◆ ◆ ◆ 第 3 章

3 线程属性

前面一章介绍了使用缺省属性创建线程的基本原理。本章论述如何在创建线程时设置属性。

注 – 只有 pthreads 使用属性和取消功能。本章中介绍的 API 仅适用于 POSIX 线程。除此之外，Solaris 线程和 `pthreads` 的**功能**大致是相同的。有关相似和不同之处的更多信息，请参见第 8 章。

属性对象

通过设置属性，可以指定一种不同于缺省行为的行为。使用 `pthread_create(3C)` 创建线程时，或初始化同步变量时，可以指定属性对象。缺省值通常就足够了。

属性对象是不透明的，而且不能通过赋值直接进行修改。系统提供了一组函数，用于初始化、配置和销毁每种对象类型。

初始化和配置属性后，属性便具有进程范围的作用域。使用属性时最好的方法即是在程序执行早期一次配置好所有必需的状态规范。然后，根据需要引用相应的属性对象。

使用属性对象具有两个主要优点。

- 使用属性对象可增加代码可移植性。

  即使支持的属性可能会在实现之间有所变化，但您不需要修改用于创建线程实体的函数调用。这些函数调用不需要进行修改，因为属性对象是隐藏在接口之后的。

  如果目标系统支持的属性在当前系统中不存在，则必须显式提供才能管理新的属性。管理这些属性是一项非常容易的移植任务，因为只需在明确定义的位置初始化属性对象一次即可。

- 应用程序中的状态规范已被简化。

  例如，假设进程中可能存在多组线程。每组线程都提供单独的服务。每组线程都有各自的状态要求。

在应用程序执行初期的某一时间，可以针对每组线程初始化线程属性对象。以后所有线程的创建都会引用已经为这类线程初始化的属性对象。初始化阶段是简单和局部的。将来就可以快速且可靠地进行任何修改。

在进程退出时需要注意属性对象。初始化对象时，将为该对象分配内存。必须将此内存返回给系统。pthreads 标准提供了用于销毁属性对象的函数调用。

初始化属性

请使用 pthread_attr_init(3C) 将对象属性初始化为其缺省值。存储空间是在执行期间由线程系统分配的。

pthread_attr_init 语法

```
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);
```

表 3–1 给出了属性 (*tattr*) 的缺省值。

表 3–1 *tattr* 的缺省属性值

属性	值	结果
scope	PTHREAD_SCOPE_PROCESS	新线程与进程中的其他线程发生竞争。
detachstate	PTHREAD_CREATE_JOINABLE	线程退出后，保留完成状态和线程 *ID*。
stackaddr	NULL	新线程具有系统分配的栈地址。
stacksize	0	新线程具有系统定义的栈大小。
priority	0	新线程的优先级为 0。

表 3–1 *tattr* 的缺省属性值 （续）

属性	值	结果
inheritsched	PTHREAD_EXPLICIT_SCHED	新线程不继承父线程调度优先级。
schedpolicy	SCHED_OTHER	新线程对同步对象争用使用 Solaris 定义的固定优先级。线程将一直运行，直到被抢占或者直到线程阻塞或停止为止。

pthread_attr_init 返回值

pthread_attr_init() 成功完成后将返回零。其他任何返回值都表示出现了错误。如果出现以下情况，该函数将失败并返回对应的值。

ENOMEM
描述: 如果未分配足够的内存来初始化线程属性对象，将返回该值。

销毁属性

请使用 pthread_attr_destroy(3C) 删除初始化期间分配的存储空间。属性对象将会无效。

pthread_attr_destroy 语法

```
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);
```

pthread_attr_destroy 返回值

pthread_attr_destroy() 成功完成后将返回零。其他任何返回值都表示出现了错误。如果出现以下情况，该函数将失败并返回对应的值。

EINVAL
描述: 指示 *tattr* 的值无效。

设置分离状态

如果创建分离线程 (PTHREAD_CREATE_DETACHED)，则该线程一退出，便可重用其线程 *ID* 和其他资源。如果调用线程不准备等待线程退出，请使用 pthread_attr_setdetachstate(3C)。

pthread_attr_setdetachstate(3C) 语法

```
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);
```

