# Linux下多线程技术分析及应用

金惠芳 （上海华东政法学院 200042）
陶利民 张基温 （无锡江南大学信息工程学院 214036）

**摘要**：本文介绍了进程、线程的基本概念及多线程的实现方式；分析了Linux中多线程的实现技术；最后，介绍了Linuxpthreads库中的关键函数，并以经典的读者-写者问题的实现来阐述多线程编程的核心核技术。

**关键词**：进程 线程 多线程 互斥 同步

## 1 多线程的实现

多线程是指操作系统支持一个进程中执行多个线程的能力，多线程的实现方式分为用户级多线程（user-level multithread）、内核级多线程(kernel-level multithread)和混合多线程三种方式。用户级多线程方式没有操作系统内核的支持，完全在用户级提供一个库程序来实现多线程管理，这些库提供了创建、同步、调度与管理线程的所有功能，无需操作系统特别支持，多线程库实现的多线程分时地在进程中运行，由多线程库线程调度器负责线程的调度与切换，在内核看来，该进程运行着用户的程序而完全不知何时在用户多线程中切换。内核线程是由操作系统支持实现的线程，操作系统维护内核级线程的各种管理表格、负责线程在处理机上的调度和切换，用户层上无需内核级线程的管理代码，操作系统提供了一系列系统调用界面让用户程序请求操作系统作线程创建、结束等操作，用户级多线程方式的缺陷是不能做到进程内线程在处理机上真正并行运行，容易因进程等待而引起阻塞；而内核级线程可以支持进程内多线程在多处理机上真正并发执行，但内核级线程比用户级线程切换的开销要大一些，这是因为内核级线程的管理都由内核程序进行，需要用户态到核心态的切换。混合级多线程方式（即用户级和内核级结合）则是综合两者的优点，这样既利于用户编写并行程序又能够最大限度的发挥多处理机的并行性。现代UNIX操作系统一般都实现了混合方式的多线程，如Solarisz.x、UNIX SVR4.2MP等。

线程库提供了多线程的应用程序接口（API），IEEE制定了多线程的标准POSIX1003.1c，它不仅支持用户级和内核级多线程，而且支持混合多线程。POSIX1003.1c定义了可移植的线程接口标准，它已经获得大多数UNIX系统的承认和支持。因此，只要是符合POSIX1003.1c标准的多线程应用程序，就能无缝移植到大部分UNIX平台。Linux多线程符合POSIX多线程接口标准，称为pthreads。

## 2 Linux下的多线程技术

Linux作为一个多用户多任务操作系统，支持多道程序设计、分时处理和"软"实时处理。但Linux的内核本身并不涉及多线程处理，Linux中也没有为线程单独定义数据结构。因此，Linux中进程和线程并没有区别。Linux中多线程纯粹是以进程为处理器调度单位，多进程处理就是多线程处理。不过在Linux中提供了LinuxThreads库，它是一个符合POSIX1003.1c接口标准的内核级方式多线程函数库。其实，LinuxThreads中的每个线程仍然是单独的UNIX进程，但该进程却和线程一样只占用较少的系统资源，我们也将此类进程称为轻权进程LWP。

Linux中的进程由一个task_struct数据结构表示。Linux中维护了一个task表，它存储了当前定义的每个task_struct的指针，task_struct数据结构包含以下几类信息：进程状态、进程调度信息、标识号、进程间通信、进程链接信息、时间和计时器、文件系统信息、虚存信息和处理器专用上下文环境。Linux进程有执行、就绪、挂起、停止、僵死五种状态。Linux每个进程拥有自己的虚拟地址空间，它们之间互不干扰，通过内核提供的进程间通信机制（IPC）相互作用。在传统的UNIX中用fork系统调用创建子进程，如果创建成功，则父进程和子进程处于不同的位置空间，子进程创建过程实质上是产生一个父进程的副本，子进程的proc结构（即PCB中的一部分）相对于父进程的proc结构的个别项（如增加了父进程名、修订了该子进程的创建时间等等）作了变动，这种方式创建子进程不但增加了CPU的开销，而且消耗了大量的内存空间，导致加大了系统的负荷。而在LinuxThreads中进程的创建不是使用传统的方法（即通过系统调用fork），而是通过克隆（clone，在文件arch/i386/kernel/process.c中，函数原型：asmlinkage int sys_clone(struct pt_regs regs)）系统调用来复制当前进程的属性，从而创建一个新进程。克隆出来的"线程"共享父进程的资源（如文件、信号处理程序和虚存），这样就节省了内存空间、减少了CPU的开销。为了提高创建进程的效率，Linux采用了"写时复制（copy on write）"技术，只有在两个进程中任意一个向虚拟内存中写入数据时复制相应的虚拟内存，而没有写入的任何内存页均可以在两个进程间共享采用copy on wrie技术复制多个进程，也不会明显加大系统负荷，LinuxThreads库这种clone系统调用实现了效率较高的内核级多线程支持，从而大大增进了多线程编程的效率。

