Solaris 多线程体系结构研究及多线程应用

王　晨　张毅谨　宋俊德　（北京邮电大学计算机系　100084）

摘要：本文介绍了从单线程向多线程操作系统过渡的原因，着重论述了 Solaris 双层多线程体系结构模型、多线程同步机制与函数接口，并对多线程实例进行了剖析。

关键词：多线程　Solaris　体系结构

一、使用多线程的原因

传统老式 UNIX 操作系统的进程只有一个线程，这时，线程指的就是程序中被执行的指令序列，同时还包括程序计数器（PC）和用来保留局部变量和返回地址的堆栈。而多线程 UNIX 操作系统（如 SUN Solaris2.6、SCO UnixWare7、IBM AIX4.1 以及 Linux 等）的进程可以包括一个或多个相互关联的线程，而且这些线程间独立运行。产生这种从单线程操作系统向多线程操作系统过渡的不外乎是下述原因。

一是对多处理器（CPU）硬件支持的要求。例如进行数学计算中的矩阵乘法，n 个处理器上产生 n 个线程，每一个线程负责计算一列，这种多线程的方法充分利用了系统的硬件资源，同时提高了数学运算的速度；二是对应用程序同时性（concurrency）的要求。例如：应用程序一方面要进行长时间、大规模的计算，另一方面要求计算的同时能够响应用户界面操作。利用操作系统对多线程的支持，可以很容易地实现（即一个线程负责计算，另一线程响应用户操作），达到了对程序同时性的要求。第三是从节省系统资源的角度考虑。每一个进程都有其占用的地址空间、全局变量、文件标识符、管道等系统资源以及操作系统状态表项，在多线程操作系统中，一个进程内的所有线程可以共享该进程占用的系统资源，而如果采用单线程的操作系统（线程将变为进程），就会占用多得多的系统资源，有可能出现系统资源不足，新进程无法运行的不良后果，而且用于创建和维护多进程大量的状态表项同多线程方法相比，时间（大约是线程创建时间的 5 至 30 倍）与空间都更加昂贵，系统资源花费的代价更大。

总之，使用多线程可以提高应用程序的响应速度，充分利用与节省系统资源，同时可以改善程序的结构与性能。

二、多线程体系结构

1. 双层多线程结构模型

Solaris 多线程结构模型分为两个层次（见图 1）。第一层为用户级线程接口，主要负责处理程序员书写的多线程程序；第二层为轻量级进程（LWP）接口，该层负责与多线程操作系统的内核进行通信。通过该接口提供了从用户级到系统内核级线程实现的桥梁。采用这种双层多线程模型最大的优点在于将线程的实现同操作系统的内核隔离起来，能够使系统达到更高的效率。例如，当线程读写某一文件时，由于等待文件输入/输出结束，轻量级进程处于阻塞状态，此时，这种将系统内核隔离的双层模型就允许内核进行其他事务的处理；而如果采用单层模型，即由内核直接处理用户线程，内核就将频繁处于为线程分配数据结构和线程上下文的状态，系统效率就会明显降低。由此可见，Solaris 双层多线程模型对线程的实现是起着重要作用的。

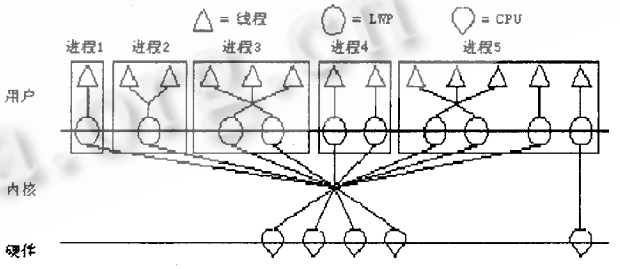

图 1　Solaris 多线程结构模型

2. 用户级线程与轻量级进程

根据 Solaris 多线程结构模型，提出了用户级线程与轻量级进程两个概念。

用户级线程（即通常所说的线程，thread），它通过系统程序库函数来实现。用户级线程对系统内核来说是透明的，也就是说线程的创建、销毁、阻塞和激活都不需要内核的参与。它仅在进程内部可见，共享进程的地址空间、进程指令以及某些数据（称为共享数据）等所有资源，