如果使用 PTHREAD_CREATE_JOINABLE 创建非分离线程，则假设应用程序将等待线程完成。也就是说，程序将对线程执行 pthread_join()。

无论是创建分离线程还是非分离线程，在所有线程都退出之前，进程不会退出。有关从 main() 提前退出而导致的进程终止的讨论，请参见第 42 页中的 “结束”。

注 – 如果未执行显式同步来防止新创建的分离线程失败，则在线程创建者从 pthread_create() 返回之前，可以将其线程 ID 重新分配给另一个新线程。

非分离线程在终止后，必须要有一个线程用 join 来等待它。否则，不会释放该线程的资源以供新线程使用，而这通常会导致内存泄漏。因此，如果不希望线程被等待，请将该线程作为分离线程来创建。

示例 3–1 创建分离线程

```
#include <pthread.h>



pthread_attr_t tattr;

pthread_t tid;

void *start_routine;
```

示例 3–1 创建分离线程　　（续）

```
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);
```

pthread_attr_setdetachstate 返回值

pthread_attr_setdetachstate() 成功完成后将返回零。其他任何返回值都表示出现了错误。如果出现以下情况，该函数将失败并返回对应的值。

EINVAL
描述: 指示 *detachstate* 或 *tattr* 的值无效。

获取分离状态

请使用 pthread_attr_getdetachstate(3C) 检索线程创建状态（可以为分离或连接）。

pthread_attr_getdetachstate 语法

```
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);
```

pthread_attr_getdetachstate 返回值

pthread_attr_getdetachstate() 成功完成后将返回零。其他任何返回值都表示出现了错误。如果出现以下情况，该函数将失败并返回对应的值。

EINVAL
描述: 指示 *detachstate* 的值为 NULL 或 *tattr* 无效。

设置栈溢出保护区大小

pthread_attr_setguardsize(3C) 可以设置 *attr* 对象的 *guardsize*。

pthread_attr_setguardsize(3C) 语法

```
#include <pthread.h>


int pthread_attr_setguardsize(pthread_attr_t *attr, size_t  guardsize);
```

出于以下两个原因，为应用程序提供了 guardsize 属性：

- 溢出保护可能会导致系统资源浪费。如果应用程序创建大量线程，并且已知这些线程永远不会溢出其栈，则可以关闭溢出保护区。通过关闭溢出保护区，可以节省系统资源。
- 线程在栈上分配大型数据结构时，可能需要较大的溢出保护区来检测栈溢出。

guardsize 参数提供了对栈指针溢出的保护。如果创建线程的栈时使用了保护功能，则实现会在栈的溢出端分配额外内存。此额外内存的作用与缓冲区一样，可以防止栈指针的栈溢出。如果应用程序溢出到此缓冲区中，这个错误可能会导致 SIGSEGV 信号被发送给该线程。

如果 *guardsize* 为零，则不会为使用 *attr* 创建的线程提供溢出保护区。如果 *guardsize* 大于零，则会为每个使用 *attr* 创建的线程提供大小至少为 *guardsize* 字节的溢出保护区。缺省情况下，线程具有实现定义的非零溢出保护区。

允许合乎惯例的实现，将 *guardsize* 的值向上舍入为可配置的系统变量 PAGESIZE 的倍数。请参见 sys/mman.h 中的 PAGESIZE。如果实现将 *guardsize* 的值向上舍入为 PAGESIZE 的倍数，则以 *guardsize*（先前调用 pthread_attr_setguardsize() 时指定的溢出保护区大小）为单位存储对指定 *attr* 的 pthread_attr_getguardsize() 的调用。

pthread_attr_setguardsize 返回值

如果出现以下情况，pthread_attr_setguardsize() 将失败：

EINVAL
描述: 参数 *attr* 无效，参数 *guardsize* 无效，或参数 *guardsize* 包含无效值。

获取栈溢出保护区大小

pthread_attr_getguardsize(3C) 可以获取 *attr* 对象的 *guardsize*。

pthread_attr_getguardsize 语法

```
#include <pthread.h>



int pthread_attr_getguardsize(const pthread_attr_t *attr,

                                        size_t  *guardsize);
```

