NAME

pthread_create, thr_create – thread creation

SYNOPSIS

POSIX

cc [*flag* ...] *file* ... **–lpthread** [*library* ...]

#include <pthread.h>
int pthread_create(pthread_t **new_thread_ID,* **const pthread_attr_t** **attr,*
void * **(****start_func***)(void *), void** **arg***);**

Solaris

cc [*flag* ...] *file* ... **–lthread** [*library* ...]

#include <thread.h>
int thr_create(void **stack_base*, **size_t** *stack_size*, **void** ***(****start_func***)(void *),**
void **arg*, **long** *flags*, **thread_t** **new_thread_ID***);**

MT-LEVEL

MT-Safe

DESCRIPTION

Thread creation adds a new thread of control to the current process. The procedure **main( )**, itself, is a single thread of control. Each thread executes simultaneously with all the other threads within the calling process, and with other threads from other active processes.

A newly created thread shares all of the calling process' global data with the other threads in this process; however, it has its own set of attributes and private execution stack. The new thread inherits the calling thread's signal mask, possibly, and scheduling priority. Pending signals for a new thread are not inherited and will be empty.

The call to create a thread takes the address of a user-defined function, specified by *start_func*, as one of its arguments, which is the complete execution routine for the new thread.

The lifetime of a thread begins with the successful return from **pthread_create( )** or **thr_create( )**, which calls *start_func*() and ends with either:

- the normal completion of *start_func*(),
- the return from an explicit call to **pthread_exit**(3T) or **thr_exit**(3T),
- a thread cancellation (see **pthread_cancel**(3T)). or
- the conclusion of the calling process (see **exit**(2)).

The new thread performs by calling the function defined by *start_func* with one argument, *arg*. If more than one argument needs to be passed to *start_func*, the arguments can be packed into a structure, and the address of that structure can be passed to *arg*.

If *start_func* returns, the thread will terminate with the exit status set to the *start_func* return value (see **pthread_exit**(3T) or **thr_exit**(3T)).

Note that when the thread returns in which **main( )** originated from, the effect is the same as if there were an implicit call to **exit( )** using the return value of **main( )** as the exit status. This differs from a *start_func* return. However, if **main( )** itself calls either **pthread_exit**(3T) or **thr_exit**(3T), only the **main** thread exits, not the entire process.

If the thread creation itself fails, a new thread is not created and the contents of the location referenced by the pointer to the new thread are undefined.

Attributes

The configuration of a set of attributes defines the behavior of a thread. At creation, each attribute of a new thread may be user-defined or set to the default. All attributes are defined upon thread creation, however, some may be dynamically modified after creation. Establishing these attributes varies depending upon whether POSIX or Solaris threads are used. Both implementations offer a few attributes the other does not.

The available attributes are:

Attribute	Description	API
contentionscope	Scheduled by threads library (local scope) or scheduled by the OS (global scope)	both
detachstate	Allows other threads to wait for a particular thread to terminate	both
stackaddr	Sets a pointer to the thread's stack	both
stacksize	Sets the size of the thread's stack	both
concurrency	Elevates concurrency, if possible	Solaris
priority	Sets ranking within the policy (scheduling class)	both
policy	Sets scheduling class; **SCHED_OTHER**	POSIX
inheritsched	Determines whether scheduling parameters are inherited or explicitly defined	POSIX
suspended	Sets thread to runnable vs. suspended	Solaris
daemon	Defines a thread's behavior to be like a daemon	Solaris

POSIX

pthread_create( ) creates a new thread within a process with attributes defined by *attr*. Default attributes are used if *attr* is **NULL**. If any attributes specified by *attr* are changed in the attribute object prior to the call to **pthread_create( )**, the new thread will acquire those changes. However, if any attributes specified by *attr* are changed after the call to **pthread_create( )**, the attributes of existing threads will not be affected. Since **pthread_create( )** can use an attribute object in its call, a user-defined thread creation must be preceded by a user-defined attribute object (see **pthread_attr_init**(3T)). Upon successful completion, and if the return value is not **NULL, pthread_create( )** will store the ID of the created thread in the location referenced by *new_thread_ID*.

