多线程编程技术

线程是比进程更小的单位，可以认为进程是由一个或多个线程组成的。据说以前的 400 版本并不支持真正的多线程技术，在 4.2 版后才从内核上提供了对多线程的支持。总之写这份文档的时候，绝大部分版本应该可以支持。

主要资料来源于 IBM 信息中心的《Programming Multithreaded applications》，加上部分个人观点。

多线程编程有以下几点特殊性（说好听点叫特殊性，说得直接一点完全就是为什么不使用多线程技术的理由，按理由的充分性，由小到大排列）：

1. 多线程的编程在对 C 程序的使用上要特别小心，详细内容见调用 C 程序的注意事项。

2. 事务处理的作用范围是 JOB 级，或活动作业组级。这也就是说多线程并发时，一个线程的 COMMIT 操作可能导致另一个线程也执行了 COMMIT 操作。（这是 IBM 说的，不是我的猜想）。所以实际上也就可以认为多线程的并发不能支持事务操作。

3. 对于用户的应用程序来说，大部分 C、RPG、RPGLE 编写的 PGM 和 MODULE 都不具备线程安全性，也就是不能被一个进程下的多个线程同时调用，要注意。（不排除是因为某些参数未掌握好，总之目前测试的结果就是如此）同时也基于此，多线程之下程序的复用性，维护性就没有单线程下那么方便和自由。

4. 最后，实际测试多线程的效率，只能用令人惊讶来形容 -- 和多进程并发效率居然几乎是一样的！-_- （程序写起来还麻烦得要死！）

   测试方法：

   读一个 380 多万条记录的文件，再根据某个键值去 CHAIN 另一个 620 多万条记录的文件，仅此而已。

   根据这个 380 多万条记录的文件的某个关键字，拆分成 200 多个任务，每个任务只处理自己需要处理的数据。（可以简单的认为每个任务是处理 380w/210 条记录吧，其实并非如此）

   多进程分为 10 个进程来处理这 200 多个任务，耗时约 3 分钟（14:40:45 – 14:43:59）。

   多线程分为 8 个线程来处理这 200 多个任务，耗时竟然也是约 3 分钟（14:47:03 – 14:50:14），严格的来说，少了 3 秒钟，我估计这 3 秒就是启动 8 个线程比启动 10 个 JOB 要少的时间吧。我在多线程里还特地增加了共享了 ODP 的处理，居然还是只有这个效果。

看到这里，如果对多线程技术还有兴趣的话，那就请继续往下看吧。我要早知道，也就不写这么多了。

目 录

1 概念

1.1 JOB

JOB，包含了存储和其它资源这两方面的内容，JOB 自身并不能运行。(A *job* is a container for storage and other resources，and it cannot run by itself)

存储，是指数据和堆栈 (Data and Stack)

其它资源， 包括环境变量、地址、文件描述、该 JOB 所打开的文件信息、当前工作目录。

绝大部分 400 上的操作管理命令都是基于 JOB 的。

JOB 的概念我们平时接触得很多。例如说在交互式作业中，用户登录之后，即是启动了一个 JOB，系统就需要为其分配相应的资源。我们可以通过 WRKACTJOB 命令查看活动的作业。而由 USER、JOBNAME、JOBID 可以定位到唯一的一个 JOB，可以使用各种系统命令，来取得指定 JOB 当前的状态（比如说 ACTIVE、MSGW、TIMW 等），或者是 JOB 的信息（RTVJOBA），也可以使用各种系统命令来操作 JOB（HOLD，END 等）。对 JOB 的操作、管理都很直观方便。

1.2 Process (进程)

进程，是管理程序运行的资源。(Process is container of the memory and resoures of the program)。

在 400 系统中，一个 JOB 就代表了一个进程（On IBM system i platforms, a job represent a process）

按照我的理解，启动一个 JOB，相当于系统就分配了相应的资源给这个 JOB，JOB 自身并不负责运行程序；进程则是用来负责管理这个 JOB 下运行的程序（这里的程序应该并不专指可以 CALL 的程序，我觉得所有的人机交互应该都是通过程序来完成的，即都是由进程来管理的）。按照上文所说，一个 JOB 有且仅有一个进程。所以我个人认为在理解上，似乎没有太大必要将进程与 JOB 刻意区分开来。一个进程可以对应多个运行中的程序（比如程序 A 调用程序 B，那么当前进程就同时管理着程序 A、B 的资源。也正是基于这种原理，所以 400 平台上的程序调用，允许有返回数据）

每个进程至少拥有一个线程（任务）来执行程序。

进程是基于 JOB 的，我们平时所说的多进程并发，实质上指的也就是多 JOB 并发。

多 JOB 并发的原理较为简单，就是使用 SBMJOB 命令提交作业来实现并发处理。通常为了便于管理，会建立专用的子系统，以及专用的 JOBD、JOBQ（这里的专用，只是从应用级别而言，建立的方法并没有特殊的地方），也就是说，需要启动多个 JOB。

1.3 Thread (线程)

线程是程序运行的通道，操作系统通过线程，按照一定顺序去逐步执行程序。（A *thread* is the path taken by a program while running, the steps performed, and the order in which the steps are performed.）

所有的程序都拥有至少一个线程，对于多线程的程序而言，每一个线程都是独立于其它的

线程运行的。

进程中运行的第一个线程，称为初始线程（Initial thread）；

并不是所有的 JOB 都支持多线程（实际上，大部分系统应用中，默认的 JOBD 的属性都是不支持多线程）。当进程对应的 Job 支持多线程时，该进程下的其它的非初始线程的线程，称为辅线程（Secondary thread）。

在实际操作中，我们可以通过 WRKJOB + 20，来查看线程的状态以及相关的线程信息。

1.3.1 线程的分类

线程可以分为用户线程和核心线程。

按照用户线程的分类，一个进程之下的所有程序线程都共享同一个进程线程；线程的 API 函数执行任务计划（scheduling policy），判定何时运行新的线程。任务计划决定了在一个时点上，一个进程只能有一个活动的线程。(The scheduling policy allows only one thread to be actively running in the process at a time)。而操作系统只需要关注这个进程单一的任务（Single Task）。

按照核心线程的分类，进程被拆分成若干个独立的任务，一个进程对应的多个核心线程各自处理这些独立的任务。核心线程使用优先的任务计划（preemptive scheduling policy，这里的优先，应该是相对于用户线程而言），操作系统通过这个优先的任务计划，判定当前由哪一个核心线程来使用处理器资源。i5 操作系统支持核心线程的处理方式。

这两种分类是独立开来理解的，也就是说当前的系统平台可能只支持用户线程，而不支持核心线程。如果不支持核心线程的话，那么一个进程之上即使并发了多个用户线程，对应于操作系统而言，仍然只是一个单一的任务，也就是前文所说的“操作系统只需要关注的单一的任务”（Single task）；在这种情况下，表面上并发的用户线程，在操作系统层面其实仍然是串行的（通过线程的 API 函数来分配任务），这也就是前文所说的“同一时点上，一个进程只能有一个活动的线程”的真正含义。（这段话是我个人的理解，仅供参考）

基于系统设计时的向下兼容原理，支持核心线程的平台，也一定会支持用户线程。这时的操作系统就同时支持了用户线程和核心线程。（事实上我想目前大部分的 400 版本都应该是这种类型）

通常，我们使用 M*N 的方式来表达这种类型：一个进程上运行着 M 个用户线程，这 M 个用户线程共享该进程对应系统底层的 N 个核心线程。

用户线程位于核心线程的上层，可由我们用户控制的；核心线程是由系统控制的，对用户是不可见的。系统只对更加昂贵的核心线程分配资源。用户线程到核心线程的解析由系统来完成。

1.3.2 线程的程序模型

400 支持 call-return 的程序模型。在其它的平台中，如果程序 A 想用调用程序 B，那么系统就必须再启动一个进程来运行这个程序 B，或者是在当前进程中用程序 B 来代替程序 A。而 400 的进程管理机制，决定了一个进程可以对应多个程序，这多个程序共享相同的进程资源，所以可以很轻易的实现 call --return 这种方式。

这里稍微说明一下，所谓 CALL –return 模型，也就是说程序 A 调用程序 B，程序 B 的返回值，也就是我们所说的接口数据，可以在程序 A 中直接使用。举个常见的与之相反的例子，在 C 语言中，程序的调用就只有输入，没有返回。（函数才有返回，两个 main 函数之前的信息传递是单向的）

启动另一个进程的需要耗费时间，以及系统资源。为了避免这种消耗，程序员们通常会使用动态链接库（DLL）。每当程序需要使用到动态链接库中的服务时，程序就载入动态链接库，然后调用函数，来实现程序所需要的服务。

尽管 400 的 I5 平台中，多线程程序支持 call – return 的程序模型，但是 IBM 公司还是强烈建议我们在调用的活动作业组中使用服务程序（service program），或动态链接态。这种建议主要是基于程序应用跨平台时的考虑。

虽然不是必须，但多线程编程所调用的程序最好是 ILE 环境下的程序，而不是 OPM 程序（也就是不建议使用 RPG，尽量用 RPGLE）。因为 OPM 程序在多线程程序中，需要注意到一些特殊的地方，比如说线程的安全性。

1.3.3 JOB 和 JOB 资源

前面已经提到过，JOB 包含了存储和其它资源。存储（storage）又包括了数据和堆栈

数据：

数据是指存放程序变量的地点。具体而言，可分为三类：

全局变量和静态变量（简称 static）

动态分配的存储（简称 heap）、

本地函数变量（简称 automatic）

程序变量的存储空间由活动作业组（activation group）分配，运行中的程序信息都保存在活动作业组中。

所有运行在同一个活动作业组中的线程都可以共享使用 static 和 heap；

而当前程序的变量，以及 automatic 则由按当前线程分配使用，线程之间不做共享。（也就是说各个线程原则上互不干扰）

堆栈：

堆栈包含线程中调用的程序流或过程流的数据（The stack contains data about the program or procedure call flow in a thread）

当建立一个线程的时候，系统分配堆栈，以及随机自动分配的存储空间。使用一个线程的时候，堆栈以及随机自动分配的存储空间都被视为线程资源。当该线程结束时，这些资源将会返回给进程，由进程再分配给之后启动的其它线程去使用。

一个活动作业组下的所有线程，都共享这个活动作业组中的资源，比如 static, heap（当然，这个活动作业组中的资源也是属于 JOB 的资源）。所以当一个线程更改了这类 JOB 资源时，其它线程查询到的，将会更改后的值。

1.3.4 线程的私有数据和特有数据

有些数据资源，虽然在程序中定义为全局变量，但线程间不能共享，而是由每个线程自己为这些数据资源分配存储空间。这类数据称之为线程的特有数据 thread-specific data。

线程之间之间完全独立的数据，称为线程的私有数据 thread –private data.

(Threads cannot share certain resources, but they can have their own view of data items called thread-specific data. Data that threads cannot share between themselves are called thread-private data.)

以下这几种资源都是属于线程的私有数据：

线程标识符（系统唯一）
线程优先级（默认与 JOB 的相关）
Security information
库列表
signal blocking mask
Call stack
Automatic storage
错误信息
这个是指系统自带的 errno

有关线程的特有数据，详见线程的特有数据(thread specific data)

1.3.5 多线程编程的环境

交互式作业不支持多线程（还有一种通讯类的作业也不支持多线程？），所以要使用、测试多线程必须在子系统下，比如用 SBMJOB 提交。

同时，提交到 JOB 时指定的 JOBD 要支持多线程，附带再说明一下，使用 SBMJOB 命令时，如果不指定 JOBD，那么就默认使用当前 USER 的 JOBD。

可用 WRKJOBD 查看使用的 JOBD 的 **Allow Multithread (ALWMLTTHD)**这个参数（排位比较靠后）， 该参数值为 YES 时表示该 JOBD 支持多线程，为 NO 时表示不支持多线程。该值可用 CHGJOBD 命令来更改，或在创建 JOBD 时指定。

1.4 活动作业组（Activation group）

1.4.1 基本资料

一个 JOB 下， 容纳活动中的程序、服务程序的子结构被称为活动作业组（activation group），活动作业组包含了运行程序所必须的资源，这些资源是指：

static, heap 以及管理临时数据的资源 (static, heap 是简称，具体解释详见上文的 JOB 和 JOB 资源);

活动作业组的作用范围与程序编译时的参数相关，一个 JOB 就有可能对应多个活动作业组（详见下面的注意事项）。

一个 JOB 如果支持多线程编程的话，那么这个 JOB 下的多个线程将可以共享这个 JOB 下的同一个活动作业组中的资源；

而一个线程可以运行不同的活动作业组中被激活的程序。

系统不会保存线程与活动作业组之间对应关系表，也就是说我们不知道一个活动作业组中运行了哪些线程，也不知道一个线程对应哪些活动作业组。

所以当一个活动作业组中还有线程在运行就将其关闭的话，系统无法进行检测，只能按照指令强制关闭该活动作业组，此时就可能会产生无法预见的错误，比如说进程非正常中断。

为了避免这种情况，在辅线程中所有关闭活动作业组的操作，都会导致系统采用有序的方式 end 掉 JOB（也就是说这种情况发生时，系统为了避免非正常中断，就主动正常中断）。

所谓的有序的中断，也就是系统在 JOB 中 End 掉所有线程，然后在初始线程中调用 exit 事务，最后关闭掉所有文件。

1.4.2 注意事项

程序编译时，如果 ACTGRP 参数是“*NEW”的话，那么每次调用这个程序，都会产生一个新的活动作业组；当程序执行 RETURN 语句时，系统将会关闭这个新的活动作业组。也就是说，程序如果是这样编译的话，在辅线程中程序的 Return 将会导致 JOB 的 End。

进一步解释，应用系统中，如果程序编译时，ACTGRP 参数使用了“NEW”；而在程序结尾又使用了 Return 语句的话，那么辅线程中程序的结束将会导致整个 JOB（含初始线程）都 End 掉，这往往与我们的预想有偏差，所以在使用时要加以注意。不过我们通常在编译程序时都会直接使用系统的默认参数（QILE），一般也就不会出现这个错误。

还有一种情况也会导致 Job 的 End，不过我翻译不出来：

If an exception has not been handled by the time it has percolated to the control boundary and the control boundary is a program entry procedure (PEP or main entry point), the multithread-capable job is ended.

然后，使用 RCLACTGRP 这个命令时（看这个名字就知道，是回收活动作业组资源的），只允许初始线程使用，如果是辅线程使用的话，系统将会报一个 CPF180B 的错。

1.4.3 编译时的参数

如果编译程序时，ACTGRP 参数采用系统默认参数(QILE)，或者是 named 的参数，那么程序 Return 后，活动作业组仍会保留。

默认参数（QILE）是指在启动 Job 时，就产生活动作业组；在 End Job 的时候，销毁活动作业组；

Named 参数是指在 Job 中，首次调用程序时产生活动作业组；当程序 Return 之后，这个活动作业组将会处于一个 last-used 的状态，但不会被删除掉。

1.4.4 几点疑惑

不知道编译程序时，ACTGRP 的参数又不会影响到调用程序时的版本，也就是如果不是每次 Return 就销毁掉活动作业组，程序更新而又没有 end 掉 JOB 的时候，这个 Job 有没有可能还是调用活动作业组所指向的原来旧版本的程序？如果旧版本的程序在内存中未清除的话，会不会有可能产生更新了程序但仍然执行的是旧版本的问题？

如果是 C 程序，用 linkage 绑定 RPG 程序来调用的话，会不会即使编译程序时使用了“*NEW”参数，也还是会有更新程序执行旧版本的问题？

还有，如果程序中没有 Return 语句的话，那么是否程序结束活动作业组也不会销毁？

RPG 程序在编译时没有这个参数，是不是活动作业组的概念并不针对于 RPG 程序？

很多事情都可能有关联的。

1.5 调用 C 程序的注意事项

C 程序的调用有一定的特殊性，系统隐含了一种可能会导致提前结束线程，或是整个 JOB 的情况。就应用级程序而言，这种情况出现的频率较上文提到的情况更为常见，所以特别在此提示：

假设线程中最外层的程序 A，调用了其它的程序 B、C、D；

如果 B、C、D 是 C 程序，或它们自身又调用了 C 程序（这里的调用 C 程序，是指调用 main()函数），当这个被调用的 C 程序结束时，系统会有一个隐含的 pthread_exit()，或 exit()，这个隐含的操作在当前 JOBD 支持多线程时，将会结束当前的线程；如果当前线程是初始线程，那还将会结束当前的 JOB。如果当前 JOBD 不支持多线程，则不会这个问题。

如果 B、C、D 是 RPGLE 程序，它们没有调用 C 程序，只是调用了 C 函数，具体来说，就是通过将 C 程序编译为 MODULE，RPGLE 程序也编译为 MODULE，然后在 RPGLE 程序中使用 CALLB 的方式来调用 C 函数，最后使用 CRTPGM 来生成 PGM。在这种情况下，RPGLE 程序的结束（并不是 C 函数的结束），也将会有一个隐含的 pthread_exit()或 exit()，如果当前 JOBD 支持多线程，会结束当前线程；如果当前线程是初始线程，会结束当前 JOB。如果 JOBD 不支持多线程，不会存在上述问题。

附带再说明一下 C 程序中 exit()与 return()的区别：

return 是返回上层调用，exit 是结束当前程序，然后再返回上层调用。也就是 exit 有一个结束当前程序的动作。

如果 JOBD 支持多线程，那么 C 函数中的 exit 还将会导致线程结束，无论调用 C 函数的程序是否位于最上层调用。如果当前线程为初始线程，会结束当前 JOB。

如果 JOBD 不支持多线程，C 函数中的 exit 只会导致当前程序结束（不仅限于当前调用的 C 函数，而是当前程序），但仍会返回上层调用程序，不会结束线程。当然，如果当前程序已位于最上层调用，那么还是会结束线程的。

无论 JOBD 是否支持多线程，如果是采用调用 C 函数而不是调用 C 程序的方式，return 的使用都只会结束当前 C 函数，返回上层调用它的程序或 Module，不会导致程序结束，也不会导致线程结束。但是在多线程环境中，当前调用了 C 函数的程序结束，仍会导致线程的结束，无论是否还有上层调用，参见上文描述。

