use of this function is:

```
#include <pthread.h>

pthread_attr_t tattr;
int scope;
int ret;

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

If successful the approriate (`PTHREAD_SCOP_SYSTEM` or `PTHREAD_SCOPE_PROCESS`) wil be stored in `scope`.

`pthread_att_getscope()` returns zero after completing successfully. Any other returned value indicates that an error occurred.

Thread 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 **only** are supported for ***real time bound threads***.

Howver Note, currently, only the Solaris `SCHED_OTHER` default value is supported in pthreads. Attempting to set policy as `SCHED_FIFO` or `SCHED_RR` will result in the error `ENOSUP`.

The function is used to set the scheduling policy.It is prototyped by:

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

`pthread_attr_setschedpolicy()` returns zero after completing successfully. Any other returned value indicates that an error occurred.

To set the scheduling policy to `SCHED_OTHER` simply do:

```
#include <pthread.h>
pthread_attr_t tattr;
int ret;

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

There is a function `pthread_attr_getschedpolicy()` that retrieves the scheduling policy. But, currently, it is not of great use as it can only return the (Solaris-based) `SCHED_OTHER` default value

Thread Inherited Scheduling Policy

The function `pthread_attr_setinheritsched()` can be used to the inherited scheduling policy of a thread. It is prototyped by:

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

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.

The function returns zero after completing successfully. Any other returned value indicates that

an error occurred.

An example call of this function is:

```
#include <pthread.h>
pthread_attr_t tattr;
int ret;

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

The function `pthread_attr_getinheritsched(pthread_attr_t *tattr, int *inherit)` may be used to inquire a current threads scheduling policy.

Set Scheduling Parameters

Scheduling parameters are defined in the `sched_param` structure; **only** priority `sched_param.sched_priority` is supported. This priority is an integer value the higher the value the higher a thread's proiority for scehduling. Newly created threads run with this priority. The `pthread_attr_setschedparam()` is used to set this stucture appropiately. It is prototyped by:

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

and returns zero after completing successfully. Any other returned value indicates that an error occurred.

An example call to `pthread_attr_setschedparam()` is:

```
#include <pthread.h>
pthread_attr_t tattr;
int newprio;
sched_param param;


/* set the priority; others are unchanged */
newprio = 30;
param.sched_priority = newprio;

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

The function `pthread_attr_getschedparam(pthread_attr_t *tattr, const struct sched_param *param)` may be used to inquire a current thread's priority of scheduling.

Thread Stack Size

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()` 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 such a thread has terminated is through `pthread_join()`.

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 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 appropriate 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 a program compliant with ABI standards cannot determine its stack size statically. Its size is dependent on the needs of the particular runtime environment in which it executes.

Building Your Own Thread 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` (defined in <pthread.h>), 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.

The function `pthread_attr_setstacksize()` is used to set this a thread's stack size, it is prototyped by:

```
int pthread_attr_setstacksize(pthread_attr_t *tattr, int stacksize);
```

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.

`pthread_attr_setstacksize()` returns zero after completing successfully. Any other returned value indicates that an error occurred.

An example call to set the stacksize is:

```
#include <pthread.h>

pthread_attr_t tattr;
int stacksize;
int ret;


/* setting a new size */
stacksize = (PTHREAD_STACK_MIN + 0x4000);
ret = pthread_attr_setstacksize(&tattr, stacksize);
```

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.

The function `pthread_attr_getstacksize(pthread_attr_t *tattr, size_t *size)` may be used to inquire about a current threads stack size as follows:

```
#include <pthread.h>

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

The function only returns the minimum stack size (in bytes) allocated for the created threads stack to the variable `stacksize`. **It DOES NOT RETURN the actual stack size** so use the function with care.

You may wish to specify the base adress of thread's stack. The function `pthread_attr_setstackaddr()` does this task. It is prototyped by:

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

The `stackaddr` parameter 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.

The function returns zero after completing successfully. Any other returned value indicates that an error occurred.

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);
```

The function `pthread_attr_getstackaddr(pthread_attr_t *tattr,void * *stackaddr)` can be used to obtain the base address for a current thread's stack address.

Dave Marshall
1/5/1999