允许一致的实现将 *guardsize* 中包含的值向上舍入为可配置系统变量 PAGESIZE 的倍数。请参见 sys/mman.h 中的 PAGESIZE。如果实现将 *guardsize* 的值向上舍入为 PAGESIZE 的倍数，则以 *guardsize*（先前调用 pthread_attr_setguardsize() 时指定的溢出保护区大小）为单位存储对指定 *attr* 的 pthread_attr_getguardsize() 的调用。

pthread_attr_getguardsize 返回值

如果出现以下情况，pthread_attr_getguardsize() 将失败：

EINVAL
描述: 参数 *attr* 无效，参数 *guardsize* 无效，或参数 *guardsize* 包含无效值。

设置范围

请使用 pthread_attr_setscope(3C) 建立线程的争用范围（PTHREAD_SCOPE_SYSTEM 或 PTHREAD_SCOPE_PROCESS）。使用 PTHREAD_SCOPE_SYSTEM 时，此线程将与系统中的所有线程进行竞争。使用 PTHREAD_SCOPE_PROCESS 时，此线程将与进程中的其他线程进行竞争。

注 – 只有在给定进程中才能访问这两种线程类型。

---

pthread_attr_setscope 语法

```
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);
```

本示例使用三个函数调用：用于初始化属性的调用、用于根据缺省属性设置所有变体的调用，以及用于创建 pthreads 的调用。

```
#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);
```

pthread_attr_setscope 返回值

pthread_attr_setscope() **成功**完成后将返回零。其他任何返回值都表示出现了错误。如果出现以下情况，该函数将失败并返回对应的值。

EINVAL
描述: 尝试将 *tattr* 设置为无效的值。

获取范围

请使用 pthread_attr_getscope(3C) 检索线程范围。

pthread_attr_getscope 语法

```
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);
```

pthread_attr_getscope 返回值

pthread_attr_getscope() 成功完成后将返回零。其他任何返回值都表示出现了错误。如果出现以下情况，该函数将失败并返回对应的值。

EINVAL
描述: *scope* 的值为 NULL 或 *tattr* 无效。

设置线程并行级别

针对标准符合性提供了 pthread_setconcurrency(3C)。应用程序使用 pthread_setconcurrency() 通知系统其所需的并发级别。对于 Solaris 9 发行版中引入的线程实现，此接口没有任何作用，所有可运行的线程都将被连接到 LWP。

pthread_setconcurrency 语法

```
#include <pthread.h>


int pthread_setconcurrency(int new_level);
```

pthread_setconcurrency 返回值

如果出现以下情况，pthread_setconcurrency() 将失败：

EINVAL
描述: *new_level* 指定的值为负数。

EAGAIN
描述: *new_level* 指定的值将导致系统资源不足。

获取线程并行级别

pthread_getconcurrency(3C) 返回先前调用 pthread_setconcurrency() 时设置的值。

pthread_getconcurrency 语法

```
#include <pthread.h>


int pthread_getconcurrency(void);
```

如果以前未调用 pthread_setconcurrency() 函数，则 pthread_getconcurrency() 将返回零。

pthread_getconcurrency 返回值

pthread_getconcurrency() 始终会返回先前调用 pthread_setconcurrency() 时设置的并发级别。如果从未调用 pthread_setconcurrency()，则 pthread_getconcurrency() 将返回零。

设置调度策略

请使用 pthread_attr_setschedpolicy(3C) 设置调度策略。POSIX 标准指定 SCHED_FIFO（先入先出）、SCHED_RR（循环）或 SCHED_OTHER（实现定义的方法）的调度策略属性。

pthread_attr_setschedpolicy(3C) 语法

```
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);
```

- SCHED_FIFO

  如果调用进程具有有效的用户 ID 0，则争用范围为系统 (PTHREAD_SCOPE_SYSTEM) 的先入先出线程属于实时 (RT) 调度类。如果这些线程未被优先级更高的线程抢占，则会继续处理该线程，直到该线程放弃或阻塞为止。对于具有进程争用范围 (PTHREAD_SCOPE_PROCESS)) 的线程或其调用进程没有有效用户 ID 0 的线程，请使用 SCHED_FIFO。SCHED_FIFO 基于 TS 调度类。