It is recommended that for POSIX thread creation, all attribute objects, *attr*s, which will be used later during creation calls, be initialized and modified in the early stages of program execution.

The default creation attributes for **pthread_create**(3T) are:

Attribute	Default Value	Meaning of Default Value
contentionscope	**PTHREAD_SCOPE_PROCESS**	Resource competition within process
detachstate	**PTHREAD_CREATE_JOINABLE**	Joinable by other threads
stackaddr	**NULL**	Allocated by system
stacksize	**NULL**	1 megabyte
priority	**NULL**	Parent (calling) thread's priority
policy	**SCHED_OTHER**	Determined by system

inheritsched	**PTHREAD_EXPLICIT_SCHED**	Scheduling attributes explicitly set, e.g., policy is **SCHED_OTHER**.

Default thread creation:

```
pthread_t tid;
void *start_func(void *), *arg;

pthread_create(&tid, NULL, start_func, arg);
```

This would have the same effect as:

```
pthread_attr_t attr;

pthread_attr_init(&attr); /* initialize attr with default attributes */
pthread_create(&tid, &attr, start_func, arg);
```

User-defined thread creation:

To create a thread that is scheduled on a system-wide basis (i.e., a bound thread, as per the Solaris API), use:

```
pthread_attr_init(&attr); /* initialize attr with default attributes */
pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);  /* system-wide contention */
pthread_create(&tid, &attr, start_func, arg);
```

To customize the attributes for POSIX threads, see **pthread_attr_init**(3T).

A new thread created with **pthread_create( )** uses the stack specified by the *stackaddr* attribute, and the stack continues for the number of bytes specified by the *stacksize* attribute. By default, the stack size is 1 megabyte (see **pthread_attr_setstacksize**(3T)). If the default is used for both the *stackaddr* and *stacksize* attributes, **pthread_create( )** creates a stack for the new thread with at least 1 megabyte. (For customizing stack sizes, see NOTES).

Solaris

In the Solaris API, **thr_create( )** either results in the creation of a default thread or a thread whose attributes are defined by the *flags* passed to **thr_create( )**. There is no attribute object to configure, as there is in POSIX. The attributes are either the separate arguments, *stackaddr* or *stacksize*, or the result of bitwise inclusive OR-ing the possible values for *flags*.

The creation attributes for **thr_create**(3T) are:

Attribute	Default Value	Meaning of Default Value	Specified Via
contentionscope	NULL	Resource competition within process	flags
detachstate	NULL	Joinable by other threads	flags
stackaddr	NULL	Allocated by system	separate argument
stacksize	NULL	1 megabyte	separate argument
priority	NULL	Parent (calling) thread's priority	
concurrency	NULL	Determined by system	flags

suspended	NULL	Runnable, not suspended	flags
daemon	NULL	Not a daemon	flags

flags specifies which attributes are modifiable for the created thread. The value in *flags* is determined by the bitwise inclusive OR of the following:

THR_BOUND This flag affects the contentionscope attribute of the thread. The new thread is created permanently bound to an LWP (i.e. it is a *bound thread*). This thread will now contend among system-wide resources. The bind flag is equivalent to setting the *contentionscope* to the **PTHREAD_SCOPE_SYSTEM** in POSIX.

THR_DETACHED This flag affects the detachstate attribute of the thread. The new thread is created detached. The exit status of a detached thread is not accessible to other threads. Its thread ID and other resources may be re-used as soon as the thread terminates. **thr_join**(3T) (nor **pthread_join**(3T)) will not wait for a detached thread. This is equivalent to **PTHREAD_CREATE_DETACHED** in POSIX, which is the default for POSIX.

THR_NEW_LWP This flag affects the concurrency attribute of the thread. The desired concurrency level for unbound threads is increased by one. This is similar to incrementing concurrency by one via **thr_setconcurrency**(3T)**.** Typically, this adds a new LWP to the pool of LWPs running unbound threads.

THR_SUSPENDED This flag affects the suspended attribute of the thread. The new thread is created suspended and will not execute *start_func* until it is started by **thr_continue( )**.