也就是说，如果使用多线程技术，那么：

1、C 程序的 main()函数只允许在最外层使用，不能被任何程序调用；

2、调用了 C 函数的 RPGLE 程序，尽量位于当前线程的最外层，或是调用它的程序在调用之后不再进行其它处理，否则会造成应用级别的异常中断；

3、如非必要，C 函数中尽量使用 return， 不使用 exit。

如果 JOBD 不支持多线程，就不会有这些问题；所以应用系统如果要从单线程转换成为多线程，那么在 C 程序或 C 函数的调用上需要特别留意。

1.6 多线程编程的通讯问题

i5 支持的唯一的一种能保障线程安全的通讯协议，就是 socket 协议（也就是说 SNA，ODBC 之类的跨平台数据交互都不能使用多线程编程？）

在 400 上使用 socket 需要考虑到以下两点：

SOCKET API：

大部分 socket 接口都能保障线程安全，但是绝大多数网络路由器不能使用静态存储空间。那么我们通讯时使用的函数可能需要加上_r 的后缀，比如说，原 gethostbyaddr()，就需要改为 gethostbyaddr_r()。这类带_r 后缀的程序，与 UNIX 定义的是兼容的。所有带_r 后缀的程序都存在于服务程序 QSOSRV2 中。

AnyNet：

在多线程程序中，AnyNet 也可以支持线程安全，但是未经测试。

1.7 线程的数据库、数据相关处理

数据库操作：

支持线程安全的数据库操作包括了：创建文件，增加 MEMBER，删除文件，删除 MEMBER。我们可以使用 DSPCMD 命令来查看当前环境下的这些命令是否支持线程安全

就用户层面来看，线程与线程间的数据操作，同样是记录锁。

比如说线程 1 执行了一个读操作，线程 2 如果也要对该记录执行操作时（operation against the same open instance），线程 2 将会等待线程 1 结束读操作。读出的结果放在 I/O 缓冲里。

如果线程操作的是不同的记录（我觉得对于普通用户而言，open instance 指的多半就是打开文件操作某条记录的信息），那么就不需要串行。

多线程的 JOB 不支持分布式的文件（Distributed files），因为这些文件不能保障线程的安全。

ODP 的共享：

支持多线程的的 JOB 允许共享打开文件，但是并不总是共享 ODP（Open data path）

如果文件定义了 SHARE(*YES)和 OPNSCOPE(*ACTGRPDFN)，那么一个线程所 create 的子线程，如果是运行同一个活动作业组中，就可以共享 ODP；

如果文件定义了 SHARE(*YES)和 OPNSCOPE(*JOB)，那么由同一个线程所 create 的子线程都就可以共享 ODP；

(那么如果我们通常定义的 SHARE 都是 NO，所以实际上同一进程下的各个线程默认就不是共享 ODP 了？)

OVRDBF:

只有初始线程能使用 OVRDBF， 辅线程使用 OVRDBF 会报错。

只有 JOB 级的，和活动作业组级的 OVRDBF 能作用于辅线程（JOB level， activation group level）

调用级别的 OVRDBF 对辅线程无效（Call level）

同样，DLTOVR 命令也只能在初始线程中使用。

回收资源：

回收资源的命令，对于多线程来说都是不安全的，因为系统不会跟踪(track)资源，所以无法识别资源是否仍在被线程使用。

所以 RCLRSRC、RCLACTGRP 不能在辅线程中调用，但可以在初始线程中调用。如果在辅线程中调用，系统将会报错。

2 线程的基本管理操作

2.1 线程的属性

可以在启动一个线程时设置线程的属性，或在线程运行的时候更改这些属性。常见的线程属性：

优先级

系统分配的运行时间

堆栈空间

影响到线程可以调用的函数数量

名字

我们可以根据线程的名字，来 DEBUG 或是 TRACK 这个运行中的线程

线程组

我们可以通过线程组，来管理同一时间运行的多个线程

Detach state

这个状态标识了当线程结束时，我们如何回收，或保留这个线程使用过的资源

任务计划

线程在系统或在应用中是如何被安排、计划的。

继承

判断线程的属性是否继承

更改线程的属性，可以使用系统 API 函数，如 pthread_attr_setdetachstate()函数就可以更改 detach state 这个属性，详见 Pthread_attr_setdetachstate()

更改线程属性后，再启动的线程就将具备更改后的属性，见下例：

```
#define _MULTI_THREADED
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

#define checkResults(string, val) {                    \
 if (val) {                                                \
    printf("Failed with %d at %s", val, string); \
    exit(1);                                                \
 }                                                            \
}

void *theThread(void *parm)
{
    printf("Entered the thread\n");
    return NULL;
}
```

```
int main(int argc, char **argv)
{
   pthread_attr_t              attr;
   pthread_t                   thread;
   int                         rc=0;

   printf("Enter Testcase - %s\n", argv[0]);

   printf("Create a default thread attributes object\n");
   rc = pthread_attr_init(&attr);
   checkResults("pthread_attr_init()\n", rc);

   printf("Set the detach state thread attribute\n");
   rc = pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);
   checkResults("pthread_attr_setdetachstate()\n", rc);

   printf("Create a thread using the new attributes\n");
   rc = pthread_create(&thread, &attr, theThread, NULL);
   checkResults("pthread_create()\n", rc);

   printf("Destroy thread attributes object\n");
   rc = pthread_attr_destroy(&attr);
   checkResults("pthread_attr_destroy()\n", rc);

   printf("Join now fails because the detach state attribute was changed\n");
   rc = pthread_join(thread, NULL);
   if (rc==0) {
       printf("Unexpected results from pthread_join()\n");
       exit(1);
   }
   sleep(2);

   printf("Main completed\n");
   return 0;
}
```

2.2 启动线程

当我们的应用程序创建了一个线程的时候，系统将会对线程的属性、控制结构、和运行时间等内容进行初始化，以保证线程安全的运行。

当然，启动一个线程的时候，我们也需要在应用程序中对该线程可能使用到的的数据和输入输出参数进行初始化。

启动一个线程后，系统将会为这个线程分配一个唯一的线程标识号。线程标识号是一个整型变量，我们可以通过这个线程标识号，来对该线程进行 DEBUG，TRACE，或其它类型

的管理操作。但是不能通过线程标识号直接操作或控制这个线程。

大部分线程的 API 函数都会返回线程描述符，我们可以通过返回的线程描述符对线程进行直接操作，也可能通过一些同步机制等待线程结束处理。

下面的例子中，主程序启动了一个线程，并向这个线程传递了两个参数，一个是整型变量，一个是 124 位长的字符型变量。参数做为一个全局变量来定义。

启动的线程不仅打印了主程序传递过来的参数，而且还使用了 pthread_getthreadid_np() 函数取出自身的线程标识号，注意该函数的输出是一个 pthread_id_np_t 类型的结构（其实该结构里面也就是 hi,lo 两个整型变量）

这里主要用到的，就是 pthread_create()这个函数，函数说明详见 pthread_create()

```
#define _MULTI_THREADED
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#define checkResults(string, val) {                    \
 if (val) {                                                \
    printf("Failed with %d at %s", val, string); \
    exit(1);                                                \
 }                                                            \
}
typedef struct {
    int          threadParm1;
    char         threadParm2[124];
} threadParm_t;

void *theThread(void *parm)
{
    pthread_id_np_t          tid;
    threadParm_t *p = (threadParm_t *)parm;
    tid = pthread_getthreadid_np();
    printf("Thread ID %.8x, Parameters: %d is the answer to \"%s\"\n",
            tid.intId.lo, p->threadParm1, p->threadParm2);
    return NULL;
}

int main(int argc, char **argv)
{
   pthread_t                  thread;
   int                         rc=0;
   threadParm_t              *threadParm;

   printf("Enter Testcase - %s\n", argv[0]);
```

```
    threadParm = (threadParm_t *)malloc(sizeof(threadParm));
    threadParm->threadParm1 = 42;
    strcpy(threadParm->threadParm2, "Life, the Universe and Everything");

    printf("Create/start a thread with parameters\n");
    rc = pthread_create(&thread, NULL, theThread, threadParm);
    checkResults("pthread_create()\n", rc);

    printf("Wait for the thread to complete\n");
    rc = pthread_join(thread, NULL);
    checkResults("pthread_join()\n", rc);

    printf("Main completed\n");
    return 0;
}
```

2.3 结束线程

结束线程通常是由该线程自身发起的。

当一个线程完成了所有的处理之后，它将会有一个关闭自身的动作，释放系统资源以便之后其它的线程使用这些资源。

有些 API 函数要求应用程序在程序结束时，明确地给出释放资源的语句。也有些线程的处理机制没有这样要求（如 JAVA）。

可以有多种方法去结束一个线程。最好的方法就是 return 到创建这个线程的程序中。因为有关线程的 API 函数。

有些 API 函数也支持 exception 机制。这里所说的 Exception 机制，是指当发生一个一个 exception 而且没有去处理它的时候，线程将会结束。

下面的例子中，主要是在线程调用的函数中使用的 pthread_exit()函数来结束掉辅线程，以及初始线程中使用的 pthread_join()接收辅线程中的返回。

Pthread_exit 函数的说明详见 pthread_exit　结束线程

Pthread_join 函数的说明详见 pthread_join

```
#define _MULTI_THREADED
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#define checkResults(string, val) {                    \
  if (val) {                                           \
    printf("Failed with %d at %s", val, string); \
    exit(1);                                           \
  }                                                      \
```

```
}

const int THREADFAIL = 1;
const int THREADPASS = 0;

void *theThread(void *parm)
{
    printf("Thread: End with success\n");
    pthread_exit(__VOID(THREADPASS));
    printf("Thread: Did not expect to get here!\n");
    return __VOID(THREADFAIL);
}

int main(int argc, char **argv)
{
  pthread_t                thread;
  int                       rc=0;
  void                     *status;

  printf("Enter Testcase - %s\n", argv[0]);

  printf("Create/start a thread\n");
  rc = pthread_create(&thread, NULL, theThread, NULL);
  checkResults("pthread_create()\n", rc);

  printf("Wait for the thread to complete, and release its resources\n");
  rc = pthread_join(thread, &status);
  checkResults("pthread_join()\n", rc);

  printf("Check the thread status\n");
  if (__INT(status) != THREADPASS) {
      printf("The thread failed\n");
  }

  printf("Main completed\n");
  return 0;
}
```

2.4 取消线程

取消线程通常不是由该程序自身发起的，而是由其它的线程发起。

取消线程的时候要注意，如果我们应用程序中，对于清除数据与解锁的处理机制不合理时，将可能会破坏数据，或造成应用程序死锁。

在下面这个例子中，子线程所调用的函数每隔一秒钟打印一行，主程序在创建子线程 3

秒钟后，发出取消该子线程的指令。

主要使用的函数为 pthread_cancel()，API 函数说明见 pthread_cancel 取消线程

注意到使用了 pthread_cancel 之后，仍然要使用 pthread_join 函数等待子线程结束；

如果子线程被成功取消，那么 pthread_join 函数取到的状态将为 PTHREAD_CANCELED。

```
#define _MULTI_THREADED
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#define checkResults(string, val) {                    \
 if (val) {                                            \
    printf("Failed with %d at %s", val, string); \
    exit(1);                                           \
 }                                                     \
}

void *theThread(void *parm)
{
    printf("Thread: Entered\n");
    while (1) {
        printf("Thread: Looping or long running request\n");
        pthread_testcancel();
        sleep(1);
    }
    return NULL;
}

int main(int argc, char **argv)
{
   pthread_t                thread;
   int                       rc=0;
   void                     *status;

   printf("Enter Testcase - %s\n", argv[0]);

   printf("Create/start a thread\n");
   rc = pthread_create(&thread, NULL, theThread, NULL);
   checkResults("pthread_create()\n", rc);

   printf("Wait a bit until we 'realize' the thread needs to be canceled\n");
   sleep(3);
   rc = pthread_cancel(thread);
   checkResults("pthread_cancel()\n", rc);
```

```
    printf("Wait for the thread to complete, and release its resources\n");
    rc = pthread_join(thread, &status);
    checkResults("pthread_join()\n", rc);

    printf("Thread status indicates it was canceled\n");
    if (status != PTHREAD_CANCELED) {
        printf("Unexpected thread status\n");
    }

    printf("Main completed\n");
    return 0;
}
```

2.5 挂起和重新运行线程

有时我们需要暂时停止线程的运行。当我们挂起一个线程时，这个线程的状态，以及线程属性、锁住的记录， 都将维持现状，直至线程重新开始运行（resume）。

挂起线程时要小心，因为这可能会导致应用程序死锁，或超时。我们可以使用其它更安全的方式来解决大部分问题，包括挂起线程。（比如说同步机制）

挂起线程后，我们需要在应用程序中重新启用这个线程，重新启用后，线程将从挂起点继续开始运行。

2.6 等待线程结束

当我们使用线程的时候，知道线程何时结束是很重要的。等待线程执行一个操作，或等待线程发生一个事件，称之为同步机制。

常见的等待，就是等待至线程结束。当线程结束的时候，应用程序将会被提示线程分配的工作已完成，或线程运行失败。我们可以通过 API 函数中设置的参数来确认线程运行的成功与否。

在大型的应用程序中，等待一组线程结束可能是一种比较好的方式。比如说通过调用程序

在下面这个例子中，主程序启动了多个子线程，并等待这些子线程结束，然后检查子线程的结束状态。

在这个例子中，等待一个线程结束使用到的函数仍然是 pthread_join.

```
#define _MULTI_THREADED
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#define THREADGROUPSIZE     5
```

```
#define checkResults(string, val) {                    \
  if (val) {                                           \
     printf("Failed with %d at %s", val, string); \
     exit(1);                                          \
  }                                                    \
}

void *theThread(void *parm)
{
     printf("Thread %.8x %.8x: Entered\n", pthread_getthreadid_np());
     printf("Thread %.8x %.8x: Working\n", pthread_getthreadid_np());
     sleep(15);
     printf("Thread %.8x %.8x: Done with work\n", pthread_getthreadid_np());
     return NULL;
}

int main(int argc, char **argv)
{
   pthread_t                thread[THREADGROUPSIZE];
   void                     *status[THREADGROUPSIZE];
   int                      i;
   int                      rc=0;

   printf("Enter Testcase - %s\n", argv[0]);

   printf("Create/start some worker threads\n");
   for (i=0; i <THREADGROUPSIZE; ++i)
  {
      rc = pthread_create(&thread[i], NULL, theThread, NULL);
      checkResults("pthread_create()\n", rc);
   }

   printf("Wait for worker threads to complete, release their resources\n");
   for (i=0; i <THREADGROUPSIZE; ++i) {
      rc = pthread_join(thread[i], &status[i]);
      checkResults("pthread_join()\n", rc);
   }

   printf("Check all thread's results\n");
   for (i=0; i <THREADGROUPSIZE; ++i) {
      if (status[i] != NULL) {
         printf("Unexpected thread status\n");
      }
   }
```

```
    printf("Main completed\n");
    return 0;
}
```

2.7 让进程先处理另一个线程

有时，我们的应用程序需要使当前的线程让步于进程中其它线程。

当线程发出一个让步指令 yielding 时，系统将会立刻运行另一个相等或更高优先级的，活动中的线程。如果当时准备运行的线程中，没有优先极相等可更高的，那么让步指令将不会产生效果。让步机制是实时处理的，不会对之后的任务产生附加的影响。相关的 API 函数为 sched_yield.

目前 400 的 i 平台上，提供了全面的多任务计划的算法。在一个写法良好的应用程序中，线程很少需要采用让步，因为系统提供了很多用来同步线程的 API。

3 线程的安全性（Thread safety）

一个函数可以被同一进程下的所有线程同时调用，并且该函数内调用的函数也可以被所有线程同时调用时，那么这个函数就具备线程安全性（threadsafe）。

根据测试，绝大部分自己写的程序，无论是 PGM，还是 MODULE，都不具备线程的安全性，也就是不能被同一进程下的多个线程同时调用。

3.1 存储用法和线程应用

当我们定义了变量之后，多个线程就可能会访问或使用这些变量。应用程序中常用到的这些变量（的存储空间），以及其作用范围如下：

Global storage 全局变量（的存储空间）：

在一个源文件或 Module 中定义的全局变量，对该应用程序中其它的源文件、Module 都是可见的。共享范围为整个应用程序。这种共享是一个很常见的线程安全问题。（也就是在多线程并发时，要特别注意这类变量的赋值）

Static storage 静态变量（的存储空间）

静态变量也是全局变量的一种，只不过它的作用范围被限制在声明该变量的源文件、Module、或函数中。共享范围为声明静态变量的可执行 Module 中。

Heap storage 动态分配的存储空间

这类存储由我们的应用程序动态进行控制，分配、回收存储空间，比如说 C 语言里的 malloc()， free()函数。共享范围为整个应用程序。

如果在动态分配了存储空间之后，我们向另一个线程传递了该存储空间的指针（比如通过全局变量，或静态变量来传递，或者是其它的方法传递）， 那么另一个线程就可以通过这个指针，来使用或回收这个存储空间。这种共享处理也是一个很常见的线程安全问题。

Automatic storage 自动分配的存储空间

函数内部私有变量，由系统自动分配存储空间。自动分配的存储空间对该进程下的其它线程是不可见的。每次线程调用函数的时候，都会重新自动分配存储空间。每个线程都拥有它们自己的自动分配的存储空间。基于复杂的串行、同步机制，一个线程不能访问另一个线程的自动分配的存储空间

操作系统在活动作业组中，进一步限制了全局变量、静态变量、Heap 存储的作业范围。这意味着在不同的活动作业组中，使用了相同的全局变量、静态变量的程序，实际上访问的是这些变量和存储空间的不同版本？（This means that application code or threads that are running in different activation groups but are using the same global or static variables access different versions of those variables and their storage）

与此相似，尽管一个活动作业组可以使用另一个活动作业组中分配的 Heap 存储空间，但是不能回收 Heap 存储空间。（也就是使用的其实是不同的版本？）

3.2 JOB 级的资源

在编写多进程程序的时候，我们需要考虑 JOB 级的资源，当线程使用到这些资源的时候，必须不能与该进程下其它线程冲突。

有些资源的作业范围是活动作业组的级别，比如说打开数据库文件。我们编写多进程程序时，需要把这种活动作业组级别的资源也视为 JOB 级的资源来使用。

如果一个线程在使用这些资源，而另一个线程需要修改这些资源时，我们的应用程序就需要考虑使用合适的同步机制来处理这类问题。

常见的 JOB 级，活动作业组级的资源有以下几种，在使用时要注意线程间的冲突：

动态分配的存储空间、静态变量、全局变量

这是最常见的共享资源，见上一节。

打开的文件(Open Files)

当我们打开了一个文件之后，同一进程下的所有线程就可以共享文件系统的文件，以及数据库文件，这种共享可以通过线程之间传递指针或文件描述符来实现。

工作目录总是进程级的共享。(Scope of process)

Locales 地址？

The locale of an application is an activation group resource. All threads share the locale. Changing the locale affects other threads with regard to collating sequences or other locale information.

