POSIX 线程详解

线程是有趣的

了解如何正确运用线程是每一个优秀程序员必备的素质。线程类似于进程。如同进程，线程由内核按时间分片进行管理。在单处理器系统中，内核使用时间分片来模拟线程的并发执行，这种方式和进程的相同。而在多处理器系统中，如同多个进程，线程实际上一样可以并发执行。

那么为什么对于大多数合作性任务，多线程比多个独立的进程更优越呢？这是因为，线程共享相同的内存空间。不同的线程可以存取内存中的同一个变量。所以，程序中的所有线程都可以读或写声明过的全局变量。如果曾用 fork() 编写过重要代码，就会认识到这个工具的重要性。为什么呢？虽然 fork() 允许创建多个进程，但它还会带来以下通信问题: 如何让多个进程相互通信，这里每个进程都有各自独立的内存空间。对这个问题没有一个简单的答案。虽然有许多不同种类的本地 IPC (进程间通信），但它们都遇到两个重要障碍：

- 强加了某种形式的额外内核开销，从而降低性能。
- 对于大多数情形，IPC 不是对于代码的“自然”扩展。通常极大地增加了程序的复杂性。

双重坏事: 开销和复杂性都非好事。如果曾经为了支持 IPC 而对程序大动干戈过，那么您就会真正欣赏线程提供的简单共享内存机制。由于所有的线程都驻留在同一内存空间，POSIX 线程无需进行开销大而复杂的长距离调用。只要利用简单的同步机制，程序中所有的线程都可以读取和修改已有的数据结构。而无需将数据经由文件描述符转储或挤入紧窄的共享内存空间。仅此一个原因，就足以让您考虑应该采用单进程/多线程模式而非多进程/单线程模式。

线程是快捷的

不仅如此。线程同样还是非常快捷的。与标准 fork() 相比，线程带来的开销很小。内核无需单独复制进程的内存空间或文件描述符等等。这就节省了大量的 CPU 时间，使得线程创建比新进程创建快上十到一百倍。因为这一点，可以大量使用线程而无需太过于担心带来的 CPU 或内存不足。使用 fork() 时导致的大量 CPU 占用也不复存在。这表示只要在程序中有意义，通常就可以创建线程。

当然，和进程一样，线程将利用多 CPU。如果软件是针对多处理器系统设计的，这就真的是一大特性（如果软件是开放源码，则最终可能在不少平台上运行）。特定类型线程程序（尤其是 CPU 密集型程序）的性能将随系统中处理器的数目几乎线性地提高。如果正在编写 CPU 非常密集型的程序，则绝对想设法在代码中使用多线程。一旦掌握了线程编码，无需使用繁琐的 IPC 和其它复杂的通信机制，就能够以全新和创造性的方法解决编码难题。所有这些特性配合在一起使得多线程编程更有趣、快速和灵活。

线程是可移植的

如果熟悉 Linux 编程，就有可能知道 __clone() 系统调用。__clone() 类似于 fork()，同时也有许多线程的特性。例如，使用 __clone()，新的子进程可以有选择地共享父进程的执行环境（内存空间，文件描述符等）。这是好的一面。但 __clone() 也有不足之处。正如__clone() 在线帮助指出：

> “__clone 调用是特定于 Linux 平台的，不适用于实现可移植的程序。欲编写线程化应用程序（多线程控制同一内存空间），最好使用实现 POSIX 1003.1c 线程 API 的库，例如 Linux-Threads 库。参阅 pthread_create(3thr)。”

虽然 __clone() 有线程的许多特性，但它是不可移植的。当然这并不意味着代码中不能使用它。但在软件中考虑使用 __clone() 时应当权衡这一事实。值得庆幸的是，正如 __clone() 在线帮助指出，有一种更好的替代方案：POSIX 线程。如果想编写 **可移植的** 多线程代码，代码可运行于 Solaris、FreeBSD、Linux 和其它平台，POSIX 线程是一种当然之选。

第一个线程

下面是一个 POSIX 线程的简单示例程序：

thread1.c

```c
#include <pthread.h>
#include <stdlib.h>
#include <unistd.h>
 void *thread_function(void *arg) {
  int i;
  for ( i=0; i<20; i++) {
    printf("Thread says hi!\n");
    sleep(1);
  }
  return NULL;
}
int main(void) {
  pthread_t mythread;

  if ( pthread_create( &mythread, NULL, thread_function, NULL) ) {
    printf("error creating thread.");
    abort();
  }
  if ( pthread_join ( mythread, NULL ) ) {
    printf("error joining thread.");
    abort();
  }
  exit(0);
}
```