THR_DAEMON This flag affects the daemon attribute of the thread. The thread is marked as a daemon. The process will exit when all non-daemon threads exit. **thr_join**(3T) will not wait for a daemon thread. Daemon threads do not interfere with the exit conditions for a process. A process will terminate when all regular threads exit or the process calls **exit( )**. Daemon threads are most useful in libraries that want to use threads.

Default thread creation:

```
thread_t tid;
void *start_func(void *), *arg;
thr_create(NULL, NULL, start_func, arg, NULL, &tid);
```

User-defined thread creation:

To create a thread scheduled on a system-wide basis (i.e., a bound thread), use:

```
thr_create(NULL, NULL, start_func, arg, THR_BOUND, &tid);
```

Another example of customization is, if both **THR_BOUND** and **THR_NEW_LWP** are specified then, typically, two LWPs are created, one for the bound thread and another for the pool of LWPs running unbound threads.

thr_create(NULL, NULL, start_func, arg, THR_BOUND | THR_NEW_LWP, &tid);

With **thr_create( ),** the new thread will use the stack starting at the address specified by *stack_base* and continuing for *stack_size* bytes. *stack_size* must be greater than the value returned by **thr_min_stack**(3T)**.** If *stack_base* is NULL then **thr_create( )** allocates a stack for the new thread with at least *stack_size* bytes. If *stack_size* is zero then a default size is used. If *stack_size* is not zero then it must be greater than the value returned by **thr_min_stack**(3T) (see NOTES).

When *new_thread_ID* is not NULL then it points to a location where the ID of the new thread is stored if **thr_create( )** is successful. The ID is only valid within the calling process.

RETURN VALUES

Zero indicates a successful return and a non-zero value indicates an error.

ERRORS

If any of the following conditions occur, these functions fail and return the corresponding value:

EAGAIN The system-imposed limit on the total number of threads in a process has been exceeded or some system resource has been exceeded (e.g., too many LWPs were created).

EINVAL The value specified by *attr* is invalid.

If any of the following conditions are detected, **pthread_create( )** fails and returns the corresponding value:

ENOMEM Not enough memory was available to create the new thread.

If any of the following conditions are detected, **thr_create( )** fails and returns the corresponding value:

EINVAL
- *stack_base* is not NULL and *stack_size* is less than the value returned by **thr_min_stack**(3T)**.**
- *stack_base* is NULL and *stack_size* is not zero and is less than the value returned by **thr_min_stack**(3T)**.**

EXAMPLES

This is an example of concurrency with multi-threading. Since POSIX threads and Solaris threads are fully compatible even within the same process, this example uses **pthread_create**(3T) if you execute **a.out 0**, or **thr_create**(3T) if you execute **a.out 1**.

Five threads are created that simultaneously perform a time-consuming function, **sleep(**10**)**. If the execution of this process is timed, the results will show that all five individual calls to sleep for ten-seconds completed in about ten seconds, even on a uniprocessor. If a single-threaded process calls **sleep(**10**)** five times, the execution time will be about 50-seconds.

The command-line to time this process is:

/usr/bin/time a.out 0 (for POSIX threading)

or

/usr/bin/time a.out 1 (for Solaris threading)

```
/* cc thisfile.c -lthread -lpthread */
#define _REENTRANT    /* basic 3-lines for threads */
#include <pthread.h>
#include <thread.h>

#define NUM_THREADS 5
#define SLEEP_TIME 10

void *sleeping(void *);   /* thread routine */
void test_argv();         /* optional */
int i;
thread_t tid[NUM_THREADS];      /* array of thread IDs */

main( int argc, char *argv[] ) {
  test_argv(argv[1]);

  switch (*argv[1])  {
   case '0':       /* POSIX */
    for ( i = 0; i < NUM_THREADS; i++)
     pthread_create(&tid[i], NULL, sleeping, SLEEP_TIME);
    for ( i = 0; i < NUM_THREADS; i++)
     pthread_join(tid[i], NULL);
    break;

   case '1':       /* Solaris */
    for ( i = 0; i < NUM_THREADS; i++)
     thr_create(NULL,0,sleeping,NULL,0,&tid[i]);
    while (thr_join(NULL, NULL, NULL) == 0);
    break;
  }  /* switch */

  printf("main( ) reporting that all %d threads have terminated\n", i);
}  /* main */

void *sleeping(int sleep_time *)     {
     printf("thread %d sleeping %d seconds ...\n", thr_self(), SLEEP_TIME);
     sleep(sleep_time);
     printf("\nthread %d awakening\n", thr_self());
}
```

```
void test_argv(char argv1[])      {   /* optional */
 if (argv1 == NULL)  {
   printf("use 0 as arg1 to use thr_create();\n \
   or use 1 as arg1 for use pthread_create()\n");
   exit(NULL);
 }
}
```