- SCHED_RR

  如果调用进程具有有效的用户 ID 0，则争用范围为系统 (PTHREAD_SCOPE_SYSTEM)) 的循环线程属于实时 (RT) 调度类。如果这些线程未被优先级更高的线程抢占，并且这些线程没有放弃或阻塞，则在系统确定的时间段内将一直执行这些线程。对于具有进程争用范围 (PTHREAD_SCOPE_PROCESS) 的线程，请使用 SCHED_RR（基于 TS 调度类）。此外，这些线程的调用进程没有有效的用户 ID 0。

SCHED_FIFO 和 SCHED_RR 在 POSIX 标准中是可选的，而且仅用于实时线程。

有关调度的论述，请参见第 19 页中的 “线程调度”一节。

pthread_attr_setschedpolicy 返回值

pthread_attr_setschedpolicy() 成功完成后将返回零。其他任何返回值都表示出现了错误。如果出现以下任一情况，该函数将失败并返回对应的值。

EINVAL
: **描述:** 尝试将 *tattr* 设置为无效的值。

ENOTSUP
: **描述:** 尝试将该属性设置为不受支持的值。

获取调度策略

请使用 pthread_attr_getschedpolicy(3C) 检索调度策略。

pthread_attr_getschedpolicy 语法

```
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);
```

pthread_attr_getschedpolicy 返回值

pthread_attr_getschedpolicy() 成功完成后将返回零。其他任何返回值都表示出现了错误。如果出现以下情况，该函数将失败并返回对应的值。

EINVAL
描述: 参数 *policy* 为 NULL 或 *tattr* 无效。

设置继承的调度策略

请使用 pthread_attr_setinheritsched(3C) 设置继承的调度策略。

pthread_attr_setinheritsched 语法

```
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);
```

inherit 值 PTHREAD_INHERIT_SCHED 表示新建的线程将继承创建者线程中定义的调度策略。将忽略在 pthread_create() 调用中定义的所有调度属性。如果使用缺省值 PTHREAD_EXPLICIT_SCHED，则将使用 pthread_create() 调用中的属性。

pthread_attr_setinheritsched 返回值

pthread_attr_setinheritsched() 成功完成后将返回零。其他任何返回值都表示出现了错误。如果出现以下任一情况，该函数将失败并返回对应的值。

EINVAL
描述: 尝试将 *tattr* 设置为无效的值。

ENOTSUP
描述: 尝试将属性设置为不受支持的值。

获取继承的调度策略

pthread_attr_getinheritsched(3C) 将返回由 pthread_attr_setinheritsched() 设置的调度策略。

pthread_attr_getinheritsched 语法

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

#include <pthread.h>



pthread_attr_t tattr;

int inherit;

int ret;



/* get scheduling policy and priority of the creating thread */

ret = pthread_attr_getinheritsched (&tattr, &inherit);
```

pthread_attr_getinheritsched 返回值

pthread_attr_getinheritsched() 成功完成后将返回零。其他任何返回值都表示出现了错误。如果出现以下情况，该函数将失败并返回对应的值。

EINVAL
描述: 参数 *inherit* 为 NULL 或 *tattr* 无效。

设置调度参数

pthread_attr_setschedparam(3C) 可以设置调度参数。

pthread_attr_setschedparam 语法

```
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);
```

调度参数是在 param 结构中定义的。仅支持优先级参数。新创建的线程使用此优先级运行。

pthread_attr_setschedparam 返回值

pthread_attr_setschedparam() 成功完成后将返回零。其他任何返回值都表示出现了错误。如果出现以下情况，该函数将失败并返回对应的值。

EINVAL
描述: *param* 的值为 NULL 或 *tattr* 无效。

可以采用两种方式之一来管理 pthreads 优先级：

- 创建子线程之前，可以设置优先级属性
- 可以更改父线程的优先级，然后再将该优先级改回来

获取调度参数

pthread_attr_getschedparam(3C) 将返回由 pthread_attr_setschedparam() 定义的调度参数。

pthread_attr_getschedparam 语法

```
int    pthread_attr_getschedparam(pthread_attr_t *tattr,

    const struct sched_param *param);