要编译这个程序，只需先将程序存为 thread1.c，然后输入：

```
$ gcc thread1.c -o thread1 -lpthread
```

运行则输入：

```
$ ./thread1
```

理解 thread1.c

thread1.c 是一个非常简单的线程程序。虽然它没有实现什么有用的功能，但可以帮助理解线程的运行机制。下面，我们一步一步地了解这个程序是干什么的。main() 中声明了变量 mythread，类型是 pthread_t。pthread_t 类型在 pthread.h 中定义，通常称为“线程 id”（缩写为 "tid"）。可以认为它是一种线程句柄。

mythread 声明后（记住 mythread 只是一个 "tid"，或是将要创建的线程的句柄），调用 pthread_create 函数创建一个真实活动的线程。不要因为 pthread_create() 在 "if" 语句内而受其迷惑。由于 pthread_create() 执行成功时返回零而失败时则返回非零值，将 pthread_create() 函数调用放在 if() 语句中只是为了方便地检测失败的调用。让我们查看一下 pthread_create 参数。第一个参数 &mythread 是指向 mythread 的指针。第二个参数当前为 NULL，可用来定义线程的某些属性。由于缺省的线程属性是适用的，只需将该参数设为 NULL。

第三个参数是新线程启动时调用的函数名。本例中，函数名为 thread_function()。当 thread_function() 返回时，新线程将终止。本例中，线程函数没有实现大的功能。它仅将 "Thread says hi!" 输出 20 次然后退出。注意 thread_function() 接受 void * 作为参数，同时返回值的类型也是 void *。这表明可以用 void * 向新线程传递任意类型的数据，新线程完成时也可返回任意类型的数据。那如何向线程传递一个任意参数？很简单。只要利用 pthread_create() 中的第四个参数。本例中，因为没有必要将任何数据传给微不足道的 thread_function()，所以将第四个参数设为 NULL。

您也许已推测到，在 pthread_create() 成功返回之后，程序将包含两个线程。等一等， **两个** 线程？我们不是只创建了一个线程吗？不错，我们只创建了一个进程。但是主程序同样也是一个线程。可以这样理解：如果编写的程序根本没有使用 POSIX 线程，则该程序是单线程的（这个单线程称为“主”线程）。创建一个新线程之后程序总共就有两个线程了。

我想此时您至少有两个重要问题。第一个问题，新线程创建之后主线程如何运行。答案，主线程按顺序继续执行下一行程序（本例中执行 "if (pthread_join(...))"）。第二个问题，新线程结束时如何处理。答案，新线程先停止，然后作为其清理过程的一部分，等待与另一个线程合并或“连接”。

现在，来看一下 pthread_join()。正如 pthread_create() 将一个线程拆分为两个， pthread_join() 将两个线程合并为一个线程。pthread_join() 的第一个参数是 tid mythread。第二个参数是指向 void 指针的指针。如果 void 指针不为 NULL，pthread_join 将线程的 void * 返回值放置在指定的位置上。由于我们不必理会 thread_function() 的返回值，所以将其设为 NULL.

您会注意到 thread_function() 花了 20 秒才完成。在 thread_function() 结束很久之前，主线程就已经调用了 pthread_join()。如果发生这种情况，主线程将中断（转向睡眠）然后等待 thread_function() 完成。当 thread_function() 完成后, pthread_join() 将返回。这时程序又只有一个主线程。当程序退出时，所有新线程已经使

用 pthread_join() 合并了。这就是应该如何处理在程序中创建的每个新线程的过程。如果没有合并一个新线程，则它仍然对系统的最大线程数限制不利。这意味着如果未对线程做正确的清理，最终会导致 pthread_create() 调用失败。

无父，无子

如果使用过 fork() 系统调用，可能熟悉父进程和子进程的概念。当用 fork() 创建另一个新进程时，新进程是子进程，原始进程是父进程。这创建了可能非常有用的层次关系，尤其是等待子进程终止时。例如，waitpid() 函数让当前进程等待所有子进程终止。waitpid() 用来在父进程中实现简单的清理过程。