CCSID、环境变量

CCSID 与环境变量都是 JOB 级的资源。更改这两类资源，将会影响到该 JOB 下的所有线程。

3.3 API 的线程安全级别

每一个 API 函数都有一个线程安全级别。在使用这些 API 之前，我们需要确认在多线程中调用的 API 是否足够安全。

API 的线程安全级别分为以下几类：

安全（yes）

这类 API 函数可以在并发的多个线程中安全地同时调用，而不需要进行任何限制。这种类型的 API 内部所调用的函数，也具体线程安全性。

一定条件下安全（Conditional）

这个标志说明这类 API 函数所提供的功能，有一些是不具体线程安全性的。在 API 函数的说明中，会指出线程安全的限制条件。（即哪些条件下调用 API 函数，线程是安全的）

这类 API 函数的产生有可能因为系统底层支持不具备线程安合性，也可能是因为 API 函数会调用一个退出指针？(API can call a exit point)

举例而言，许多文件系统的 API 函数在一个具体线程安全性的文件系统中使用文件，是完全安全的。但是一些在一定条件下安全的 API 函数，在相同的环境中，就有可能拒绝访问。在 API 函数说明中，将会说明在哪些条件下，函数会拒绝访问。

不安全（No）

这类 API 函数不具备线程安全性，本来不应该在多线程程序中使用。有时，这类 API 函数可能会拒绝访问，有时也不会（大部分函数都不会拒绝访问）。

与 CL 命令不同，当调用不安全的 API 函数时，系统不会在 JOB LOG 中产生调试信息。（也就是说 CL 命令调用这些不安全的 API 函数时，会 JOBLOG 中生成调试信息）

在多线程中使用不具备线程安全性的 API 函数需要一定的技巧。

3.4 CL 命令和线程安全

ILE 环境下的 CL 命令，或是编译后的 CL 程序，是具备线程安全性的；

原始程序模型(OPM)下的 CL 程序不具备线程安全性。

OPM 环境下的 CL 代码，或者是 4.3 以前的版本中 ILE 环境下的 CL 代码，在 CL 命令执行时会发送一个 CPD000B 的调试信息，接下来继续执行命令（该执行的结果未知）。这可能会导致线程的不安全，也可能不会，取决于底层代码的支持。

对于一个命令来说，与线程安全相关的有两个参数：

线程安全属性(threadsafe attribute – THDSAFE)

多线程作业运行属性（multithreaded job action attribute – MLTTHDACN）

多线程作业运行属性仅针对不具备线程安全性的命令（即线程安全属性为 NO 时才有效），系统对多线程作业运行属性设置不同参数时的处理如下：

***NORUN**

系统先发一个 CPD000D 的调试信息，然后不执行这个命令。在发送 CPD000D 之后，将会再发送一个 CPF0001 的退出信息。

***MSG**

系统同样先发送一个 CPD000D 的调试信息，然后开始执行这个命令

***RUN**

系统不发送调试信息，直接开始运行。

如果一个 JOB 支持多线程，但并没有使用多线程时，系统也允许不具备线程安全性的程序直接运行。

当我们使用 DSPCMD 命令查看该命令的多线程作业属性时，有时系统显示的值为 *SYSVAL，也就是系统默认值。这时可以使用 DSPSYSVAL 来做进一步查看：

DSPSYSVAL SYSVAL(QMLTTHDACN)

也可以用 CHGSYSVAL 来修改这个系统值（不过一般开发人员好象没有权限更改系统值）

3.5 拒绝访问的函数和线程安全

基于系统完整性的考虑，以及为防止数据的毁坏，有些 API 函数以及 CMD 命令在一定条件下具备线程安全性，而在某些条件下则不具备线程安全性。这些 API 以及命令将有可能会拒绝部分或所有的访问。

拒绝访问的分类条件如下：

多线程能力

在这种情况下，函数是否拒绝访问，取决于当前 JOB 对多线程的支持能力。如果当前 JOB 支持多线程，但并不关注当前 JOB 中线程的数量时，那我们就不能调用这类函数。此时，函数会返回一个 CPF1892 的退出信息给调用者。

初始线程

有些函数只能在初始线程中被调用。如果我们在辅线程中调用这类函数，函数会返回一个 CPF180C 的的退出信息给调用者。如果要在辅线程中使用的话，可以通过向初始线程发出一个请求，然后初始线程调用的方式来实现这个功能。OVRDBF 就是一个常见的只能在初始线程中调用的例子。

多个线程（More than on thread）

在这种情况下，JOB 中的线程数量将会导致函数拒绝访问。如果 JOB 中超过了一个线程，那么函数将会返一个 CPF180B 的退出信息给调用者。其它函数将会在返回错误信息时将其赋值为 ENOSAFE（3524）.

这些在多线程下可能会拒绝访问的函数，在整个 JOB 中只有一个线程运行时，是不会拒绝访问的，可以随时调用。

所有访问文件系统的 API 函数，都不具备线程安全性。

3.6 退出点（exit point）

不太理解这个 exit program 和 exit point 是什么概念，没有用过。只知道在源代码处用

F13，有个 user exit program 的参数，可以写 RPG 程序，通过三个指针变量取到当前编辑代码的信息，不知道和这个有没有什么关系。但是这个是在交互式作业中使用的，和多线程的联系似乎不大。

With the i5/OS® registration facility, you can define exit points for functions in an application and register programs that run at those exit points.

Some i5/OS services also support the registration facility for registering exit programs. They have predefined exit points that are registered when those services are installed. The registration facility itself is threadsafe. You can use it to specify attributes of thread safety and multithreaded job actions for exit program entries.

Without careful evaluation, however, you should not consider existing exit programs to be threadsafe, even though you can call exit programs in a multithreaded job. The same restrictions apply to exit programs as to any other code that runs in a multithreaded job. For example, only exit programs written using a threadsafe Integrated Language Environment® (ILE) language that can be made threadsafe

4 多线程的程序技巧

在编写多进程的时候，我们需要对当前的应用系统进行评估，主要需要考虑线程安全性。在不知道调用的系统服务，API 是否具备线程安全性时，必须假设它是不安全的。

4.1 多线程的同步

当线程已具备安全性时，多个线程之间的同步就变为最重要的部分。同步，是指多个线程之间互相配合，以一定的关联方式使得操作得以连续不断地进行（Synchronization is the cooperative act of two or more threads that ensures that each thread reaches a known point of operation in relationship to other threads before continuing）。没有使用同步机制去处理共享资源，是导致应用数据被破坏的最常见的原因。

下面列出了常见的同步方式，使用的系统资源由小到大排列：

1. CMPSWP 指令 Compare and swap
2. 互斥体 Mutual exclusion(mutexes) and threads
3. 信号量 Semaphores and threads
4. 条件变量与线程 Condition variable and threads
5. 线程的同步单元 Threads as synchronization primitives
6. 空间锁 Space location lock
7. 目标锁 Object lock

4.1.1 CMPSWP

我们可以通过系统的 CMPSWP（compare and swap）指令，在多线程程序中访问数据。CMPSWP 指令的语法格式是 CMSWP（&operand1， &operand2， swap operand），也就是前两位是变量地址，第三位变量数值。（后面还有扩展参数，先不做深究）

系统根据地址，比较 operand1 与 operand2 的值。

如果这两个值相等，那么将 swap operand 的值赋到 operand2 中，返回 1

如果这两个值不等，那么将 operand2 的值将赋到 operand1 中，返回 0

如果返回 1 时（相等），那么系统将会确保在取出 operand2 的值来比较和将 swap operand 的值赋到 operand2 这段时间之内，没有别的 CMPSWP 指令访问 operand2。这也就保障了数据的安全性。

如果返回不等时，并不能保证 operand1 不被别的 CMPSWP 指令访问。因此只允许 operand2 做为并发控制中的共享变量。

Operand1, operand2, swap operand 的长度必须相等，允许长度范围在 1、2、4、8 位字节（byte）。

在下面这个例子中，注意对 CMPSWP 命令的使用。

该程序宏定义了一个 ATOMICADD 的函数原型，该函数用来将共享变量 var 加上 val。

该函数原型中，while 循环的条件，对 CMPSWP 命令的返回取反判断，也就是说：

aatemp1 与 var 的值相等时，将 aatemp2 的值赋值到 var 中（这一步是由 CMPSWP 命令自动完成），退出循环；

aatemp1 与 var 的值不等时，每次都对 aatemp2 重新赋值，使其等于当前的 aatemp1+val（因为进行完 CMPSWP 命令后，当其时的 var 的值已赋到 aatemp1 中；而且不能将 aatemp2 赋值为 var+val，因为此时的 var 的值可能又被其它并发的线程所更改）

程序执行结束后，结果应该为 1000000.

如果在线程函数 theThread 中，没有使用 ATOMICADD 这个函数，而是直接使用

```
shareData++;
```

这样的语句，那么执行出来的结果将不会是 1000000，因为程序中没有对 shareData 变量进行保护，当多个线程同时去更改该变量时，实际上只有最后一个更改的生效。

CMPSWP 命令是占用系统资源最小的一种同步机制。

例：

```
#include <mih/cmpswp.h>
#define _MULTI_THREADED
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#define ATOMICADD(var, val) {                    \
    int aatemp1 = (var);                         \
    int aatemp2 = aatemp1 + val;                 \
    while( ! _CMPSWP( &aatemp1, &var, aatemp2 ) ) \
      aatemp2 = aatemp2 + val;                   \
}
```

```
#define NUMTHREADS      10
#define LOOPCONSTANT 100000

int      shareData=0;

void *theThread(void *parm)
{
     int loop;
     printf("Thread %.8x, %.8x is Entered\n", pthread_getthreadid_np());
     for(loop=0;loop<LOOPCONSTANT;loop++)
        ATOMICADD(shareData, 1);
     return NULL;
}

int main(int argc, char **argv)
{
   pthread_t                  thread[NUMTHREADS];
   int                         rc=0;
   int                         i;

   for (i=0; i<NUMTHREADS; i++)
        rc = pthread_create(&thread[i], NULL, theThread, NULL);

   for (i=0; i <NUMTHREADS;i++)
        rc = pthread_join(thread[i], NULL);
   printf("Data = %d\n", shareData);
   return 0;
}
```

4.1.2 互斥体

互斥体 mutex 是 Mutual exclusion 的简称。我们可以通过对互斥体的处理，来实现多线程并发时，一个时点上只允许一个线程处理数据的目的。

互斥体的常见操作是 create, lock , unlock ,destory。一个线程成功的对互斥体执行了 lock 操作后，这个线程就成为了该互斥体的拥有者，直到该线程对互斥体执行 unlock 操作。当执行了 unlock 操作后，系统就会将互斥体交由另一个排队等待 lock 该互斥体的线程中。一个互斥体只能有一个拥有者。

互斥体操作可以递归。递归的互斥体允许拥有者重复地 lock 互斥体。互斥体的拥者将会保持当前状态直到 unlock 的请求次数与 lock 的请求次数相同。

互斥体可以设置超时等待，也可以设置为立刻返回。

互斥体是我们在进行多线程编程是常用的一种方式。

更具体信息以及使用方法，可查看操作互斥体的 API 互斥体操作 API -- Mutex Operation API.

4.1.3 信号量

信号量（有时可以认为是信号量计数器）可以用来控制对共享资源的访问。一个信号量，实际上也是一个整型的计数器。每个信号量当前都有一个数值，这个数值大于或等于 0。（有的系统可能允许小于零，用来标识当前阻塞的线程数量，按文中的表述，在 400 上信号量的值不会小于零）

当线程 lock 信号量时，信号量的值就会减 1；如果信号量的值已为 0，那么该线程将会阻塞，直到另一个线程对信号量执行 unlock 操作。

当线程 unlock 信号量时，信号量的值会加 1，然后唤起一个之前被阻塞的线程。

可以认为信号量的初始值标识的其实是针对某类资源，允许并发的线程数量。当初始值为 1 的时候，一个 lock 操作就会将信号量的值减为 0；之后其它的线程再进行 lock 操作时，将会被阻塞。此时的信号量就与互斥体很类似了。不过与互斥体的不同点在于信号量没有所有权的概念，也就是当信号量由 A 线程执行 lock 操作后，可以由 B 线程去执行 unlock 操作；而互斥体必须由当前 lock 的线程来执行 unlock。信号量这个特性可能会导致一些不可预测的结果，需要注意。

JAVA（或者说 400 上的 JAVA 程序）不能使用信号量。

信号量的使用方法，可查看操作信号量的 API 信号量的 API

4.1.4 Condition variable and threads

条件变量的设置（Condition variable，简写作 CV）允许线程等待一定的事件发生或一定条件满足时，才开始继续运行。

线程可以等待指定条件的发生，同时另一些线程会将发生的条件事件广播出去以激活那些等待这些条件事件的线程。可以认为条件变量类似于别的平台上使用 event 去同步线程的机制。

条件变量没有所有者，也没有状态（stateless）。没有状态，也就意味着当一个线程发出一个信号，标识某个事件发生，如果当时没有符合这个事件的线程在等待，那么这个信号将会被抛弃，系统不再进行任何处理，这个信号就丧失了有效性。进一步说，条件变量的有效性是实时的，如果线程 A 挂起，等待某个事件发生而激活，线程 B 则发出一个信号标识这个事件发生用来激活已挂起的线程 A。但如果线程 B 的处理在线程 A 挂起之前，那么线程 A 将会一直挂起，因为这个条件已经发生。

具体信息详见条件变量的 API。

4.1.5 Threads as synchronization primitives

当一个指定线程等待另一个线程结束运行，才开始继续运行时，我们也可以认为这也是一种线程之间的同步。

这种比较原始的同步机制没有所有者的概念，它仅仅只是一个线程等待另一个线程结束而已。

比如说 sched_yield 这个函数就可以使当前线程在另一线程结束之后运行，详见 sched_yield

4.1.6 Space location lock

空间地址锁(space location lock)，是一个存放在单字节空间中的逻辑锁。空间地址锁不会改变应用程序所使用到的存储空间，它是系统自身所使用到的一个信息记录片。

空间地址锁用起来，与互斥体有点类似，它与互斥体有几下几个方面的不同：

1、 我们可以直接使用空间地址锁来操作数据。空间地址锁不需要应用程序去创建和管理额个的目标。（相比之互斥体就需要创建一个互斥体变量，使用完毕之后还需要销毁掉）。互斥体是作用于线程的，而空间地址锁是直接作用于某个变量的。也就是说它只关注某个变量是否被锁，在使用上与互斥体类似，但在概念与互斥体有较大差异。

2、 Space location locks allow an application to coordinate the use of different locking request types. For example, more than one thread can use space location locks to acquire a shared lock on the same data.（不同的线程可以使用空间地址锁获取同一个数据的共享锁？）

3、 Due to the extra lock types that are provided by space location locks, the concept of an owner is slightly different than with mutexes. There can be multiple owners of a shared lock if each owner has successfully acquired the shared lock. For a thread to get an exclusive lock, all of the shared locks must be unlocked.

4、 与互斥体的性能比较。空间地址锁锁住一个共享数据的路径，大概需要 500 个 RISC 指令（recude instrction set computer），而互斥体只需要 50 个；但是空间地址锁不需要创建、销毁等等指令。而互斥体大约需要 1000 个 RISC 指令。

下面的例子，说明了对空间地址锁的一个简单的用法，主要用到了 locksl，unlosksl 这两个函数。注意空间地址锁直接对关键数据的操作（锁、解锁）。

```
#define _MULTI_THREADED
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <mih/milckcom.h>       /* Lock types                 */
#include <mih/locksl.h>         /* LOCKSL instruction     */
#include <mih/unlocksl.h>       /* UNLOCKSL instruction */

#define checkResults(string, val) {                  \
 if (val) {                                                  \
    printf("Failed with %d at %s", val, string); \
    exit(1);                                                  \
 }                                                             \
}

#define                 NUMTHREADS     3
int                     sharedData=0;
int                     sharedData2=0;
```

```
void *theThread(void *parm)
{
    int     rc;
    printf("Thread %.8x %.8x: Entered\n", pthread_getthreadid_np());
    locksl(&sharedData, _LENR_LOCK);      /* Lock Exclusive, No Read */
    /********** Critical Section ********************/
    printf("Thread %.8x %.8x: Start critical section, holding lock\n",
            pthread_getthreadid_np());
    /* Access to shared data goes here */
    ++sharedData; --sharedData2;
    printf("Thread %.8x %.8x: End critical section, release lock\n",
            pthread_getthreadid_np());
    unlocksl(&sharedData, _LENR_LOCK); /* Unlock Exclusive, No Read */
    /********** Critical Section ********************/
    return NULL;
}

int main(int argc, char **argv)
{
   pthread_t                thread[NUMTHREADS];
   int                       rc=0;
   int                       i;

   printf("Enter Testcase - %s\n", argv[0]);

   printf("Hold Lock to prevent access to shared data\n");
   locksl(&sharedData, _LENR_LOCK);      /* Lock Exclusive, No Read */

   printf("Create/start threads\n");
   for (i=0; i <NUMTHREADS; ++i) {
       rc = pthread_create(&thread[i], NULL, theThread, NULL);
       checkResults("pthread_create()\n", rc);
   }

   printf("Wait a bit until we are 'done' with the shared data\n");
   sleep(3);
   printf("Unlock shared data\n");
   unlocksl(&sharedData, _LENR_LOCK); /* Unlock Exclusive, No Read */

   printf("Wait for the threads to complete, and release their resources\n");
   for (i=0; i <NUMTHREADS; ++i) {
 rc = pthread_join(thread[i], NULL);
       checkResults("pthread_join()\n", rc);
   }
```

```
    printf("Main completed\n");
    return 0;
}
```

4.1.7 Object lock

原文这里居然没有举例，反正有了上面那么多同步的方法，这里就略过算了。

4.2 初始化和线程安全

有的时候，我们希望延缓线程的初始化，直到我们需要使用这些的资源的时候才进行初始化。然而对于多个线程而言，需要系统一开始就为它们分配相应的资源，这些资源必须具备线程安全性，而且只能进行一次初始化。