## 3 LinuxThread中多线程编程的关键库函数

下面介绍的LinuxThread中的库函数是用于多线程编程的一些关键函数。当前大多数Linux的发行版本中pthread库是libpthread-0.9.so，头文件是<pthread.h>。这些函数提供线程的创建、结束、同步和互斥等。

### 3.1 线程的创建和终止

int pthread_create(pthread_t * pthread,const pthread_attr_t *attr,void *(*start_routine)(void *), void *arg);

每个进程创建时，系统同时创建一个核心主线程。当要创建新的线程去完成并行任务，可调用pthread_create，新线程创建后执行由start_routine指定的例程，此例程是一个用户自定义函数。调用者可通过arg给start_routine过程传递参数，参数attr是程序希望创建的线程的属性。当成功创建一个新的线程时，系统会为该线程分配一个tid（pthread ID），并将该值通过参数pthread指针返回给调用者。

void pthread_exit(void *value_ptr);

一个线程可以隐式的退出（即当例程返回时，线程则自动结束运行），也可以是显示调用pthread_exit函数退出，参数value_ptr是一个指向返回状态值的指针。

### 3.2 线程的控制

pthread_t pthread_self（void）;

每个线程都有自己的线程ID，以便于在一个进程内区分。线程ID在pthread_create调用时，会返回给创建线程的调用者。一个线程在创建后也可以使用pthread_self调用来获得自身的线程ID。

int pthread_join(pthread_t thread,void **status);

该函数的作用是等待一个线程终止。调用pthread_join的线程将被挂起，直到线程ID为参数thread指定的线程终止时为止。

int pthread_detach(pthread_t thread);

函数pthread_detach用来将一个线程脱离。参数thread是要设置成脱离状态的线程ID。具有脱离属性的线程在终止时，会立即释放该线程占有的所有资源。

### 3.3 线程之间的互斥

互斥操作就是对某段代码或某个变量修改时只能有一个线程在执行，其他线程不能同时进入该段代码或同时修改变量。Linux中可以通过pthread_mutex_t来定义互斥体机制来完成多线程的互斥操作。

int pthread_mutex_init(pthread_mutex_t *mutex,const pthread_mutexattr_t *attr);

该函数初始化一个互斥体变量mutex。函数int pthread_mutex_init按参数attr指定的属性创建一个新的互斥体变量mutex，如果参数为NULL，则互斥体变量mutex使用默认的属性。

int pthread_mutex_lock(pthread_mutex_t *mutex);

该函数用来锁住互斥体变量。如果参数mutex所指定的互斥体已经被锁住了，那么发出调用的线程将被阻塞直到其他线程对mutex解锁为止。

int pthread_mutex_unlock(pthread_mutex_t *mutex);

该函数用来对参数mutex指定的互斥体解锁。

int pthread_mutex_trylock(pthread_mutex_t *mutex);

该函数用来锁住参数mutex所指定的互斥体，如果指定的互斥体已经被上锁，该调用不会阻塞等待互斥体的解锁，而会迅速返回。

int pthread_mutex_destroy(pthread_mutex_t *mutex);

该函数用来释放对互斥体mutex分配的资源。

### 3.4 线程之间的同步

同步（synchronize）就是若干个线程等待某个事件的发生，当等待的事件发生时，一起开始继续执行，否则线程挂起。Linux下可以通过pthread_cond_t定义条件变量来实现多线程的同步。

int pthread_cont_init(pthread_cond_t *cond, const pthread_condattr_t *attr);

该函数按参数attr指定的属性创建一个新的条件变量cond。

int pthread_cont_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);

该函数的作用是等待一个事件（条件变量）的发生，发出调用的当前线程自动阻塞，直到相应的条件变量被置上，等待状态下的线程不占CPU时间。

int pthread_cont_broadcast（pthread_cond_t *cond）;

该函数用来对所有等待一个条件变量cond的线程解除阻塞。

int pthread_cont_signal(pthread_cond_t *cond);

该函数仅解除一个等待参数cond指定的条件变量的线程的阻塞状态。

int pthread_cont_destroy(pthread_cond_t *cond);