#include <pthread.h>



pthread_attr_t attr;

struct sched_param param;

int ret;



/* get the existing scheduling param */

ret = pthread_attr_getschedparam (&tattr, &param);
```

使用指定的优先级创建线程

创建线程之前，可以设置优先级属性。将使用在 sched_param 结构中指定的新优先级创建子线程。此结构还包含其他调度信息。

创建子线程时建议执行以下操作：

- 获取现有参数

- 更改优先级
- 创建子线程
- 恢复原始优先级

创建具有优先级的线程的示例

示例 3–2 给出了使用不同于其父线程优先级的优先级创建子线程的示例。

示例 3–2 创建具有优先级的线程

```
#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 */
```

示例 3–2 创建具有优先级的线程　　（续）

```
ret = pthread_attr_setschedparam (&tattr, &param);



/* with new priority specified */

ret = pthread_create (&tid, &tattr, func, arg);
```

pthread_attr_getschedparam 返回值

pthread_attr_getschedparam() 成功完成后将返回零。其他任何返回值都表示出现了错误。如果出现以下情况，该函数将失败并返回对应的值。

EINVAL
描述: *param* 的值为 NULL 或 *tattr* 无效。

设置栈大小

pthread_attr_setstacksize(3C) 可以设置线程栈大小。

pthread_attr_setstacksize 语法

```
int    pthread_attr_setstacksize(pthread_attr_t *tattr,

                                 size_t size);

#include <pthread.h>



pthread_attr_t tattr;

size_t size;

int ret;



size = (PTHREAD_STACK_MIN + 0x4000);



/* setting a new size */

ret = pthread_attr_setstacksize(&tattr, size);
```

stacksize 属性定义系统分配的栈大小（以字节为单位）。*size* 不应小于系统定义的最小栈大小。有关更多信息，请参见第 67 页中的 “关于栈”。

size 包含新线程使用的栈的字节数。如果 *size* 为零，则使用缺省大小。在大多数情况下，零值最适合。

PTHREAD_STACK_MIN 是启动线程所需的栈空间量。此栈空间没有考虑执行应用程序代码所需的线程例程要求。

pthread_attr_setstacksize 返回值

pthread_attr_setstacksize() 成功完成后将返回零。其他任何返回值都表示出现了错误。如果出现以下情况，该函数将失败并返回对应的值。

EINVAL
描述: *size* 值小于 PTHREAD_STACK_MIN，或超出了系统强加的限制，或者 *tattr* 无效。

获取栈大小

pthread_attr_getstacksize(3C) 将返回由 pthread_attr_setstacksize() 设置的栈大小。

pthread_attr_getstacksize 语法

```
int    pthread_attr_getstacksize(pthread_attr_t *tattr,
                                 size_t *size);

#include <pthread.h>



pthread_attr_t tattr;

size_t size;

int ret;