再具体一点，比如说我们希望在子线程中，只对某个全局变量进行一次初始化，但子线程里的程序该如何写？有多种方法来实际这个目的，比如说可以一个布尔型变量，让应用程序对这个变量进行判断来决定是否执行初始化。不过使用 pthread_once 这个函数就会显得比较专业一点。函数的具体使用方法详见 pthread_once 执行一次初始化。

4.3 线程的特有数据(thread specific data)

有时在应用程序中使用的全局变量，我们希望它只作用于当前的线程。即我们希望每个子线程有针对它们自身的私有的全局变量，这个全局变量对别的线程不可见，这时就可以使用线程的特有数据（Thread specific data）。这个全局变量由当前线程分配空间，并加以保存。可以为这个全局变量分配一个销毁函数，当当前线程结束时，系统将会自动运行这个销毁函数，来清空当前线程分配的存储空间。我们可以使用线程的特有数据来代替全局变量，因为一个线程中，所有的函数对这片存储空间的请求，都会得到一个相同的值。而其它线程中的函数在相同的语句中访问的，是调用函数的线程自身的存储空间。（也就是线程与线程之间互不干扰）。

与线程的特有数据相关的函数为：

pthread_key_create()

pthread_setspecific()

pthread_getspecific()

pthread_key_delete()

详见线程特有数据 API Thread specific storage API

在下面这个例子中，定义了一个 pthread_key_t 结构的变量 tlskey，这个变量虽然是全局变量，但在这个程序的使用中，实际上就是一个线程的特有数据，即它的作用空间仅限于当前线程。

首先，需要使用 pthread_key_create 命令来创建生成这个线程特有数据，同时为这个数据指定一个销毁函数 globalDestructor。当子线程结束时，如果 tlskey 自身，以及它所指向的数据都不为空的话，那么系统将会自动运行销毁函数 globalDestructor。同时，

pthread_key_create 函数在销毁函数参数不为空的时候，也就相当于将 tlskey 指定成为的销毁函数的入口参数，所以在销毁函数中，直接对参数进行 free()操作，其实也就是将 tlskey 指向的地址空间释放，最后再把这片地址空间赋值为空。

在主线程中，每次创建子线程之前，都先分配一片存储空间，然后就这片空间的指针传递到子线程中；子线程通过 pthread_setspecific 函数，将线程特有数据 tlskey 指向了这片存储空间。于是，在当前线程中，所有的函数都可以共享这个特有数据 tlskey，然后通过这个 tlskey 再共享它所指向的存储空间。比如说 showdata 函数，就是通过 pthread_getspecific 函数使用了 tlskey，从而访问到了存储空间中的数据。

当每个子线程结束的时候，系统将会自动运行销毁函数，打印出一句话，然后执行 free 操作，最后再使用 pthread_setspecific 函数，将线程特有数据 tlskey 指向的存储空间清空。

```
#define _MULTI_THREADED
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#define checkResults(string, val) {                    \
 if (val) {                                                \
    printf("Failed with %d at %s", val, string); \
    exit(1);                                                \
 }                                                             \
}

#define NUMTHREADS      3
pthread_key_t             tlsKey = 0;
typedef struct {
   int data1;
   int data2;
} mystruct;

void globalDestructor(void *value)
{
   printf("%.8x,%.8x: In the data destructor\n", \
            pthread_getthreadid_np());
   free(value);
   pthread_setspecific(tlsKey, NULL);
}

void showdata()
{
   mystruct *gdata;
   gdata=pthread_getspecific(tlsKey);
   printf("%.8x, %.8x:    get data data1= %d, data2=%d\n", \
            pthread_getthreadid_np(),    gdata->data1, gdata->data2);
}
```

```
void *threadfunc(void *parm)
{
   mystruct *gdata;
   gdata=(mystruct *)parm;
   printf("%.8x, %.8x:    set data data1= %d, data2=%d\n", \
                pthread_getthreadid_np(), gdata->data1, gdata->data2);
   pthread_setspecific(tlsKey, gdata);
   showdata();
   printf("%.8x, %.8x:    Ready exit thread, run function destruct\n",
              pthread_getthreadid_np());
   return NULL;
}

int main(int argc, char **argv)
{
   pthread_t                    thread[NUMTHREADS];
   int                           rc=0;
   int                           i=0;
   mystruct *gData;
   printf("Enter Testcase - %s\n", argv[0]);

   printf("Create a thread local storage key\n");
   rc = pthread_key_create(&tlsKey, globalDestructor);
   checkResults("pthread_key_create()\n", rc);
   /* The key can now be used from all threads */

   for(i=0;i<NUMTHREADS;i++){
      gData=(mystruct *)malloc(sizeof(mystruct));
      gData->data1=i;
      gData->data2=(i+1)*i;
        pthread_create(&thread[i], NULL, threadfunc, gData);
   }

   for(i=0;i<NUMTHREADS;i++)
         pthread_join(thread[i], NULL);

   printf("Delete a thread local storage key\n");
   rc = pthread_key_delete(tlsKey);
   checkResults("pthread_key_delete()\n", rc);
   /* The key and any remaining values are now gone. */
   printf("Main completed\n");
   return 0;
}
```

4.4 调用不具备线程安全性的函数

有些时候，应用程序必须去调用那些不具备线程安全性的函数，也一些变通的方法，可以安全的调用这些函数。

举例来说，假如有个程序调用和 API 函数 foo(), 因为我们已经知道这个函数 foo()是不具备线程安全性的，所以必须通过一种安全的方法来调用它。有两种常见的方法：

通过互斥体的控制，来调用这个函数；

不过这个方法仅仅只限于调用已知源码的函数，因为我们只能通过互斥体来控制同一时点只有一个线程在调用这个不安全的函数，而如果这个函数中又调用了其它不安全的函数时，就必须要对其它不安全的函数进行控制，这一点在不知道源码时，是无法实现的。

也因为这个原因，所以我们不要试图在自己的程序中，用自己的串行逻辑去控制不具备线程安全性的函数使其达到线程安全性。

另起一个 JOB 来完成我们需要的调用

有几种方法，来完成这个目的：

1、 如果应用程序中使用到不具备安全性的 CL 命令，那么可以使用 Qp0zSystem（）这个函数来调用这些命令，这是一个类似于 C 语言里面 system()的函数，系统会启动一个启的进程来完成这个 CL 命令，并在当前线程中等待 CL 命令执行结束并返回（0—成功； 1 – CL 命令不成功； -1--Qp0zSystem()函数执行不成功）。

例：

```
#include <stdio.h>
#include <qp0z1170.h>
int main(int argc, char *argv[])
{
  if (Qp0zSystem("CRTLIB LIB(XYZ)") != 0)
    printf("Error creating library XYZ.\n");
  else
    printf("Library XYZ created.\n");

  return(0);
}
```

2、 如果应用程序中调用了不具备安全性的 API 或程序，可以使用 spawn()函数来启一个 JOB 去运行。Spawn()函数可以继承原线程中的资源，比如 IFS 文件，socket 描述符。Spawn 的例子可见 spawn

3、 如果应用程序频繁的调用了不具备安全性的多个函数，那么可以考虑通过上述方法，启动一个新 JOB，专门去运行这些函数。JOB 之间可以通过消息队列，数据队列进行通讯。

4.5 常见的多线程错误

在多线程编程中，常见的错误有以下几种：

调用不具备线程安全性的函数

这几乎是最常见的错误。在应用程序之中，需要确保它调用的每一个 API 函数都具备线程的安全性。

当前 JOB 不允许创建多线程

要注意 JOBD 中 ALWMLTTHD 的值，为*YES 时才可以。

交互式作业不支持多线程。

如果当前 JOB 对应 JOBD 不支持多线程，那么将无法运行多线程程序

关闭活动作业组

进程下的一个活动作业组，可能对应多个线程，系统无法安全的关闭活动作业组，所以当线程执行了关闭活动作业组的动作时（比如说 C 程序中的 exit(),abort()），系统将会结束掉整个进程。

在前面的活动作业组，以及调用 C 程序的注意事项中，已就这个问题进行了应用层的表述。

混合使用线程 API

IBM 要我们不要把 pthread 的 API 和系统提供的其它线程管理的 API 混用，比如说 JAVA。

事务处理

事务处理是 JOB 级，或活动作业组级的。因为我们无法知道，也无法控制线程运行与活动作业组的对应关系，于是事实上，事务处理就不能针对单个线程了。

如果同时有多个线程在进行数据库操作，那一个一个线程的 commit 操作，可能会导致另一个活动中的线程也执行了 commit 操作。

于是多线程编程，在实际上就不支持事务处理。

5 多线程 JOB 的 DEBUG

居然没有调试成功，失败，略。

6 多线程 JOB 的性能

网上有很多这方面的资料，所以略。

7.2.3 pthread_attr_setdetachstate() 设置线程属性

语法：

```
#include <pthread.h>
int    pthread_attr_setdetachstate(pthread_attr_t    *attr, int detachstate);
```

功能：

设置线程属性中的 detach 状态（detach state），这个状态标识了当一个线程结束时，系统是否会释放线程的资源。这里，“线程结束”这个用语，包括但不仅限于线程的正常退出（即也包括异常中断）。另外，有部分资源（如 automatic storage，具体含义在 JOB 和 JOB 资源中），是当线程结束时总是会被释放的。

Detach 状态的值必须在下面两个中选择其一：

PTHREAD_CREATE_DETACHED　　//就是 0， 表示释放资源？

PTHREAD_CREATE_JOINABLE　　//就是 1， 表示不释放资源？

系统的默认状态值是 PTHREAD_CREATE_JOINABLE

参数：

attr　　(输入参数)

　　标识线程属性结构的地址

detachstate　(输入参数)

　　标识修改 detach state 状态的值，必须为 PTHREAD_CREATE_DETACHED 或 PTHREAD_CREATE_JOINABLE

返回：

0　　表示成功

非 0　表示失败

简单举例：

```
#define _MULTI_THREADED
#include <pthread.h>
#include <stdio.h>
int main(int argc, char **argv)
{
  int                    rc=0;
  int              detachstate;
  pthread_attr_t          pta;

  rc = pthread_attr_init(&pta);
  rc = pthread_attr_getdetachstate(&pta, &detachstate);
  printf("detach state = %d\n", detachstate);

  rc = pthread_attr_setdetachstate(&pta, PTHREAD_CREATE_DETACHED);
  rc = pthread_attr_getdetachstate(&pta, &detachstate);
```

```
    printf("detach state = %d\n", detachstate);

    return 0;
}
```

这个例子就是先对线程属性进行初始化，然后更改 detach state。

为了标识出更改前后的变化，还使用到了 pthread_attr_getdetachstate 这个函数取出当前 detach state，以示区分

7.2.4 pthread_attr_setinheritsched

7.2.5 pthread_attr_setschedparam

7.3 取线程内容 API -- Get Thread Content API

7.3.1 pthread_getconcurrency 取线程的并发等级

7.3.2 pthread_getpthreadoption_np

7.3.3 pthread_getschedparam

7.3.4 pthread_getthreadid_np 取当前线程的唯一标识号

7.3.5 pthread_getunique_np 取指定线程的标识号

7.3.6 pthread_self 取当前运行线程的线程描述符

7.4 设置线程内容 API -- Set Thread Content API

7.4.1 pthread_setconcurrency 设置进程并发等级

7.4.2 pthread_setpthreadoption_np

7.4.3 pthread_setschedparam

7.5 检查线程 API -- Check Thread API

7.5.1 pthread_equal

7.5.2 pthread_is_initialthread_np 检查当前线程是否为初始线程

7.5.3 pthread_is_multithreaded_np 检查当前进程是否拥有超过一个线程

7.6 线程管理 API

7.6.1 pthread_clear_exit_np 清除线程的 EXIT 状态

7.6.2 pthread_delay_np 线程 DELAY

7.6.3 pthread_detach

7.6.4 pthread_once 执行一次初始化

语法：

```
#include <pthread.h>
 int pthread_once(pthread_once_t *once_control,
                  void (*init_routine)(void));
```

功能：

这个函数针对指定的变量 once_control，只执行一次初始化。当多个线程先后执行了同样的 pthread_once 语句时，初始化函数 init_routine 只会执行一次。

初始化函数 init_routine 必须进行如下定义：

```
void initRoutine(void);
```

参数：

once_control

输入参数，分配给初始化事件的控制变量。如果有不同初始化事件（即需要调用不同的初始化函数），那么需要定义不同的控制变量。该变量是一个 pthread_once_t 类型的结构体

init_route

输入参数，初始化函数的指针，这个函数没有入口参数，也没有返回.

返回：

0 – 成功； 非 0 -- 失败

例子：

下面这个例子，就充分体现了 pthread_once 的用法。

初始线程创建了三个子线程，这三个子线程都使用了 pthread_once 语句，但 pthread_once 语句中使用的初始化函数 initRoutine 就只运行了一次（即只会打印一次“In the initRouinte”），而且 number 的值也只为 1。

```
#define _MULTI_THREADED
#include <pthread.h>
#include <stdio.h>
#define checkResults(string, val) {                    \
 if (val) {                                              \
    printf("Failed with %d at %s", val, string); \
    exit(1);                                             \
 }                                                       \
}

#define             NUMTHREADS      3
int                 number          = 0;
int                 okStatus        = 777;
pthread_once_t    onceControl = PTHREAD_ONCE_INIT;

void initRoutine(void)
{
   printf("In the initRoutine\n");
   number++;
}

void *threadfunc(void *parm)
{
   printf("Inside secondary thread\n");
   pthread_once(&onceControl, initRoutine);
   return __VOID(okStatus);
}

int main(int argc, char **argv)
{
   pthread_t                    thread[NUMTHREADS];
```

```
    int                     rc=0;
    int                     i=NUMTHREADS;
    void                    *status;

    printf("Enter Testcase - %s\n", argv[0]);

    for (i=0; i < NUMTHREADS; ++i) {
        printf("Create thread %d\n",
                i);
        rc = pthread_create(&thread[i], NULL, threadfunc, NULL);
        checkResults("pthread_create()\n", rc);
    }


    for (i=0; i < NUMTHREADS; ++i) {
        printf("Wait for thread %d\n", i);
        rc = pthread_join(thread[i], &status);
        checkResults("pthread_join()\n", rc);
        if (__INT(status) != okStatus) {
            printf("Secondary thread failed\n");
            exit(1);
        }
    }

    if (number != 1) {
        printf("An incorrect number of 1 one-time init routine was called!\n");
        exit(1);
    }
    printf("One-time init routine called exactly once\n");
    printf("Main completed\n");
    return 0;
}
```

7.6.5 pthread_trace_init_np

7.6.6 PTHREAD_TRACE_NP

7.6.7 sched_yield

语法：

```
#include <sched.h>
int sched_yield(void);
```

功能：

这个函数可以使用另一个级别等于或高于当前线程的线程先运行。

如果没有符合条件的线程，那么这个函数将会立刻返回然后继续执行当前线程的程序。

参数：

无

返回：

0 – 成功； 非 0 – 失败

例子：

下面这个例子中，只是使用了 sched_yield 这个函数，其实就实际效果上，并未体现出其真正的意义，主要旨在体会用法。

```
#define _MULTI_THREADED
#include <pthread.h>
#include <stdio.h>
#include <errno.h>
#define checkResults(string, val) {                    \
 if (val) {                                                \
    printf("Failed with %d at %s", val, string); \
    exit(1);                                                \
 }                                                            \
}

#define                 LOOPCONSTANT        1000
#define                 THREADS             3

pthread_mutex_t       mutex = PTHREAD_MUTEX_INITIALIZER;
int                     i,j,k,l;

void *threadfunc(void *parm)
{
   int     loop = 0;
   int     localProcessingCompleted = 0;
   int     numberOfLocalProcessingBursts = 0;
   int     processingCompletedThisBurst = 0;
   int     rc;

   printf("Entered secondary thread\n");
   for (loop=0; loop<LOOPCONSTANT; ++loop) {
      rc = pthread_mutex_lock(&mutex);
      checkResults("pthread_mutex_lock()\n", rc);
      /* Perform some not so important processing */
      i++, j++, k++, l++;

      rc = pthread_mutex_unlock(&mutex);
      checkResults("pthread_mutex_unlock()\n", rc);
```

```
        /* This work is not too important. Also, we just released a lock
           and would like to ensure that other threads get a chance in
           a more co-operative manner. This is an admittedly contrived
           example with no real purpose for doing the sched_yield().
           */
        sched_yield();
    }
    printf("Finished secondary thread\n");
    return NULL;
}

int main(int argc, char **argv)
{
    pthread_t                   threadid[THREADS];
    int                          rc=0;
    int                          loop=0;

    printf("Enter Testcase - %s\n", argv[0]);

    rc = pthread_mutex_lock(&mutex);
    checkResults("pthread_mutex_lock()\n", rc);

    printf("Creating %d threads\n", THREADS);
    for (loop=0; loop<THREADS; ++loop) {
        rc = pthread_create(&threadid[loop], NULL, threadfunc, NULL);
        checkResults("pthread_create()\n", rc);
    }

    sleep(1);
    rc = pthread_mutex_unlock(&mutex);
    checkResults("pthread_mutex_unlock()\n", rc);

    printf("Wait for results\n");
    for (loop=0; loop<THREADS; ++loop) {
        rc = pthread_join(threadid[loop], NULL);
        checkResults("pthread_join()\n", rc);
    }

    pthread_mutex_destroy(&mutex);

    printf("Main completed\n");
    return 0;
}
```

7.7 线程操作 API -- Operation Thread API

7.7.1 pthread_create() 创建线程

语法：

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

功能：

按指定的属性创建一个线程。（即设置线程属性之类的 API 函数应在创建线程之前就调用），同时会在线程中运行指定的线程函数，主程序与线程函数之间可以通过一个指针来传递参数。当 pthread_create()成功结束时，返回的线程描述符将会保存下来，用来指向该线程（以便程序的后续处理，例如主程序就可以通过这个返回线程描述符对线程进行操作）。

当线程函数正常返回时，系统隐含地调用了函数 pthread_exit()。

创建的线程有可能（但并不一定）在 pthread_create 函数返回前就开始运行了。

如果线程属性的值被更改的话，之前已创建的线程并不受影响。

如果不特别指定线程属性的话将会使用默认的线程属性。

也就是说，假如我们在程序段中已进行如下定义：

```
pthread_t t;
void *foo(void *);
pthread_attr_t attr;
pthread_attr_init(&pta);
```