该函数释放为条件变量cond所分配的资源。

## 4 多线程应用实例

我们可以用多线程来实现经典的读者-写者问题。读者-写者问题（readers-writers problem）是指多个进程对一个共享资源进行读写操作的问题。假设"读者"进程可对共享资源进行读操作，"写者"进程可对共享资源进行写操作；任一时刻写者只允许一个，而"读者"则允许多个。即对共享资源的读写操作的限制关系包括："读-写"互斥、"写-写"互斥和"读读"允许。因此，我们可以为写者之间，写者与第一个读者之间要求共享资源进行互斥访问，而后续者不需要互斥访问。为此，可设置两个互斥体wmutex(允许写)、rmutex（允许读）和一个公共变量readcount（表示正在读的进程数）。下面是完整的程序段及注释，该例子综合运用了前面所述的多线程编程技术：

```
/* readers-writers problem*/
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <sys/types.h>
#include <sys/stat.h>
struct BUFFER{/* 缓冲区相关数据结构 */
char buffer;/* 缓冲区存放的字符信息 */
pthread_mutex_t lock;/*互斥体lock用于对缓冲区进行互斥操作 */
```

```
int buffer_has_item;/* 缓冲区有无信息的标志 buffer_has_item 等于 1 表示有信息，若等 */
/* 于 0 表示无信息 */
pthread_cond_t empty;/* 缓冲区为空的条件变量*/
pthread_cond_t notempty; /* 缓冲区非空的条件变量 */
};
/* 初始化缓冲区数据结构 */
void init(struct BUFFER *a){
pthread_mutex_init(&a->lock,NULL);
pthread_cond_init(&a->empty,NULL);
pthread_cond_init(&a->notempty,NULL);
a->buffer_has_item=0;
}
/* 将信息放入缓冲区，这里存入一个字符 */
void putdata(struct BUFFER *a,char ch){
pthread_mutex_lock(&a->lock);
/* 等待缓冲区空 */
if(a-> buffer_has_item= =1)
    pthread_cond_wait(&a->empty,&a->lock);
/* 存数据，并修改标志 buffer_has_item */
a->buffer=ch;
    buffer_has_item=1;
/* 设置缓冲区非空的条件变量 */
pthread_cond_signal(&a->notempty);
pthread_mutex_unlock(&a->lock);
return NULL;
}
/* 从缓冲区取出字符 */
char getdata(struct BUFFER *a){
char ch;
pthread_mutex_lock(&a->lock);
/* 等待缓冲区非空 */
if(a->buffer_has_item= =0)
    pthread_cond_wait(&a->notempty,&a->lock);
/* 读字符，修改标志 buffer_has_item */
ch=a->buffer;
buffer_has_item=0;
/* 设置缓冲区已空的条件变量 */
pthread_mutex_unlock(&a->lock);
return ch;
}
/* 写者进程 */
void *writer(void *data){
while(1){
    pthread_mutex_lock(&wmutex);
    putdata(&bf,'A');/* 往缓冲区写字符 */
    pthread_mutex_unlock(&wmutex);
    }
return NULL;
}
/* 读者进程 */
void *reader(void *data){
char c;
while(1){
    pthread_mutex_lock(&rmutex);
    if(readcount= =0)
      pthread_mutex_lock(&wmutex);
    readcount++;
    pthread_mutex_unlock(&rmutex);
    c=getdata(&bf);/* 读操作 */
    printf("read-->%c",c);
    pthread_mutex_lock(&rmutex);
    readcount--;
    if(readcount= =0)
      pthread_mutex_unlock(&wmutex);
    pthread_mutex_unlock(&rmutex);
    }
return NULL;
}
pthread_mutex_t rmutex;/* 读进程互斥体 */
pthread_mutex_t wmutex;/* 写进程互斥体 */
int readcount=0;/*readcount 表示正在读的进程数目 */
struct BUFFER bf;
/* 初始化互斥体变量 */
pthread_mutex_init(&rmutex,NULL);
pthread_mutex_init(&wmutex,NULL);
/* 主函数，进行读写测试 */
int main(void){
pthread_t th_w;/*th_w 表示写进程的线程 ID*/
pthread_t th_r; /*th_r 表示读进程的线程 ID*/
void *status;/* status 指针指向线程返回值的存储位置 */
init(&bf); /* 初始化缓冲区数据结构 */
/* 创建读者和写者进程 */
pthread_create(&th_w,NULL,(void *)&writer,
NULL);
pthread_create(&th_r,NULL,(void *)&rriter,
NULL);
/* 等待两个线程结束 */
pthread_join(th_w,&status);
pthread_join(th_r,&status);
return 1;
}
```

## 5 结束语

Linux 中基于 POSIX 标准的 pthread 函数较好地支持了多线程，可以在单线程中实现多任务；在具体编程时，用 -D _REENTRANT 编译，包含头文件 pthread.h，链接时链入 pthread 库，在作者安装的 RedHat7.0 中为 libpthread-0.9.so。在编译链接时使用如下命令：gcc -D_REENTRANT -libpthread-0.9.so filename.c。


**参考文献**

1 汤子瀛、哲凤屏、汤小舟，计算机操作系统，西安电子科技大学出版社,1999。

2 陈莉君，Linux 操作系统内核分析，人民邮电出版社,2000。

3 李善平、刘文峰、李程远等，Linux 内核 2.4 版源代码分析大全，机械工业出版社,2002。