而 POSIX 线程就更有意思。您可能已经注意到我一直有意避免使用“父线程”和“子线程”的说法。这是因为 POSIX 线程中不存在这种层次关系。虽然主线程可以创建一个新线程，新线程可以创建另一个新线程，POSIX 线程标准将它们视为等同的层次。所以等待子线程退出的概念在这里没有意义。POSIX 线程标准不记录任何“家族”信息。缺少家族信息有一个主要含意：如果要等待一个线程终止，就必须将线程的 tid 传递给 pthread_join()。线程库无法为您断定 tid。

对大多数开发者来说这不是个好消息，因为这会使有多个线程的程序复杂化。不过不要为此担忧。POSIX 线程标准提供了有效地管理多个线程所需要的所有工具。实际上，没有父/子关系这一事实却为在程序中使用线程开辟了更创造性的方法。例如，如果有一个线程称为线程 1，线程 1 创建了称为线程 2 的线程，则线程 1 自己没有必要调用 pthread_join() 来合并线程 2，程序中其它任一线程都可以做到。当编写大量使用线程的代码时，这就可能允许发生有趣的事情。例如，可以创建一个包含所有已停止线程的全局“死线程列表”，然后让一个专门的清理线程专等停止的线程加到列表中。这个清理线程调用 pthread_join() 将刚停止的线程与自己合并。现在，仅用一个线程就巧妙和有效地处理了全部清理。

同步漫游

现在我们来看一些代码，这些代码做了一些意想不到的事情。thread2.c 的代码如下：

thread2.c

```
#include <pthread.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
int myglobal;
 void *thread_function(void *arg) {
  int i,j;
  for ( i=0; i<20; i++) {
    j=myglobal;
    j=j+1;
    printf(".");
    fflush(stdout);
    sleep(1);
    myglobal=j;
  }
  return NULL;
}
int main(void) {
  pthread_t mythread;
  int i;
  if ( pthread_create( &mythread, NULL, thread_function, NULL) ) {
    printf("error creating thread.");
    abort();
  }
  for ( i=0; i<20; i++) {
    myglobal=myglobal+1;
    printf("o");
    fflush(stdout);
    sleep(1);
  }
  if ( pthread_join ( mythread, NULL ) ) {
    printf("error joining thread.");
    abort();
  }
  printf("\nmyglobal equals %d\n",myglobal);
  exit(0);
}
```

理解 thread2.c

如同第一个程序，这个程序创建一个新线程。主线程和新线程都将全局变量 myglobal 加一 20 次。但是程序本身产生了某些意想不到的结果。编译代码请输入：

```
1 $ gcc thread2.c -o thread2 -lpthread
```

运行请输入：

```
1 $ ./thread2
```

输出：

```
1 $ ./thread2
2 ..o.o.o.o.oo.o.o.o.o.o.o.o.o.o..o.o.o.o.o
3 myglobal equals 21
```

非常意外吧！因为 myglobal 从零开始，主线程和新线程各自对其进行了 20 次加一, 程序结束时 myglobal 值应当等于 40。由于 myglobal 输出结果为 21，这其中肯定有问题。但是究竟是什么呢？

放弃吗？好，让我来解释是怎么一回事。首先查看函数 thread_function()。注意如何将 myglobal 复制到局部变量 "j" 了吗? 接着将 j 加一, 再睡眠一秒，然后到这时才将新的 j 值复制到 myglobal？这就是关键所在。设想一下，如果主线程就在新线程将 myglobal 值复制给 j 后 立即将 myglobal 加一，会发生什么？当 thread_function() 将 j 的值写回 myglobal 时，就覆盖了主线程所做的修改。

当编写线程程序时，应避免产生这种无用的副作用，否则只会浪费时间（当然，除了编写关于 POSIX 线程的文章时有用）。那么，如何才能排除这种问题呢？

由于是将 myglobal 复制给 j 并且等了一秒之后才写回时产生问题，可以尝试避免使用临时局部变量并直接将 myglobal 加一。虽然这种解决方案对这个特定例子适用，但它还是不正确。如果我们对 myglobal 进行相对复杂的数学运算，而不是简单的加一，这种方法就会失效。但是为什么呢？

要理解这个问题，必须记住线程是并发运行的。即使在单处理器系统上运行（内核利用时间分片模拟多任务）也是可以的，从程序员的角度，想像两个线程是同时执行的。thread2.c 出现问题是因为 thread_function() 依赖以下论据：在 myglobal 加一之前的大约一秒钟期间不会修改 myglobal。需要有些途径让一个线程在对 myglobal 做更改时通知其它线程“不要靠近”。我将在下一篇文章中讲解如何做到这一点。到时候见