/* getting the stack size */

ret = pthread_attr_getstacksize(&tattr, &size);
```

pthread_attr_getstacksize 返回值

pthread_attr_getstacksize() 成功完成后将返回零。其他任何返回值都表示出现了错误。如果出现以下情况，该函数将失败并返回对应的值。

EINVAL
描述: *tattr* 无效。

关于栈

通常，线程栈是从页边界开始的。任何指定的大小都被向上舍入到下一个页边界。不具备访问权限的页将被附加到栈的溢出端。大多数栈溢出都会导致将 SIGSEGV 信号发送到违例线程。将直接使用调用方分配的线程栈，而不进行修改。

指定栈时，还应使用 PTHREAD_CREATE_JOINABLE 创建线程。在该线程的 pthread_join(3C) 调用返回之前，不会释放该栈。在该线程终止之前，不会释放该线程的栈。了解这类线程是否已终止的唯一可靠方式是使用 pthread_join(3C)。

为线程分配栈空间

一般情况下，不需要为线程分配栈空间。系统会为每个线程的栈分配 1 MB（对于 32 位系统）或 2 MB（对于 64 位系统）的虚拟内存，而不保留任何交换空间。系统将使用 mmap() 的 MAP_NORESERVE 选项来进行分配。

系统创建的每个线程栈都具有红色区域。系统通过将页附加到栈的溢出端来创建红色区域，从而捕获栈溢出。此类页无效，而且会导致内存（访问时）故障。红色区域将被附加到所有自动分配的栈，无论大小是由应用程序指定，还是使用缺省大小。

注 – 对于库调用和动态链接，运行时栈要求有所变化。应绝对确定，指定的栈满足库调用和动态链接的运行时要求。

极少数情况下需要指定栈和/或栈大小。甚至专家也很难了解是否指定了正确的大小。甚至符合 ABI 标准的程序也不能静态确定其栈大小。栈大小取决于执行中特定运行时环境的需要。

生成自己的栈

指定线程栈大小时，必须考虑被调用函数以及每个要调用的后续函数的分配需求。需要考虑的因素应包括调用序列需求、局部变量和信息结构。

有时，您需要与缺省栈略有不同的栈。典型的情况是，线程需要的栈大小大于缺省栈大小。而不太典型的情况是，缺省大小太大。您可能正在使用不足的虚拟内存创建数千个线程，进而处理数千个缺省线程栈所需的数千兆字节的栈空间。

对栈的最大大小的限制通常较为明显，但对其最小大小的限制如何呢？必须存在足够的栈空间来处理推入栈的所有栈帧，及其局部变量等。

要获取对栈大小的绝对最小限制，请调用宏 PTHREAD_STACK_MIN。PTHREAD_STACK_MIN 宏将针对执行 NULL 过程的线程返回所需的栈空间量。有用的线程所需的栈大小大于最小栈大小，因此缩小栈大小时应非常谨慎。

```
#include <pthread.h>


pthread_attr_t tattr;

pthread_t tid;

int ret;


size_t 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);
```

设置栈地址和大小

pthread_attr_setstack(3C) 可以设置线程栈地址和大小。

pthread_attr_setstack(3C) 语法

```
int    pthread_attr_setstack(pthread_attr_t *tattr,void *stackaddr,

                             size_t stacksize);
```

```
#include <pthread.h>



pthread_attr_t tattr;

void *base;

size_t size;

int ret;



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



/* setting a new address and size */

ret = pthread_attr_setstack(&tattr, base,PTHREAD_STACK_MIN + 0x4000);
```

stackaddr 属性定义线程栈的基准（低位地址）。*stacksize* 属性指定栈的大小。如果将 *stackaddr* 设置为非空值，而不是缺省的 NULL，则系统将在该地址初始化栈，假设大小为 *stacksize*。

base 包含新线程使用的栈的地址。如果 *base* 为 NULL，则 pthread_create(3C) 将为大小至少为 *stacksize* 字节的新线程分配栈。

pthread_attr_setstack(3C) 返回值

pthread_attr_setstack() 成功完成后将返回零。其他任何返回值都表示出现了错误。如果出现以下情况，该函数将失败并返回对应的值。

EINVAL
描述: *base* 或 *tattr* 的值不正确。*stacksize* 的值小于 PTHREAD_STACK_MIN。

以下示例说明如何使用自定义栈地址和大小来创建线程。

```
#include <pthread.h>



pthread_attr_t tattr;

pthread_t tid;

int ret;
```

```
void *stackbase;

size_t size;


/* initialized with default attributes */

ret = pthread_attr_init(&tattr);


/* setting the base address and size of the stack */

ret = pthread_attr_setstack(&tattr, stackbase,size);


/* address and size specified */

ret = pthread_create(&tid, &tattr, func, arg);
```

获取栈地址和大小

pthread_attr_getstack(3C) 将返回由 pthread_attr_setstack() 设置的线程栈地址和大小。

pthread_attr_getstack 语法

```
int    pthread_attr_getstack(pthread_attr_t *tattr,void * *stackaddr,

                             size_t *stacksize);

#include <pthread.h>


pthread_attr_t tattr;

void *base;

size_t size;

int ret;
```

```
/* getting a stack address and size */

ret = pthread_attr_getstackaddr (&tattr, &base, &size);
```

pthread_attr_getstack返回值

`pthread_attr_getstackaddr()`成功完成后将返回零。其他任何返回值都表示出现了错误。如果出现以下情况，该函数将失败并返回对应的值。

`EINVAL`
描述: *tattr*的值不正确。