如果中间不做其它的处理，那么下面这两句话是等价的：

```
pthread_create(&t, NULL, foo, NULL);
pthread_create(&t, &attr, foo, NULL);
```

新创建的线程中，cancellation state 的值是 PTHREAD_CANCEL_ENABLE。

Cancellation type 的值是 PTHREAD_CANCEL_DEFERRED

初始线程是特殊的，任何初始线程的结束（如使用 pthread_exit()，或其它结束初始线程的操作），都会导致整个进程的结束。

也就是说，如果初始线程启动了子线程，如果不做任何其它操作就结束初始线程的话，那么所有的子线程都会立刻结束。

系统并没有限制一个进程中可以启动的线程最大数。在实际使用中，线程数量的限制决定于 JOB 中可用的存储空间。

在创建线程之后，最后总是使用 pthread_join()或 pthread_detach()函数，使用这两个函数将可以使得线程结束时，资源被回收。

参数：

thread （输出参数）

创建的线程的描述符，可以通过该描述符来操作这个建立的线程

attr （输入参数）

表明创建线程的属性，如果使用 NULL，则表示使用默认的线程属性。

Start_routine 输入参数

创建的线程中调用的函数

arg 输入参数

主线程与创建的子线程之间传递参数的地址

返回：

0 表示成功

非 0 表示失败

例子：

在下面这个例子中，主程序创建了两个线程，这两个线程均调用函数 threadfunc()；

第一次创建的方式，是使用 NULL 方式指定使用默认的线程属性；

第二次创建，是通过一个 pthread_attr_t 结构的变量 pta，来指定使用线程属性。因为主程序中进行了线程属性初始化之后，没有再更改线程属性，所以这两种创建方式实质上都是使用了默认的线程属性。

创建了两个线程之后，再将线程属性的目标 destory。根据上文所说，主程序中线程属性的变更不会影响到已创建的线程。所以这里的 destory 对刚才已创建的线程没有影响。

主程序与线程之间可以通过一个指针来传递参数，本程序中指针对应的参数设为一个结构体变量，结构体中含有一个整型变量，和一个 128 位长的字符变量。我们自己写的程序可以参照这种方式来传递多个变量。

程序打印出来的结果，有可能是：

Create a thread attributes object
Create thread using the NULL attributes
Create thread using the default attributes
Destroy thread attributes object
Inside secondary thread, parm = 5
Inside secondary thread, parm = 77
Main completed

或

Create a thread attributes object
Create thread using the NULL attributes
Create thread using the default attributes
Inside secondary thread, parm = 5
Destroy thread attributes object
Inside secondary thread, parm = 77
Main completed

注意到“Destroy thread attributes object”这句话有可能在第二个线程运行之前开始执行，

也有可能在第一个线程运行之前就开始执行，这是由系统去分配资源执行的，我们无法控制。如果把 sleep(5)这句话去掉的话，那么线程执行函数 threadfunc()中的打印语句，本来应该打印两句，就有可能一句都没有打印或是只打印一句出来。因为主程序（初始线程）的结束，会导致整个进程的结束。当然，使用 sleep 这种等待方式只是用于测试，既不安全，又没有效率。在实际应用中，我们会使用其它更有效率，更安全的语句来等待辅线程的结束，比如 pthread_join().

```
#define _MULTI_THREADED
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#define checkResults(string, val) {                    \
   if (val) {                                          \
      printf("Failed with %d at %s", val, string); \
      exit(1);                                         \
   }                                                   \
 }

typedef struct {
   int      value;
   char    string[128];
} thread_parm_t;

void *threadfunc(void *parm)
{
   thread_parm_t *p = (thread_parm_t *)parm;
   printf("%s, parm = %d\n", p->string, p->value);
   free(p);
   return NULL;
}

int main(int argc, char **argv)
{
   pthread_t                thread;
   int                       rc=0;
   pthread_attr_t          pta;
   thread_parm_t           *parm=NULL;

   /*************************************************************/
   /* 线程属性初始化 */
   printf("Create a thread attributes object\n");
   rc = pthread_attr_init(&pta);
```

```
    checkResults("pthread_attr_init()\n", rc);


    /*************************************************************/
    /* 用 NULL 属性的方式创建一个新线程 */
    printf("Create thread using the NULL attributes\n");
      /* 接口数据赋值 */
    parm = malloc(sizeof(thread_parm_t));
    parm->value = 5;
    strcpy(parm->string, "Inside secondary thread");
      /* 创建线程 */
    rc = pthread_create(&thread, NULL, threadfunc, (void *)parm);
    checkResults("pthread_create(NULL)\n", rc);


    /*************************************************************/
    /* 使用默认的线程属性创建一个新线程 */
    printf("Create thread using the default attributes\n");
      /* 接口数据赋值 */
    parm = malloc(sizeof(thread_parm_t));
    parm->value = 77;
    strcpy(parm->string, "Inside secondary thread");
      /* 创建线程 */
    rc = pthread_create(&thread, &pta, threadfunc, (void *)parm);
    checkResults("pthread_create(&pta)\n", rc);

    /*************************************************************/
    printf("Destroy thread attributes object\n");
    rc = pthread_attr_destroy(&pta);
    checkResults("pthread_attr_destroy()\n", rc);
    /* 通过 sleep() 的方式来等待线程结束，并不健壮
       这里仅是举例表示等待线程结束而已 */
    sleep(5);
    printf("Main completed\n");
    return 0;
}
```

7.7.2 pthread_exit 结束线程

语法：

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

功能：

这个函数用来结束一个运行中的线程，并返回该线程的状态。通常来说，这个状态是会

返回给创建这个子线程的线程。

当辅线程 return 的时候，系统会隐含地调用 pthread_exit 函数；

初始线程 return 的时候，系统则会隐含地调用 exit()函数

pthread_exit 函数与 exit()函数较为类似，不过只针对单条线程。

再次说明初始线程是较为特殊的，使用 pthread_exit()或是其它的方式结束初始线程，将会导致整个进程的结束。

当使用 return，或 pthread_exit（结束），或 cancellation（取消）的方式结束一个线程的时候，系统将会按照如下步骤进行处理：

1、 所以列入堆栈但未取出堆栈的，需要取消的程序，都会执行回滚操作，且该回滚不可撤消。(Any cancellation cleanup handlers that have been pushed and not popped will be executed in reverse order with cancellation disabled.)

2、 Data destructors are called for any thread specific data entries that have a non NULL value for both the value and the destructor.（不会译）

3、 结束线程

4、 线程的结束有可能导致系统执行 cancel 类的处理（寄存器中#pragma cancel_handler 的指令）

5、 初始线程的结束，将会导致系统结束该进程下所有的其它线程，然后清除活动作业组，调用 atexit()函数

6、 在被结束的线程中挂起的互斥体将处于“abandoned”状态，并且不再有效（Any mutexes that are held by a thread that terminates, become `abandoned' and are no longer valid）。之后如果其它的线程试图通过 pthread_mutex_lock 函数来获取这个互斥体时，将会死锁；试图通过 pthread_mutex_trylock 函数来获取这个互斥体时，将么返回 EBUSY 这个错误码。

7、 应用程序中可见的进程资源将不会被释放，这些资源包括但不仅限于：互斥体、文件描述符，或其它进程级的 cleanup 操作。

参数：

status 输入参数

表示该线程的状态

返回：

无返回

例子：

```
#define _MULTI_THREADED
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#define checkResults(string, val) {                    \
 if (val) {                                            \
   printf("Failed with %d at %s", val, string); \
   exit(1);                                            \
 }                                                     \
}

int   theStatus=5;
```

```
void *threadfunc(void *parm)
{
   printf("Inside secondary thread\n");
   pthread_exit(__VOID(theStatus));
   return __VOID(theStatus); /* Not needed, but this makes the compiler smile */
}

int main(int argc, char **argv)
{
   pthread_t                  thread;
   int                         rc=0;
   void                       *status;

   printf("Enter Testcase - %s\n", argv[0]);

   printf("Create thread using attributes that allow join\n");
   rc = pthread_create(&thread, NULL, threadfunc, NULL);
   checkResults("pthread_create()\n", rc);

   printf("Wait for the thread to exit\n");
   rc = pthread_join(thread, &status);
   checkResults("pthread_join()\n", rc);
   if (__INT(status) != theStatus) {
      printf("Secondary thread failed\n");
      exit(1);
   }

   printf("Got secondary thread status as expected\n");
   printf("Main completed\n");
   return 0;
}
```

7.7.3 pthread_join 等待线程结束并释放线程资源

语法：

```
#include <pthread.h>
  int pthread_join(pthread_t thread, void **status);
```

功能：

这个函数等待一个线程结束，分离这个线程（回收或不回收资源），然后返回线程的状态。

如果 status 参数为 NULL，则不返回线程的状态。

线程的返回状态通常是由应用程序指定的，但下面这两种情况除外：

1、 线程使用 pthread_cancel() 函数取消。这种情况下，返回的状态为 PTHREAD_CANCELLED

2、 线程被 exception 机制结束时，返回的状态为 PTHREAD_EXCEPTION_NP

最后，在线程属性中，detach 属性为 PTHREAD_CREATE_JOINABLE 这类线程使用 pthread_join, pthread_detach, pthread_extendedjoin_np 函数时，不需要特别指定 leaveThreadAllocated 这个选项，以便系统能回收分配给这些线程的资源。当对这类线程执行 join to 或是 deatch 操作如果失败的话，将会导致内存泄露，直到进程结束为止。

参数：

thread 输入参数

要操作线程的线程描述符，（在 pthread_create 中返回）

status 输出参数

接收到的线程状态返回变量的地址。

返回：

0 函数成功执行

非 0 函数执行失败

错误信息：

EINVAL 3021

函数参数不正确

ESRCH 3515

指定的线程不存在

例子：

```
#define _MULTI_THREADED
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#define checkResults(string, val) {                    \
 if (val) {                                                \
    printf("Failed with %d at %s", val, string); \
    exit(1);                                                \
 }                                                          \
}

int   okStatus            = 34;

void *threadfunc(void *parm)
{
   printf("Inside secondary thread\n");
   return __VOID(okStatus);
}

int main(int argc, char **argv)
{
   pthread_t                  thread;
   int                         rc=0;
   void                       *status;
```

```
    printf("Enter Testcase - %s\n", argv[0]);

    printf("Create thread using attributes that allow join\n");
    rc = pthread_create(&thread, NULL, threadfunc, NULL);
    checkResults("pthread_create()\n", rc);

    printf("Wait for the thread to exit\n");
    rc = pthread_join(thread, &status);
    checkResults("pthread_join()\n", rc);
    if (__INT(status) != okStatus) {
        printf("Secondary thread failed\n");
        exit(1);
    }

    printf("Got secondary thread status as expected\n");
    printf("Main completed\n");
    return 0;
}
```

7.7.4 pthread_join_np 等待线程结束

7.7.5 pthread_extendedjoin_np 根据一些扩展条件等待线程

7.7.6 pthread_cancel 取消线程

语法：

```
#include <pthread.h>
int pthread_cancel(pthread_t thread);
```

功能：

该函数用来取消一个指定的线程，所有基于该线程的应用都将被取消。

如果 cancelability 参数为 disable，那么所有的取消操作将被挂起，直到线程更改这个参数。

如果 cancelability 参数为 deferred，那么所有的取消操作将被挂起，直到线程更改该参数，同时调用 pthread_testcancel()产生一个取消操作的指针。

如果 cancelability 参数为 asynchronous，那么所有的取消操作将立刻执行，中断线程当前的。

不能通过调用 pthread_setcanceltype 函数，更改 PTREAD_CANCEL_ASYNCHRONOUS 的方式来取消一个异步的线程。

取消线程时，系统操作与结束线程类似。

参数：

thread 输入参数

要取消的线程的线程描述符

返回：

0 成功

非 0 失败

错误信息：

EINVAL 3021

函数参数不正确

ESRCH 3515

指定的线程不存在

例子：

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#define checkResults(string, val) {                    \
 if (val) {                                                \
    printf("Failed with %d at %s", val, string); \
    exit(1);                                                \
 }                                                            \
}

void *threadfunc(void *parm)
{
   printf("Entered secondary thread\n");
   while (1) {


      printf("Secondary thread is looping\n");
      pthread_testcancel();
      sleep(1);
   }
   return NULL;
}

int main(int argc, char **argv)
{
   pthread_t                  thread;
   int                         rc=0;

   printf("Entering testcase\n");

   /* Create a thread using default attributes */
```

```
    printf("Create thread using the NULL attributes\n");
    rc = pthread_create(&thread, NULL, threadfunc, NULL);
    checkResults("pthread_create(NULL)\n", rc);

    /* sleep() is not a very robust way to wait for the thread */
    sleep(2);

    printf("Cancel the thread\n");
    rc = pthread_cancel(thread);
    checkResults("pthread_cancel()\n", rc);

    /* sleep() is not a very robust way to wait for the thread */
    sleep(3);
    printf("Main completed\n");
    return 0;
}
```

8 线程特有数据 API Thread specific storage API

线程的特有数据，就是一个针对当前线程的全局变量(一个 pthread_key_t 类型的变量，其实也就是一个整型变量，其实也可以认为就是一个指针)，这个指针将指向某些已分配空间的地址。

我是这样理解：

在初始线程是使用 pthread_key_create 函数就类似于声明了这种特殊全局变量，即指针，在调用了这个函数之后，初始线程每创建一个子线程，子线程就会为这个指针创建一个当前子线程的线程内部共享的存储空间。同时 pthread_key_create 也会为这个指针分配一个销毁函数。也就是说，对于每个子线程来说，这个指针自身所存在的地址和它自身数值（即指向的地址）都是不同的。

然后在子线程内，需要首先通过 pthread_setspecific 函数将这个指针指向某片已分配存储空间的地址。

于是子线程内的各个函数可以通过 pthread_getspecfic 函数取出这个指针所指向的存储空间的数据。也就是通过这个特殊的，作用范围为单个线程的全局变量指针，实现了线程内部的全局存储空间的共享。

最后，子线程结束时，系统自动调用销毁函数，free()和 pthread_setspecific(key, NULL)是销毁函数中标准的处理方式，销毁函数中的其它语句可根据实际情况酌情撰写。

之所以要使用 pthread_setspecific 和 pthread_getspecific 函数来处理指针，而不是直接处理存储空间，当然是基于线程安全性的考虑。这两个函数都具备线程安全性。

8.1 pthread_key_create

语法：

```
#include <pthread.h>
int pthread_key_create( pthread_key_t *key,
                        void (*destructor)(void *));
```

功能：

这个函数创建了一个针对当前线程的线程特有数据 key，同时为这个 key 分配了一个销毁函数(destructor fuction)。这时，销毁函数的入口参数即与这个 key 对应起来。

key 创建之后，可以用来存放和读取（set & get）每个线程自己所拥有的数据的指针。

当线程结束时，如果 key 自身以及 key 所指向的值都不为空的话，那么系统将会调用销毁函数。我们应该在销毁函数中，将 key 值，以及 key 所指向的值都清空。

如上所述，销毁函数的参数，是当前 key。

不要在销毁函数中使用 pthread_exit 函数。

参数：

略

返回：

0 – 成功； 非 0 – 失败

例子：

在下面这个例子中，仅仅只是演示创建和删除线程特有数据的语法，并没有实际使用这个线程特有数据，所以销毁函数并没有被调用。

```
#define _MULTI_THREADED
#include <pthread.h>
#include <sched.h>
#include <stdio.h>
#define checkResults(string, val) {                    \
  if (val) {                                                \
     printf("Failed with %d at %s", val, string); \
     exit(1);                                               \
  }                                                          \
}

pthread_key_t             tlsKey = 0;

void globalDestructor(void *value)
{
   printf("In the data destructor\n");
   free(value);
   pthread_setspecific(tlsKey, NULL);
}
```

```
int main(int argc, char **argv)
{
   int                  rc=0;
   int                  i=0;

   printf("Enter Testcase - %s\n", argv[0]);

   printf("Create a thread local storage key\n");
   rc = pthread_key_create(&tlsKey, globalDestructor);
   checkResults("pthread_key_create()\n", rc);
   /* The key can now be used from all threads */

   printf("- The key can now be used from all threads\n");
   printf("- in the process to storage thread local\n");
   printf("- (but global to all functions in that thread)\n");
   printf("- storage\n");

   printf("Delete a thread local storage key\n");
   rc = pthread_key_delete(tlsKey);
   checkResults("pthread_key_delete()\n", rc);
   /* The key and any remaining values are now gone. */
   printf("Main completed\n");
   return 0;
}
```

8.2 pthread_key_delete

语法：

```
#include <pthread.h>
int pthread_key_delete(pthread_key_t key);
```

功能：

删除之前创建的线程的特有数据（也就是那个 key）。这个 delete 函数不会运行任何销毁函数。

这个 delete 函数通常在初始线程中进行删除。

参数：

略

返回：

0 – 正常； 非 0 – 失败

例子：

同 pthread_key_create，略

8.3 pthread_setspecific

语法：

```
#include <pthread.h>
int pthread_setspecific(pthread_key_t key,
                        const void *value);
```

功能：

将线程特有数据 key 值指向本地线程所分配的存储空间 value 中，以便线程内的各个函数都可以共享这个 value

参数：

key

输入参数

value

表明 KEY 所指向的存储空间

返回：

0 – 成功 ； 非 0 -- 失败

例子：

```
#define _MULTI_THREADED
#include <pthread.h>
#include <stdio.h>
#define checkResults(string, val) {                    \
 if (val) {                                            \
    printf("Failed with %d at %s", val, string); \
    exit(1);                                           \
 }                                                     \
}