If **main( )** had not waited for the completion of the other threads (using **pthread_join**(3T) or **thr_join**(3T)), it would have continued to process concurrently until it reached the end of its routine and the entire process would have exited prematurely (see **exit**(2)).

The following example shows how to create a default thread with a new signal mask. **new_mask** is assumed to have a different value than the creator's signal mask (orig_mask). **new_mask** is set to block all signals except for SIGINT. The creator's signal mask is changed so that the new thread inherits a different mask, and is restored to its original value after **thr_create()** returns.

This example assumes that SIGINT is also unmasked in the creator. If it is masked by the creator, then unmasking the signal opens the creator up to this signal. The other alternative is to have the new thread set its own signal mask in its start routine.

```
thread_t tid;
sigset_t new_mask, orig_mask;
int error;

(void)sigfillset(&new_mask);
(void)sigdelset(&new_mask, SIGINT);
(void)thr_sigsetmask(SIG_SETMASK, &new_mask, &orig_mask):
error = thr_create(NULL, 0, do_func, NULL, 0, &tid);
(void)thr_sigsetmask(SIG_SETMASK, &orig_mask, NULL);
```

SEE ALSO

_lwp_create(2), **exit**(2), **exit**(3C), **pthread_attr_init**(3T), **pthread_cancel**(3T), **pthread_exit**(3T), **pthread_join**(3T), **thr_suspend**(3T), **thr_min_stack**(3T), **thr_setconcurrency**(3T), **threads**(3T)

NOTES

MT application threads execute independently of each other, thus their relative behavior is unpredictable. Therefore, it is possible for the thread executing **main( )** to finish before all other user application threads.

Using **thr_join**(3T) in the following syntax,

```
while (thr_join(NULL, NULL, NULL) == 0);
```

will cause the invoking thread (which may be **main( )**) to wait for the termination of all other undetached and non-daemon threads; however, the second and third arguments to **thr_join**(3T) need not necessarily be **NULL.**

pthread_join(3T), on the other hand, must specify the terminating thread (IDs) for which it will wait.

A thread has not terminated until **thr_exit**() has finished. The only way to determine this is by **thr_join**(). When **thr_join**() returns a departed thread, it means that this thread has terminated and its resources are reclaimable. For instance, if a user specified a stack to **thr_create**(), this stack can only be reclaimed after **thr_join**() has reported this thread as a departed thread. It is not possible to determine when a *detached* thread has terminated. A detached thread disappears without leaving a trace.

Typically, thread stacks allocated by **thr_create( )** begin on page boundaries and any specified (a red-zone) 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 will result in a **SIGSEGV** signal being sent to the offending thread. Thread stacks allocated by the caller are used as is.

Using a default stack size for the new thread, instead of passing a user-specified stack size, results in much better **thr_create( )** performance. The default stack size for a user-thread is 1 megabyte, in this implementation.

A user-specified stack size must be greater than the value **THR_MIN_STACK** or **PTHREAD_STACK_MIN.** A minimum stack size may not accommodate the stack frame for the user thread function *start_func*. If a stack size is specified, it must accommodate *start_func* requirements and the functions that it may call in turn, in addition to the minimum requirement.

It is usually very difficult to determine the runtime stack requirements for a thread. **THR_MIN_STACK** or **PTHREAD_STACK_MIN** specifies how much stack storage is required to execute a NULL *start_func*. The total runtime requirements for stack storage are dependent on the storage required to do runtime linking, the amount of storage required by library runtimes (like **printf( )**) that your thread calls. Since these storage parameters are not known before the program runs, it is best to use default stacks. If you know your runtime requirements or decide to use stacks that are larger than the default, then it makes sense to specify your own stacks.