#define              NUMTHREADS      3
pthread_key_t      tlsKey = 0;

void globalDestructor(void *value)
{
   printf("In global destructor\n");
   free(value);
   pthread_setspecific(tlsKey, NULL);
}


void showGlobal(void)
{
   void                        *global;
   pthread_id_np_t             tid;

   global    = pthread_getspecific(tlsKey);
```

```
    pthread_getunique_np((pthread_t *)global, &tid);
    printf("showGlobal: global data stored for thread 0x%.8x %.8x\n",
            tid);
}

void *threadfunc(void *parm)
{
    int                 rc;
    int                *myThreadDataStructure;
    pthread_t          me = pthread_self();

    printf("Inside secondary thread\n");

    myThreadDataStructure = malloc(sizeof(pthread_t) + sizeof(int) * 10);
    memcpy(myThreadDataStructure, &me, sizeof(pthread_t));
    pthread_setspecific(tlsKey, myThreadDataStructure);
    showGlobal();
    return NULL;
}

int main(int argc, char **argv)
{
    pthread_t                   thread[NUMTHREADS];
    int                         rc=0;
    int                         i=0;

    printf("Enter Testcase - %s\n", argv[0]);

    printf("Create a thread local storage key\n");
    rc = pthread_key_create(&tlsKey, globalDestructor);
    checkResults("pthread_key_create()\n", rc);
    /* The key can now be used from all threads */

    printf("Create %d threads using joinable attributes\n",
            NUMTHREADS);
    for (i=0; i<NUMTHREADS; ++i) {
        rc = pthread_create(&thread[i], NULL, threadfunc, NULL);
        checkResults("pthread_create()\n", rc);
    }

    printf("Join to threads\n");
    for (i=0; i<NUMTHREADS; ++i) {
        rc = pthread_join(thread[i], NULL);
        checkResults("pthread_join()\n", rc);
    }
```

```
    printf("Delete a thread local storage key\n");
    rc = pthread_key_delete(tlsKey);
    checkResults("pthread_key_delete()\n", rc);
    /* The key and any remaining values are now gone. */
    printf("Main completed\n");
    return 0;
}
```

8.4 pthread_getspecific

语法：

```
#include <pthread.h>
void *pthread_getspecific(pthread_key_t key);
```

功能：

根据 key，取出线程内部共享的数据。

需要注意，这个函数返回的是共享数据的地址。

例子见 pthread_setspecific， 其它略。

9 取消线程 API Thread cancellation API

9.1 pthread_cancel

9.2 pthread_cleanup_peek_np

9.3 pthread_cleanup_pop

9.4 pthread_cleanup_push

9.5 pthread_getcancelstate_np

9.6 pthread_setcancelstate

9.7 pthread_setcanceltype

9.8 pthread_test_exit_np

9.9 pthread_testcancel

10 条件变量的 API

所谓条件变量，其实就是一个信号。线程在执行 pthread_cond_wait 函数时将当前线程挂起，等待别的线程发出信号；别的线程执行 pthread_cond_signal 或 pthread_cond_broadcast 函数时，向系统发出一个信号，然后挂起的线程收到这个信号，于是就被唤醒，继续操作。基于这种原理，我们就可以在应用程序中，针对不同的情况或事件来发出这个信号，也就可以实现根据某个事件，或某个数值来唤醒我们所之前挂起的线程。

条件变量需要与互斥体配合使用，详见 pthread_cond_wait。

每个条件变量最后都必须使用 pthread_cond_destory 函数来销毁它。

当一个条件创建之后，就不能再 COPY 和或 MOVE 到新地址中，否则将不能再有效使用。

10.1 pthread_cond_init

语法：

#include <pthread.h>
int pthread_cond_init(pthread_cond_t *cond,
const pthread_condattr_t *attr);
或
pthread_cond_t cond = PTHREAD_COND_INITIALIZER

功能：

对条件目标进行初始化。如果 attr 参数设置为 NULL，那么将使用默认参数。
如果已有如下定义：

```
pthread_cond_t      cond2;
pthread_cond_t      cond3;
pthread_condattr_t  attr;
pthread_condattr_init(&attr);
```

那么下面这三句语句是等效的，都是对条件变量进行初始化，使用默认参数

```
pthread_cond_t      cond1 = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_init(&cond2, NULL);
pthread_cond_init(&cond3, &attr);
```

但是使用 PTHREAD_COND_INITIALIZER 这种方式对条件变量进行初始化，将不会立刻生效，而是在其后首次使用 pthread_cond_wait 或 pthread_cond_timewait 或 pthread_cond_signal，或 pthread_cond_broadcast 函数时才会进行条件变量初始化。如果没有使用这些函数，就直接用 pthread_cond_destroy 函数，系统将会报一个 EINVAL 的错误信息。

参数：

cond
　　条件变量的地址
attr
　　初始化条件变量的 attr 的地址

返回：

0 – 成功；　非 0 -- 失败

例子：

下面这个例子，仅仅只是使用不同的方法，对几个条件变量进行了初始化，然后再销毁，并没有真正使用条件变量进行应用处理。

```
#define _MULTI_THREADED
#include <pthread.h>
#include <stdio.h>
#include "check.h"

pthread_cond_t      cond1 = PTHREAD_COND_INITIALIZER;
pthread_cond_t      cond2;
```

```
pthread_cond_t          cond3;

int main(int argc, char **argv)
{
   int                          rc=0;
   pthread_condattr_t       attr;

   printf("Enter Testcase - %s\n", argv[0]);

   printf("Create the default cond attributes object\n");
   rc = pthread_condattr_init(&attr);
   checkResults("pthread_condattr_init()\n", rc);

   printf("Create the all of the default conditions in different ways\n");
   rc = pthread_cond_init(&cond2, NULL);
   checkResults("pthread_cond_init()\n", rc);

   rc = pthread_cond_init(&cond3, &attr);
   checkResults("pthread_cond_init()\n", rc);

   printf("- At this point, the conditions with default attributes\n");
   printf("- Can be used from any threads that want to use them\n");

   printf("Cleanup\n");
   pthread_condattr_destroy(&attr);
   pthread_cond_destroy(&cond1);
   pthread_cond_destroy(&cond2);
   pthread_cond_destroy(&cond3);

   printf("Main completed\n");
   return 0;
}
```

10.2 pthread_cond_wait

语法：

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

功能：

这个函数将会阻塞当前线程，等待条件产生（即某个线程发出信号）。

当执行这个函数时，必须先 lock 住指定的互斥体 mutex，在执行到 pthread_cond_wait 函数的语句时，将会对互斥体进行解锁操作，然后开始等待。

当等待的条件满足，或者线程被 cancel 了（线程被 cancel 时，虽然不会再执行应用程

序的语句，但系统仍会就当前线程进行处理），在线程继续执行之前，当前线程将会首先自动获取对互斥体的所有权（即锁住互斥体）。如果之前，当前线程未对互斥体执行 lock 操作，那么将会返回一个 EPERM 的错误信息。如果此时别的线程锁住了互斥体，当前线程将处理一个等待对方对互斥体解锁的状态中。在一个时点上，只能对一个条件变量分配一个互斥体。在同一时间对一个条件变量分配两个互斥体将会导致应用程序出现无法预测的串行问题。

Pthread_cond_wait 这个函数可以被取消。

基于条件变量的原理，为了确保发出信号时，不会失效（即 pthread_cond_wait 操作在 pthread_cond_signal 之后发生），最好使用一个布尔型变量进行判断

参数：

cond

输入参数，条件变量的地址

mutex

输入参数，分配给条件变量的互斥体的地址

返回：

0 – 成功； 非 0 –失败

例子：

在下面这个例子中，程序的执行流程大致上是这样的：

子线程 1 锁住互斥体，然后执行 pthread_cond_wait，解锁，等待；

子线程 2 锁住互斥体，然后执行 pthread_cond_wait，解锁，等待；

初始线程等两个子线程都开始执行 pthread_cond_wait（这里直接使用了 sleep 来实现等待的目的。原程序段中也说明了，sleep 并不是一个很健壮的写法，这里只是举例）

然后锁住互斥体，更新关键数据 conditionMet（子线程根据这个值判断是否结果循环）

发出广播，唤醒所有等待的线程

然后初始线程对互斥体解锁（必须要解锁，不然子线程唤醒后重新获取互斥体的所有权，即对互斥体进行锁操作，而此时互斥体又被初始线程锁住，然后初始线程又等待子线程结束，于是形成死锁）

子线程 1 和子线程 2 同时被唤醒

子线程 1 获得互斥体的所有权，继续向下执行；子线程 2 在等待获取互斥体所有权。

子线程 1 判断循环的条件满足，退出循环，对互斥体解锁；

子线程 2 获取互斥体的所有权，也退出循环，对互斥体解锁。

最后初始线程等到了两个子线程的操作，回收资源，程序结束。

```
#define _MULTI_THREADED
#include <pthread.h>

#include <stdio.h>
#define checkResults(string, val) {                    \
  if (val) {                                           \
    printf("Failed with %d at %s", val, string); \
    exit(1);                                           \
  }                                                    \
}

/* For safe condition variable usage, must use a boolean predicate and      */
/* a mutex with the condition.                                              */
```

```
int                 conditionMet = 0;
pthread_cond_t      cond    = PTHREAD_COND_INITIALIZER;
pthread_mutex_t     mutex = PTHREAD_MUTEX_INITIALIZER;

#define NTHREADS       5

void *threadfunc(void *parm)
{
   int              rc;

   rc = pthread_mutex_lock(&mutex);
   checkResults("pthread_mutex_lock()\n", rc);

   while (!conditionMet) {
      printf("Thread blocked\n");
      rc = pthread_cond_wait(&cond, &mutex);
      checkResults("pthread_cond_wait()\n", rc);
   }

   rc = pthread_mutex_unlock(&mutex);
   checkResults("pthread_mutex_lock()\n", rc);
   return NULL;
}

int main(int argc, char **argv)
{
   int                  rc=0;
   int                  i;
   pthread_t            threadid[NTHREADS];

   printf("Enter Testcase - %s\n", argv[0]);

   printf("Create %d threads\n", NTHREADS);
   for(i=0; i<NTHREADS; ++i) {
      rc = pthread_create(&threadid[i], NULL, threadfunc, NULL);
      checkResults("pthread_create()\n", rc);
   }

   sleep(5);    /* Sleep is not a very robust way to serialize threads */
   rc = pthread_mutex_lock(&mutex);
   checkResults("pthread_mutex_lock()\n", rc);

   /* The condition has occured. Set the flag and wake up any waiting threads */
   conditionMet = 1;
   printf("Wake up all waiting threads...\n");
```

```
    rc = pthread_cond_broadcast(&cond);
    checkResults("pthread_cond_broadcast()\n", rc);

    rc = pthread_mutex_unlock(&mutex);
    checkResults("pthread_mutex_unlock()\n", rc);

    printf("Wait for threads and cleanup\n");
    for (i=0; i<NTHREADS; ++i) {
      rc = pthread_join(threadid[i], NULL);
      checkResults("pthread_join()\n", rc);
    }
    pthread_cond_destroy(&cond);
    pthread_mutex_destroy(&mutex);

    printf("Main completed\n");
    return 0;
}
```

10.3 pthread_cond_signal

语法：

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

功能：

发出信号，用来唤起至少一个之前挂起的线程（具体作用范围不详，可以认为就只能唤起一个线程？）。如果没有符合条件的线程，那么这个发出的信号将会作废，没有影响到任何处理。

在执行 pthread_cond_wait 函数时，需要为条件变量分配一个互斥体。当前线程无论是否拥有互斥体（即是否锁住），都可以执行 pthread_cond_signal 这个函数。当然，如果应用程序需要的话，也可以在调用 pthread_cond_signal 这个函数之前锁住互斥体。

参数：

cond

输入参数，条件变量的地址

返回：

0 – 成功； 非 0 – 失败

例子：

```
#define _MULTI_THREADED
#include <pthread.h>
#include <stdio.h>
#define checkResults(string, val) {                    \
  if (val) {                                                  \
    printf("Failed with %d at %s", val, string); \
    exit(1);                                                  \
  }                                                             \
```

```
}

/* For safe condition variable usage, must use a boolean predicate and    */
/* a mutex with the condition.                                            */
int                 workToDo = 0;
pthread_cond_t      cond    = PTHREAD_COND_INITIALIZER;
pthread_mutex_t     mutex = PTHREAD_MUTEX_INITIALIZER;

#define NTHREADS        2

void *threadfunc(void *parm)
{
   int              rc;

   while (1) {
      /* Usually worker threads will loop on these operations */
      rc = pthread_mutex_lock(&mutex);
      checkResults("pthread_mutex_lock()\n", rc);

      while (!workToDo) {
         printf("Thread blocked\n");
         rc = pthread_cond_wait(&cond, &mutex);
         checkResults("pthread_cond_wait()\n", rc);
      }
      printf("Thread awake, finish work!\n");

      /* Under protection of the lock, complete or remove the work        */
      /* from whatever worker queue we have. Here it is simply a flag     */
      workToDo = 0;

      rc = pthread_mutex_unlock(&mutex);
      checkResults("pthread_mutex_lock()\n", rc);
   }
   return NULL;
}

int main(int argc, char **argv)
{
   int                  rc=0;
   int                  i;
   pthread_t            threadid[NTHREADS];

   printf("Enter Testcase - %s\n", argv[0]);

   printf("Create %d threads\n", NTHREADS);
```

```
    for(i=0; i<NTHREADS; ++i) {
        rc = pthread_create(&threadid[i], NULL, threadfunc, NULL);
        checkResults("pthread_create()\n", rc);
    }

    sleep(5);    /* Sleep is not a very robust way to serialize threads      */

    for(i=0; i<5; ++i) {
        printf("Wake up a worker, work to do...\n");

        rc = pthread_mutex_lock(&mutex);
        checkResults("pthread_mutex_lock()\n", rc);

        /* In the real world, all the threads might be busy, and              */
        /* we would add work to a queue instead of simply using a flag    */
        /* In that case the boolean predicate might be some boolean          */
        /* statement like: if (the-queue-contains-work)                          */
        if (workToDo) {
            printf("Work already present, likely threads are busy\n");
        }
        workToDo = 1;
        rc = pthread_cond_signal(&cond);
        checkResults("pthread_cond_broadcast()\n", rc);

        rc = pthread_mutex_unlock(&mutex);
        checkResults("pthread_mutex_unlock()\n", rc);
        sleep(5);    /* Sleep is not a very robust way to serialize threads */
    }

    printf("Main completed\n");
    exit(0);
    return 0;
}
```

10.4 pthread_cond_timedwait

语法：

```
#include <pthread.h>
#include <time.h>
int pthread_cond_timedwait(pthread_cond_t *cond,
                           pthread_mutex_t *mutex,
                           const struct timespec *abstime);
```

功能：

与 phtread_cond_wait 类似，但增加了超时设置，即超过指定时间后，该函数会返回一个 ETIMEOUT 的错误信息。

参数：

其它略。

Abstime 是一个独立的系统时间，注意设置方式。

返回：

0 – 成功； ETIMEOUT – 超进退出； 其它 – 失败

例子：

这个例子中，设置了超时等待的时间为 15 秒，注意对时间的处理。

```
#define _MULTI_THREADED
#include <stdio.h>
#include <qp0z1170.h>
#include <time.h>
#include <pthread.h>
#define checkResults(string, val) {                    \
 if (val) {                                            \
    printf("Failed with %d at %s", val, string); \
    exit(1);                                           \
 }                                                     \
}

/* For safe condition variable usage, must use a boolean predicate and    */
/* a mutex with the condition.                                            */
int                 workToDo = 0;
pthread_cond_t      cond    = PTHREAD_COND_INITIALIZER;
pthread_mutex_t     mutex = PTHREAD_MUTEX_INITIALIZER;

#define NTHREADS                3
#define WAIT_TIME_SECONDS       15

void *threadfunc(void *parm)
{
   int                  rc;
   struct timespec      ts;
   struct timeval       tp;

   rc = pthread_mutex_lock(&mutex);
   checkResults("pthread_mutex_lock()\n", rc);

   /* Usually worker threads will loop on these operations */
   while (1) {
      rc =    gettimeofday(&tp, NULL);
      checkResults("gettimeofday()\n", rc);

      /* Convert from timeval to timespec */
      ts.tv_sec    = tp.tv_sec;
      ts.tv_nsec = tp.tv_usec * 1000;
```

```
        ts.tv_sec += WAIT_TIME_SECONDS;

        while (!workToDo) {
            printf("Thread blocked\n");
            rc = pthread_cond_timedwait(&cond, &mutex, &ts);
            /* If the wait timed out, in this example, the work is complete, and      */
            /* the thread will end.                                                   */
            /* In reality, a timeout must be accompanied by some sort of checking    */
            /* to see if the work is REALLY all complete. In the simple example       */
            /* we will just go belly up when we time out.                             */
            if (rc == ETIMEDOUT) {
                printf("Wait timed out!\n");
                rc = pthread_mutex_unlock(&mutex);
                checkResults("pthread_mutex_lock()\n", rc);
                pthread_exit(NULL);
            }
            checkResults("pthread_cond_timedwait()\n", rc);
        }

        printf("Thread consumes work here\n");
        workToDo = 0;
    }

    rc = pthread_mutex_unlock(&mutex);
    checkResults("pthread_mutex_lock()\n", rc);
    return NULL;
}

int main(int argc, char **argv)
{
    int                 rc=0;
    int                 i;
    pthread_t           threadid[NTHREADS];

    printf("Enter Testcase - %s\n", argv[0]);

    printf("Create %d threads\n", NTHREADS);
    for(i=0; i<NTHREADS; ++i) {
        rc = pthread_create(&threadid[i], NULL, threadfunc, NULL);
        checkResults("pthread_create()\n", rc);
    }

    rc = pthread_mutex_lock(&mutex);
    checkResults("pthread_mutex_lock()\n", rc);
```

```
    printf("One work item to give to a thread\n");
    workToDo = 1;
    rc = pthread_cond_signal(&cond);
    checkResults("pthread_cond_signal()\n", rc);

    rc = pthread_mutex_unlock(&mutex);
    checkResults("pthread_mutex_unlock()\n", rc);

    printf("Wait for threads and cleanup\n");
    for (i=0; i<NTHREADS; ++i) {
        rc = pthread_join(threadid[i], NULL);
        checkResults("pthread_join()\n", rc);
    }

    pthread_cond_destroy(&cond);
    pthread_mutex_destroy(&mutex);
    printf("Main completed\n");
    return 0;
}
```

10.5 pthread_cond_broadcast

与 pthread_cond_signal 函数类似，不同点仅在于 pthread_cond_broadcast 是用来唤起所有根据该条件变量挂起的线程。而 pthread_cond_signal 只能唤起至少一个（多数时候就只是一个），达不到所有的效果。

其它略。

10.6 pthread_cond_destroy

用来销毁一个已分配资源的条件变量。如果该条件变量在使用中（即有其它线程通过该条件变量挂起进），系统将会返回一个 EBUSY 的错误信息。

其它略。

11 读/写锁的同步 API Read/write lock synchronization API

11.10 pthread_rwlockattr_destroy

11.11 pthread_rwlockattr_getpshared

11.12 pthread_rwlockattr_init

11.13 pthread_rwlockattr_setpshared

12 其它 API -- Signal APIs

12.1 pthread_kill

12.2 pthread_sigmask

12.3 pthread_signal_to_cancel_np

13 互斥体 API

13.1 互斥体操作 API -- Mutex Operation API

互斥体是最简单的一种实现对共享资源保护的方法。当一个线程成功 lock 了互斥体之后，该线程就成为这个互斥体的拥有者；当其它的线程试图对互斥体执行 lock 操作时，将不会成功，直到互斥体的拥有者对互斥体执行 unlock 操作。

也就是说互斥体并不作用于任何变量，它仅仅用来控制是否阻塞线程。

每个创建的互斥体最后都必须使用 pthread_mutex_destory 函数来销毁它。系统会检查互斥体是否被销毁。大量的互斥体使用完毕而未销毁，将会影响系统性能。所以在释放、回收利用互斥体的存储空间之前，一定要使用 phthread_mutex_destory 函数进行销毁。

当一个互斥体创建之后，就不能再 COPY 和或 MOVE 到新地址中，否则将不能再有效使用。

下表列出互斥体的主要属性，以此这些属性的默认值，可用的值。

Attribute	Default value	Supported values

pshared	PTHREAD_PROCESS_PRIVATE	PTHREAD_PROCESS_PRIVATE or PTHREAD_PROCESS_SHARED
kind (non portable)	PTHREAD_MUTEX_NONRECURSIV E_NP	PTHREAD_MUTEX_NONRECURSIVE_NP or PTHREAD_MUTEX_RECURSIVE_NP
name (non portable)	PTHREAD_DEFAULT_MUTEX_NAM E_NP "QP0WMTX UNNAMED"	Any name that is 15 characters or less. If not terminated by a null character, name is truncated to 15 characters.
type	PTHREAD_MUTEX_DEFAULT (PTHREAD_MUTEX_NORMAL)	PTHREAD_MUTEX_DEFAULT or PTHREAD_MUTEX_NORMAL or PTHREAD_MUTEX_RECURSIVE or PTHREAD_MUTEX_ERRORCHECK or PTHREAD_MUTEX_OWNERTERM_NP The PTHREAD_MUTEX_OWNERTERM_NP attribute value is non portable.

13.1.1 pthread_lock_global_np

语法：

#include <pthread.h>

int pthread_lock_global_np(void);

功能：

这个函数将会锁住一个在线程运行时，由系统提供的全局互斥体。这是个递归的互斥体，名字为“QP0W_GLOBAL_MTX”。

递归时，最大的锁次数为 32767，超过这个数量时，将会返回 ERECURSE 的错误码。

参数：

无

返回：

0 – 成功

非 0 -- 失败

例子：

注意下面例子的使用，对于这个系统创建的互斥体，我们不需要对其进行定义。同时互斥体的调用是可以递归的，注意到在子线程中，先对互斥体进行了 lock 操作，然后子线程调用的函数中再次对互斥体进行了 lock 操作。

这种可以递归的互斥体，在同一线程中的 lock 与 unlock 操作一定要匹配。

同时，递归仅限于当前线程之内。

在这个例子中，如果调用时不带参数，那么各个子线程实际将会执行串行操作，最后得到正确的结果。

Give any number of parameters to show data corruption

Creating 10 threads

Wait for results

```
Using 10 threads and LOOPCONSTANT = 5000
Values are: (should be 50000)
 ==>50000, 50000, 50000, 50000
Main completed
```

如果调用时带了参数（随便什么都可以），那么各个子线程将会实现并发操作，最后得到一个预期以外的结果（结果应该是随机的，但一定小于正确的结果）：

```
Give any number of parameters to show data corruption
A parameter was specified, no serialization is being done!
Creating 10 threads
Wait for results
Using 10 threads and LOOPCONSTANT = 5000
Values are: (should be 50000)
 ==>34785, 37629, 48219, 47632
Main completed
```

```
#include <pthread.h>
#include <stdio.h>
#define checkResults(string, val) {                    \
 if (val) {                                                \
    printf("Failed with %d at %s", val, string); \
    exit(1);                                                \
 }                                                            \
}
/*
   This example shows the corruption that can result if no
   serialization is done and also shows the use of
   pthread_lock_global_np(). Call this test with no parameters
   to use pthread_lock_gloabl_np() to protect the critical data,
   between more than one (possibly unrelated) functions.
   Use 1 or more parameters to skip locking and
   show data corruption that occurs without locking.
    */
#define                LOOPCONSTANT        5000
#define                THREADS               10

int                       i,j,k,l;
int                       uselock=1;

void secondFunction(void)
{
   int rc;
   if (uselock) {
      rc = pthread_lock_global_np();
      checkResults("pthread_lock_global_np()\n", rc);
```

```
    }
    --i; --j; --k; --l;
    if (uselock) {
        rc = pthread_unlock_global_np();
        checkResults("pthread_unlock_global_np()\n", rc);
    }
}

void *threadfunc(void *parm)
{
    int     loop = 0;
    int     rc;

    for (loop=0; loop<LOOPCONSTANT; ++loop) {
        if (uselock) {
            rc = pthread_lock_global_np();
            checkResults("pthread_lock_global_np()\n", rc);
        }
        ++i; ++j; ++k; ++l;
        secondFunction();
        ++i; ++j; ++k; ++l;
        if (uselock) {
            rc = pthread_unlock_global_np();
            checkResults("pthread_unlock_global_np()\n", rc);
        }
    }
    return NULL;
}

int main(int argc, char **argv)
{
    pthread_t                   threadid[THREADS];
    int                          rc=0;
    int                          loop=0;

    printf("Enter Testcase - %s\n", argv[0]);
    printf("Give any number of parameters to show data corruption\n");
    if (argc != 1) {
        printf("A parameter was specified, no serialization is being done!\n");
        uselock = 0;
    }

    if (uselock) {
        rc = pthread_lock_global_np();
        checkResults("pthread_lock_global_np() (main)\n", rc);
```

```
    }

    printf("Creating %d threads\n", THREADS);
    for (loop=0; loop<THREADS; ++loop) {
      rc = pthread_create(&threadid[loop], NULL, threadfunc, NULL);
      checkResults("pthread_create()\n", rc);
    }

    sleep(5);
    if (uselock) {
      rc = pthread_unlock_global_np();
      checkResults("pthread_unlock_global_np() (main)\n", rc);
    }

    printf("Wait for results\n");
    for (loop=0; loop<THREADS; ++loop) {
      rc = pthread_join(threadid[loop], NULL);
      checkResults("pthread_join()\n", rc);
    }

    printf("\nUsing %d threads and LOOPCONSTANT = %d\n",
           THREADS, LOOPCONSTANT);
    printf("Values are: (should be %d)\n", THREADS * LOOPCONSTANT);
    printf(" ==>%d, %d, %d, %d\n", i, j, k, l);

    printf("Main completed\n");
    return 0;
  }
```

13.1.2 pthread_unlock_global_np

与 pthread_lock_globak_np 的使用方法类似，只不过是用于对全局互斥体的解锁操作，略。

13.1.3 pthread_mutex_init 互斥体初始化

语法：

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

或

```
pthread_mutex_t   mutex = PTHREAD_MUTEX_INITIALIZER;
```

功能：

对互斥体进行初始化。如果 attr 参数调为 NULL，互斥体属性将会设置成为默认参数。

当已具备如下定义后：

```
pthread_mutex_t        mutex2;
pthread_mutex_t        mutex3;
pthread_mutexattr_t    mta;
pthread_mutexattr_init(&mta);
```

下面这三种对互斥体进行初始化的方法是等价的，它们都将互斥体初始化为默认属性。

```
pthread_mutex_t        mutex1 = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_init(&mutex2, NULL);
pthread_mutex_init(&mutex3, &mta);
```

但是使用 PTHREAD_MUTEX_INITIALIZER 这种方式对互斥体进行初始化，将不会立刻生效，而是在其后首次使用 pthread_mutex_lock 或 pthread_mutex_trylock 函数时才会进行互斥体初始化。这是因为一个互斥体并不仅仅只是一个存储目标，它还需要系统分配相应的资源。于是，用这种方法对互斥体进行初始化，如果在互斥体没有被锁住时就执行 pthread_mutex_destroy,或 pthread_mutex_unlock 操作，系统将会报一个 EINVAL 的错误信息。

参数：

mutex

一个互斥体目标的地址。

Attr

标识互斥体属性目标的地址。（即互斥体属性结构体的地址位），可以为 NULL

返回：

0 -- 成功

非 0 -- 失败

例子：

略

13.1.4 pthread_mutex_lock 互斥体锁

语法：

```
#include <pthread.h>
int pthread_mutex_lock(pthread_mutex_t *mutex)
```

功能：

当一个线程对互斥体成功执行了 pthread_mutex_lock 操作后，其它的线程再对这个互斥体执行 lock 操作时，将会被阻塞，直到当前线程对互斥体进行 unlock 操作。然后就会有另一个等待执行 lock 操作的线程再对互斥体进行 lock 操作。

参数：

mutex

要锁住的互斥体的地址。

返回：

0 成功； 非 0 失败

例子：

下面这个例子中，通过使用互斥体，实现各个子线程实际上的串行。

当没有调用参数时，程序会使用互斥体，来实际各个子线程实际上的串行，达到对关键数据保护的目的，最后将会计算得出预期中正确的结果。

如果有调用参数（随便什么参数均可），则程序将不会使用互斥体，子线程之间并行，对关键数据没有保护，此时将会计算出一个非预期的随机的错误结果。

```
#include <pthread.h>
#include <stdio.h>
#define checkResults(string, val) {                    \
 if (val) {                                                \
    printf("Failed with %d at %s", val, string); \
    exit(1);                                                \
 }                                                          \
}

#define                 LOOPCONSTANT        10000
#define                 THREADS             10

pthread_mutex_t       mutex = PTHREAD_MUTEX_INITIALIZER;
int                     i,j,k,l;
int                     uselock=1;

void *threadfunc(void *parm)
{
   int     loop = 0;
   int     rc;

   for (loop=0; loop<LOOPCONSTANT; ++loop) {
      if (uselock) {
         rc = pthread_mutex_lock(&mutex);
         checkResults("pthread_mutex_lock()\n", rc);
      }
      ++i; ++j; ++k; ++l;
      if (uselock) {
         rc = pthread_mutex_unlock(&mutex);
         checkResults("pthread_mutex_unlock()\n", rc);
      }
   }
   return NULL;
}

int main(int argc, char **argv)
{
   pthread_t                   threadid[THREADS];
```

```
    int                 rc=0;

    int                 loop=0;
    pthread_attr_t      pta;

    printf("Entering testcase\n");
    printf("Give any number of parameters to show data corruption\n");
    if (argc != 1) {
       printf("A parameter was specified, no serialization is being done!\n");
       uselock = 0;
    }

    pthread_attr_init(&pta);
    pthread_attr_setdetachstate(&pta, PTHREAD_CREATE_JOINABLE);

    printf("Creating %d threads\n", THREADS);
    for (loop=0; loop<THREADS; ++loop) {
       rc = pthread_create(&threadid[loop], &pta, threadfunc, NULL);
       checkResults("pthread_create()\n", rc);
    }

    printf("Wait for results\n");
    for (loop=0; loop<THREADS; ++loop) {
       rc = pthread_join(threadid[loop], NULL);
       checkResults("pthread_join()\n", rc);
    }

    printf("Cleanup and show results\n");
    pthread_attr_destroy(&pta);
    pthread_mutex_destroy(&mutex);

    printf("\nUsing %d threads and LOOPCONSTANT = %d\n",
           THREADS, LOOPCONSTANT);
    printf("Values are: (should be %d)\n", THREADS * LOOPCONSTANT);
    printf("    ==>%d, %d, %d, %d\n", i, j, k, l);

    printf("Main completed\n");
    return 0;
}
```

13.1.5 pthread_mutex_unlock 互斥体解锁

与 pthread_mutex_lock 类似，只不过是解锁指令，略

13.1.6 pthread_mutex_destroy 销毁互斥体

语法：

#include <pthread.h>

int pthread_mutex_destroy(pthread_mutex_t *mutex)

功能：

销毁指定的互斥体，销毁后的互斥体将不能再使用。

互斥体只能由它的拥有者来销毁，如果别的线程锁住互斥体，当前线程执行 pthread_mutex_destroy 函数时，将会得到一个 EBUSY 的错误信息。

如果销毁互斥体时，其它的通过调用 pthread_mutex_lock 函数，正处于阻塞状态中的线程将会由系统返回信息，并得到一个 EDESTROYED 的错误信息。

参数：

mutex

互斥体结构的地址。

返回：

0 – 正常； 非 0 -- 失败

例子：

见 pthread_mutex_lock 中的例子，略。

13.1.7 pthread_mutex_timedlock_np (带超时设置的互斥体锁)

语法：

#include <pthread.h>

#include <time.h>

int pthread_mutex_timedlock_np(pthread_mutex_t *mutex,
const struct timespec *deltatime);

功能：

带超时设置的互斥体的 lock 操作。即如果当前互斥体已被别的线程锁住，那么当前线程会阻塞等待，超过指定的时间之后，将会返回一个 EBUSY 的返回信息码。

参数：

mutex

指定的互斥体

deltatime

指定的超时时长

返回：

0 -- 成功； EBUSY(3029) – 超时退出； 其它值 – 其它错误；

例子：

这个例子中，主线程一直锁住了互斥体，如果直接使用 pthread_mutex_lock 操作的话，子线程将会与直等待，而主线程又等待子线程的结束，结果将会造成死锁。

所以说，合理使用 pthread_mutex_timedlock_np，以及 pthread_mutex_trylock，可以有效的避免死锁。

```
        #define _MULTI_THREADED
#include <pthread.h>
#include <stdio.h>
#include <time.h>
#define checkResults(string, val) {                    \
 if (val) {                                            \
    printf("Failed with %d at %s", val, string); \
    exit(1);                                           \
 }                                                     \
}

pthread_mutex_t       mutex = PTHREAD_MUTEX_INITIALIZER;

void *threadFunc(void *parm)
{
   int          rc;
   int          i;
   struct timespec deltatime;

   deltatime.tv_sec = 5;
   deltatime.tv_nsec = 0;

   printf("Timed lock the mutex from a secondary thread\n");
   rc = pthread_mutex_timedlock_np(&mutex, &deltatime);
   if (rc != EBUSY) {
        printf("Got an incorrect return code from pthread_mutex_timedlock_np\n");
   }
   printf("Thread mutex timeout\n");
   return 0;
}

int main(int argc, char **argv)
{
   int                          rc=0;
   pthread_t                    thread;

   printf("Enter Testcase - %s\n", argv[0]);

   printf("Acquire the mutex in the initial thread\n");
   rc = pthread_mutex_lock(&mutex);
   checkResults("pthread_mutex_lock()\n", rc);

   printf("Create a thread\n");
   rc = pthread_create(&thread, NULL, threadFunc, NULL);
   checkResults("pthread_create()\n", rc);
```

```
    printf("Join to the thread\n");
    rc = pthread_join(thread, NULL);
    checkResults("pthread_join()\n", rc);

    printf("Destroy mutex\n");
    pthread_mutex_destroy(&mutex);

    printf("Main completed\n");
    return 0;
}
```

13.1.8 pthread_mutex_trylock （不进行阻塞处理的互斥体锁）

语法：

```
#include <pthread.h>
int pthread_mutex_trylock (pthread_mutex_t *mutex)
```

功能：

这个函数试图锁住一个互斥体，但并不进行阻塞处理。也就是如果执行时，发现该互斥体已被别的线程锁住时，函数将不会阻塞，而是返回一个 EBUSY 的错误信息。

如果当前线程已锁住这个互斥体时，函数将会返回一个 EDEADLK 的错误信息。

参数、返回与函数 pthread_mutex_lock 相同，略。

例子：

这个例子比较具有实用，因为通过 pthread_mutex_lock 来控制保护关键数据，造成了对关键数据的操作实质上是处于串行中。

此处通过非阻塞的 pthread_mutex_trylock，充分利用了各个线程自己的资源去进行计算，同时尽可能的实时更改关键数据。最后，计算出各个线程本地计算量的百分比。得出的结果显示大部分计算量都是由各个线程自己完成，程序效率大大提高。

最后输出的参考结果：

```
Creating 10 threads
Wait for results
Thread processed about 95% of the problem locally
Thread processed about 93% of the problem locally
Thread processed about 88% of the problem locally
Thread processed about 85% of the problem locally
Thread processed about 98% of the problem locally
Thread processed about 95% of the problem locally
Thread processed about 94% of the problem locally
Thread processed about 91% of the problem locally
Thread processed about 81% of the problem locally
Thread processed about 60% of the problem locally
```

```
Cleanup and show results
Using 10 threads and LOOPCONSTANT = 100000
Values are: (should be 1000000)
    ==>1000000, 1000000, 1000000, 1000000
 Main completed

#include <pthread.h>
#include <stdio.h>
#include <errno.h>
#define checkResults(string, val) {                    \
 if (val) {                                                \
    printf("Failed with %d at %s", val, string); \
    exit(1);                                                \
 }                                                            \
}

/*
   This example simulates a number of threads working on a parallel
   problem. The threads use pthread_mutex_trylock() so that
   they do not spend time blocking on a mutex and instead spend more
   of the time making progress towards the final solution. When
   trylock fails, the processing is done locally, eventually to
   be merged with the final parallel solution.

   This example should complete faster than the example for
   pthread_mutex_lock() in which threads solve the same parallel
   problem but spend more time waiting in resource contention.
   */
#define                 LOOPCONSTANT        10000
#define                 THREADS             10

pthread_mutex_t      mutex = PTHREAD_MUTEX_INITIALIZER;
int                  i,j,k,l;

void *threadfunc(void *parm)
{
   int     loop = 0;
   int     localProcessingCompleted = 0;
   int     numberOfLocalProcessingBursts = 0;
   int     processingCompletedThisBurst = 0;
   int     rc;

   for (loop=0; loop<LOOPCONSTANT; ++loop) {
      rc = pthread_mutex_trylock(&mutex);
      if (rc == EBUSY) {
```

```
            /* Process continue processing the part of the problem      */
            /* that we can without the lock. We do not want to waste */
            /* time blocking. Instead, we'll count locally.                  */
            ++localProcessingCompleted;
            ++numberOfLocalProcessingBursts;
            continue;
        }
        /* We acquired the lock, so this part of the can be global*/
        checkResults("pthread_mutex_trylock()\n", rc);
        /* Processing completed consist of last local processing    */
        /* plus the 1 unit of processing this time through               */
        processingCompletedThisBurst = 1 + localProcessingCompleted;
        localProcessingCompleted = 0;
        i+=processingCompletedThisBurst; j+=processingCompletedThisBurst;
        k+=processingCompletedThisBurst; l+=processingCompletedThisBurst;

        rc = pthread_mutex_unlock(&mutex);
        checkResults("pthread_mutex_unlock()\n", rc);
    }
    /* If any local processing remains, merge it with the global*/
    /* problem so our part of the solution is accounted for         */
    if (localProcessingCompleted) {
        rc = pthread_mutex_lock(&mutex);
        checkResults("final pthread_mutex_lock()\n", rc);

        i+=localProcessingCompleted; j+=localProcessingCompleted;
        k+=localProcessingCompleted; l+=localProcessingCompleted;

        rc = pthread_mutex_unlock(&mutex);
        checkResults("final pthread_mutex_unlock()\n", rc);
    }
    printf("Thread processed about %d%% of the problem locally\n",
                (numberOfLocalProcessingBursts * 100) / LOOPCONSTANT);
    return NULL;
}

int main(int argc, char **argv)
{
    pthread_t                   threadid[THREADS];
    int                          rc=0;
    int                          loop=0;
    pthread_attr_t             pta;

    printf("Entering testcase\n");
```

```
    pthread_attr_init(&pta);
    pthread_attr_setdetachstate(&pta, PTHREAD_CREATE_JOINABLE);

    printf("Creating %d threads\n", THREADS);
    for (loop=0; loop<THREADS; ++loop) {
      rc = pthread_create(&threadid[loop], &pta, threadfunc, NULL);
      checkResults("pthread_create()\n", rc);
    }

    printf("Wait for results\n");
    for (loop=0; loop<THREADS; ++loop) {
      rc = pthread_join(threadid[loop], NULL);
      checkResults("pthread_join()\n", rc);
    }

    printf("Cleanup and show results\n");
    pthread_attr_destroy(&pta);
    pthread_mutex_destroy(&mutex);

    printf("\nUsing %d threads and LOOPCONSTANT = %d\n",
            THREADS, LOOPCONSTANT);
    printf("Values are: (should be %d)\n", THREADS * LOOPCONSTANT);
    printf("    ==>%d, %d, %d, %d\n", i, j, k, l);

    printf("Main completed\n");
    return 0;
}
```

13.2 互斥体属性设置

13.2.1 pthread_mutexattr_init 互斥体属性初始化

13.2.2 pthread_mutexattr_destroy 销毁互斥体属性

13.2.3 pthread_mutexattr_setkind_np 设置互斥体种类属性

13.2.4 pthread_mutexattr_setname_np 设置互斥体属性名字

13.2.5 pthread_mutexattr_setpshared 设置互斥体中进程属性

13.2.6 pthread_mutexattr_settype 设置互斥体类型属性

13.2.7 pthread_mutexattr_default_np 设置互斥体为默认属性

13.3 取互斥体属性 API – Mutex Attribute API

13.3.1 pthread_mutexattr_getkind_np 取互斥体种类

13.3.2 pthread_mutexattr_getname_np 取互斥体属性目标名

13.3.3 pthread_mutexattr_getpshared 从互斥体中取出进程共享的属性

13.3.4 pthread_mutexattr_gettype 取互斥体类型

14 信号量的 API

与信号量相关的 API，其实操作的都是一个信号量组(semaphore set)，或者是一套信号量，总之就是多个信号量的集合。一个信号量组里面，会有多个信号量。

14.1 semget – 带 KEY 值取信号量描述符

语法：

```
#include <sys/sem.h>
#include <sys/stat.h>

int semget(key_t key, int nsems, int semflg);
```

功能：

semget 这个函数既可以创建一个新的信号量组并返回其描述符，也可以通过 KEY 值返回一个已存在的信号量组的描述符。当满足以下任一一个条件时，该函数将会创建新的信号量组：

5、 key 参数为 IPC_PRIVATE

6、 根据 key 参数没有找到已在的信号量组， 同时 semflg 参数为 IPC_CREAT（此时生成的信号量组即带 key 参数）

该函数不会更改已存在的信号量组的状态。

当创建新信号量组时，系统将会对 semid_ds 结构体中的成员进行如下的初始化赋值：

sem_perm.cuid, sem_perm.uid 将会赋值为当前线程的用户 ID;

sem_perm.guid, sem_perm.gid 将会赋值为当前线程的 group ID;

sem_perm.mode 的低 9 位，赋值成为输入参数 semflg 的低 9 位。

sem_nsems 赋值成为输入参数 nsems

sem_ctime 赋值成为当前时间

sem_otime 赋值为 zero

参数：

KEY

输入参数，根据此 KEY 查找，或生成信号量组。

这个参数为 IPC_PRIVATE(0x00000000)时，此时会生成一个信号量组。

用户可以自行指定这个参数，或由函数 ftok（）来产生

nsems

输入参数，应该是仅在生成信号量时使用。表明该信号量组中，信号量的数量。

当信号量生成之后，这个值不能再更改。如果访问的是一个已存在的信号量，这个值可以赋为 zero..

Semflg

输入参数，操作权限的标识。可以赋值为 zero,或是以下的值：

S_IRUSR

允许该信号量的拥有者从中读取数据

S_IWUSR
允许该信号量的拥有者向其中写数据
S_IRGRP
允许该信号量的用户组从中读取数据
S_IWGRP
允许该信号量的用户组向其中写数据
S_IROTH
允许其它用户读取该组信号量
S_IWOTH
允许其它用户向该信号量中写
IPC_CREAT
如果相应 KEY 值的信号量不存在，那么建立新的信号量
IPC_EXCL
如果设置了 IPC_CREAT，而信号量已存在时，返回错误

返回：

-1 函数发生错误
其它值 该信号量的唯一标识符

例子：

如下面这个不完整的例子中，定义了一个整型变量 semaphoreID，做为信号量的描述符，要赋初始值为-1，即与函数的错误返回相同，以便错误判断处理。

这里就创建了一个信号量组，该组中只有一个信号量。

```
#include <sys/ipc.h>
#include <sys/sem.h>
int semaphoreID = -1 ;
semaphoreId = semget(IPC_PRIVATE, 1, S_IRUSR|S_IWUSR);
```

14.2 semop 对信号量组进行操作

语法：

```
#include <sys/sem.h>
 int semop(int semid,
           struct sembuf *sops,
           size_t nsops);
```

功能：

对已存在的信号量组执行操作。

根据输入的参数，该命令可以对一组信号量中的多个信号量执行操作，也可以对同一信号量执行多次操作。（可能后者的用法会比较少吧，虽然理论上是可行的）

参数：

semid
信号量组的描述符
sops
一个数组的指针，该数组是由一个 sembuf 结构的结构体组成。这个结构体用标明

用来具体的操作内容项。

sembuf 结构体的定义如下：

```
struct sembuf {                    /* sops 即是 semaphore operation structure 的简称*/
    unsigned short sem_num;        /* 信号量组中，要操作的信号量的编号（位置） */
    short sem_op;                  /* 操作数值 */
    short sem_flg;                 /* 操作标志 SEM_UNDO and IPC_NOWAIT */
}
```

该数组中，一条记录（即一个结构体），就标识了一次操作。如果有多条记录（多个结构体），那么可以执行多次操作。

结构体中的 sem_num 变量标识的是信号量组中，指定的信号量的编号，由 0 开始。即该组信号量中，第一个信号量的编号为 0，第二个信号量为 1。

Nsops

上述数组中的记录个数（即结构体的个数），也就是要执行的操作次数。

返回：

0 操作成功

非 0 操作失败

注意事项：

系统根据入口参数中的 sops，逐笔对信号量执行操作直至完毕。当执行 sem_op 操作时，其它的线程都不能再操作这个信号量组，直到当前线程结束操作，或被挂起。

如果结构体中的 sem_op 参数是正数，函数将会增加指定的信号量的数值，然后唤起相应数量的？重新运行条件为信号量增加而的线程，这也就相当于通过信号量释放资源。

如果结构体中的 sem_op 参数是负数，函数将会减去指定的信号量的数值。

当结果为负数的时候，线程将会挂起，直到该信号量增加；（即当前线程重新运行条件为指定的信号量增加）

如果结果是正数，就仅仅只是减去信号量的数值而已；

如果结果是 0，将会唤起相应数量的？重新运行条件为信号量数值等于 0 的线程。

如果结构体中的 sem_op 的参数为 0， 就表示挂起当前线程，直到指定的信号量数值为 0.

如果 sem_flg 参数设置为 IPC_NOWAIT，那么当前线程不能运行时，将不会被挂起，而是直接返回 EAGAIN 的错误码。

如果 sem_flg 参数设置为 SEM_UNDO，那么当线程结束时，对指定信号量的操作将会回滚，也就是通过信号量来控制资源的回收与申请。

如果 sem_flg 参数设置为 0，就表示只进行标准处理，没有其它特殊操作。

当使用 sem_op 操作挂起线程的时候，这个线程可以被其它的线程中断。

例子：

在下面这个例子里，先生成了一个信号量组，这个组里只有一个信号量。然后对这个信号量进行操作。

LockOperation 就是一个 sembuf 结构的变量，

第一位 0.，表示操作信号量组中的第一个信号量；

第二位 -1，表示对信号量进行减操作；
第三位 0， 表示没有特殊处理

在执行 semop 函数时，
首位参数 semaporeId，表示信号量组的描述符；
第二位参数传递的是 lockOperation 地址
第三位参数 1，表示第二位参数传递的内容中，只含一次操作的内容。

```
#include <sys/ipc.h>
#include <sys/sem.h>
int rc;
int semaphoreID    =    -1    ;
struct sembuf lockOperation = { 0, -1, 0};
semaphoreId = semget(IPC_PRIVATE, 1, S_IRUSR|S_IWUSR);
rc = semop(semaphoreId, &lockOperation, 1);
```

14.3 semctl -- 对信号量进行控制操作

语法：

```
#include <sys/sem.h>
  int semctl(int semid, int semnum, int cmd,  可选参数);
```

功能：

这个函数允许调用者控制指定的信号量。
调用者通过 cmd 参数，来表达需要进行控制的内容：

IPC_RMID（0x00000000）

从系统中移除信号量组描述符，销毁信号量组。所有执行 sem_op 操作后，基于这个信号量组而挂起的线程，将会返回一个-1， 同时错误码为 EIDRM

IPC_SET （0x00000001）

通过第 4 个可选参数，设置 semid_ds 数据结构中的 sem_perm.uid，sem_perm.gid，sem_perm.mode 的值。这里，第 4 个参数此时是一个指向 semid_ds 类型结构体的指针。

IPC_STAT （0x00000002）

通过第 4 个可选参数，保存 semid_ds 数据结构中的当前值。第 4 个参数此时是一个指向 semid_ds 类型结构体的指针。

GETNCNT （0x00000003）

返回唤起条件为“指定的信号量数值增加”的线程数量。这个数值对应 semaphore_t 结构中的 semncnt。

GETPID (0x00000004)

返回最后一个操作指定信号量的线程所属的进程 ID。这个数值对应 semaphore_t 结构中的 sempid.

GETVAL（0x00000005)

返回指定的信号量的当前数值。这个数值对应 semaphore_t 结构中的 semval.

GETALL（0x00000006）

通过第四个可选参数，返回这个信号量组中，所有信号量当前的值。这个参数此时是一个数组指针，数组中元素类型为 unsigned short.

GETZCNT（0x00000007）

返回唤起条件为“指定的信号量数值等于 0”的线程数量。这个数值对应 smeaphore_t 结构中的 semzcnt.

SETVAL（0x00000008）

将指定的信号量的数值赋值为第 4 个可选参数，此时该参数类型需要为 int.同时清除与信号量有关联的每个线程的信号量调整值（Set the value of semaphore semnum to the integer value of type int specified in the fourth parameter and clear the associated per-thread semaphore adjustment value.）

SETALL(0x00000009)

通过第 4 个可选参数，将信号量组中的每一个信号量的值都进行相应更改。此时第 4 个参数是一个数组指针，数组中元素类型为 unsigned short.同时，与每个信号量有关联的线程，都会清除其信号量调整值。（In addition, the associated per-thread semaphore-adjustment value is cleared for each semaphore）

参数：

semid

输入参数， 正整数，信号量组描述符。

Semnum

输入参数，非负整数。标识指定信号量在信号量组中的编号。编号从 0 开始，即该组信号量中，第一个信号量的编号为 0，第二个信号量的编号为 1.

Cmd

控制符，见上文中的描述。

可选参数

根据具体情况而定，有时表示输入，有时表示输出。有时是一个整型变量，有时是一个数组指针，详情见上文中的描述。

返回：

当返回成功时，视 cmd 参数的不同，会有不同的返回值

GETVAL	返回信号量的数值
GETPID	返回进程号
GETNCNT	返回符合条件的线程数量
GETZCNT	返回符合条件的线程数量
其它操作	返回 0

当函数失败时，返回-1

例子：

下面这个简单的例子中，接上面 semop 操作的内容，对信号量组进行了 SETVAL 的控制操作，将这个信号量组中的第一个信号量的数值设置为 1.

```
#include <sys/ipc.h>
#include <sys/sem.h>
int rc;
int semaphoreID    =    -1    ;
struct sembuf lockOperation = { 0, -1, 0};
semaphoreId = semget(IPC_PRIVATE, 1, S_IRUSR|S_IWUSR);
rc = semop(semaphoreId, &lockOperation, 1);
rc = semctl(semaphoreId, 0, SETVAL, (int)1);
```

15 其它

15.1 spawn

语法：

```
#include <spawn.h>
pid_t spawn( const char                    *path,
             const int                     fd_count,
             const int                     fd_map[],
             const struct inheritance      *inherit,
             char * const                  argv[],
             char * const                  envp[]);
```

功能：

参数：

path

输入参数，可执行文件名。

Fd_count

输入参数，子进程可以继承的文件描述符

fd_map[]

输入参数，子进程接收到当前进程传递给它的文件描述符数组

inherit

输入参数，一个 inheritance 类型的结构的地址。

返回：

例子：

下面这个例子演示了通过 spawn 函数，创建一个进程，新的进程继承了当前进程的 socket 描述符。

```
/*****************************************************************************/
/* Application creates an child process using spawn().                       */
/*****************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <spawn.h>

#define SERVER_PORT    12345

main (int argc, char *argv[])
{
    int        i, num, pid, rc, on = 1;
    int        listen_sd, accept_sd;
    int        spawn_fdmap[1];
    char    *spawn_argv[1];
    char    *spawn_envp[1];
    struct inheritance      inherit;
    struct sockaddr_in      addr;

    /************************************************/
    /* If an argument was specified, use it to      */
    /* control the number of incoming connections   */
    /************************************************/
    if (argc >= 2)
        num = atoi(argv[1]);
    else
        num = 1;

    /************************************************/
    /* Create an AF_INET stream socket to receive   */
    /* incoming connections on                      */
    /************************************************/
    listen_sd = socket(AF_INET, SOCK_STREAM, 0);
    if (listen_sd < 0)
    {
        perror("socket() failed");
        exit(-1);
    }
```

```
/*************************************************/
/* Allow socket descriptor to be reuseable       */
/*************************************************/
rc = setsockopt(listen_sd,
                SOL_SOCKET,  SO_REUSEADDR,
                (char *)&on, sizeof(on));
if (rc < 0)
{
    perror("setsockopt() failed");
    close(listen_sd);
    exit(-1);
}

/*************************************************/
/* Bind the socket                               */
/*************************************************/
memset(&addr, 0, sizeof(addr));
addr.sin_family      = AF_INET;
addr.sin_port        = htons(SERVER_PORT);
addr.sin_addr.s_addr = htonl(INADDR_ANY);
rc = bind(listen_sd,
          (struct sockaddr *)&addr, sizeof(addr));
if (rc < 0)
{
    perror("bind() failed");
    close(listen_sd);
    exit(-1);
}

/*************************************************/
/* Set the listen back log                       */
/*************************************************/
rc = listen(listen_sd, 5);
if (rc < 0)
{
    perror("listen() failed");
    close(listen_sd);
    exit(-1);
}

/*************************************************/
/* Inform the user that the server is ready      */
/*************************************************/
printf("The server is ready\n");
```

```
/*********************************************/
/* Go through the loop once for each connection    */
/*********************************************/
for (i=0; i < num; i++)
{
    /*********************************************/
    /* Wait for an incoming connection                */
    /*********************************************/
    printf("Interation: %d\n", i+1);
    printf("    waiting on accept()\n");
    accept_sd = accept(listen_sd, NULL, NULL);
    if (accept_sd < 0)
    {
        perror("accept() failed");
        close(listen_sd);
        exit(-1);
    }
    printf("    accept completed successfully\n");

    /*********************************************/
    /* Initialize the spawn parameters                */
    /*                                                */

    /* The socket descriptor for the new              */
    /* connection is mapped over to descriptor 0    */
    /* in the child program.                          */
    /*********************************************/
    memset(&inherit, 0, sizeof(inherit));
    spawn_argv[0]    = NULL;
    spawn_envp[0]    = NULL;
    spawn_fdmap[0] = accept_sd;

    /*********************************************/
    /* Create the worker job                          */
    /*********************************************/
    printf("    creating worker job\n");
    pid = spawn("/QSYS.LIB/QGPL.LIB/WRKR1.PGM",
                      1, spawn_fdmap, &inherit,
                      spawn_argv, spawn_envp);
    if (pid < 0)
    {
        perror("spawn() failed");
        close(listen_sd);
        close(accept_sd);
```

```
                exit(-1);
            }
            printf("    spawn completed successfully\n");

            /*************************************************/
            /* Close down the incoming connection since      */
            /* it has been given to a worker to handle       */
            /*************************************************/
            close(accept_sd);
        }

        /*************************************************/
        /* Close down the listen socket                  */
        /*************************************************/
        close(listen_sd);
    }
```

语法：

功能：

参数：

返回：

例子：