但线程的标识符、寄存器状态(包括程序计数器和堆栈指针)、堆栈、信号掩码、优先级和线程局部存储数据是每一线程私有的。值得特别注意的是共享数据,如果被一个线程改变,则该变化可立即被其他线程所见。此外,进程中任何一个线程调用退出函数 exit(),则其他线程也将自行销毁,并结束进程。由于线程间没有系统强制的保护措施,在进行多线程编程时就需要格外的留心谨慎。

而轻量级进程(Lightweight Process)负责与系统内核通信,真正将用户级线程实现。每一个 LWP 可以看成是用来执行代码和系统调用的"虚"CPU,它是由系统内核单独分派的,在多处理器(CPU)环境中可以并行执行。系统中所有的 LWP 根据它们的优先级由内核负责调度。

轻量级进程与线程间的调度可以由图 2 表示。其具体过程如下:首先,LWP 在进程内存空间中确定线程状态后,选择某一线程准备开始运行(a);在 LWP 装载该线程的标识符和寄存器之后,开始执行线程指令(b);当该线程不能继续执行或应该执行其他线程时,LWP 将存储该线程的状态(c)并选择另一个线程执行(d)。

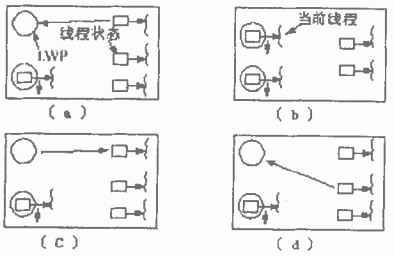

图 2　线程与轻量级进程间的调度

3. 线程的绑定与非绑定

用户级线程与轻量级进程之间并不是一一对应的关系,根据线程的实现方式、使用的库函数或程序员编码参数的不同,用户级线程在其生命期可以与同一个或不同的轻量级进程相关联,这分别就是线程的绑定(bound)与非绑定(unbound)。从图 1 来看,进程 1 属于单线程结构,用户级线程与 LWP 为绑定关系;进程 2 属于多线程对单 LWP 的非绑定关系;进程 3 属于多线程对多 LWP 的非绑定关系;进程 4 属于多线程对多 LWP 的绑定关系;而进程 5 是多线程与多 LWP 绑定与非绑定关系同时存在的综合,更为特殊的是其最后一个线程通过 LWP 直接与处理器绑定,则该处理器只为这一个线程服务。

线程的绑定与非绑定两种方式各有其优缺点,其比较见下表所示。

非绑定线程	绑定线程
缺省创建的线程	在线程创建时需指明线程绑定参数
可在轻量级进程(LWP)间进行切换	与 LWP 一一对应
创建速度快	创建速度慢,大约比非绑定线程慢 7 倍
线程阻塞时,放弃对应 LWP,允许其它等待 LWP 的线程使用该 LWP	线程阻塞时,仍与对应 LWP 绑定,不允许其它等待 LWP 的线程使用该 LWP
处理器需花费时间处理线程间的相互切换	处理器无需花费时间处理线程间的相互切换
适宜多线程的一般使用	适宜数学计算中使用的多线程

由于绑定线程比起非绑定线程的开销要大,因此,仅当线程所需资源(如虚拟的定时器或者一个指定的堆栈)只能从当前对应的 LWP 中获得,或者为了实现实时调度而必须使线程对于内核可见的场合下,才需要使用绑定线程。

4. 线程局部存储

单线程的 C 语言程序有全局变量和局部变量两种数据类型,而在多线程操作系统中增加另一种特殊类型的数据——线程局部存储(Thread Local Storage)数据。

所谓线程局部存储数据就是指一个进程可以拥有以线程为单元的全局变量,即某个进程中的所有线程可以拥有好象在全局内起作用的数据,但实际上只限于特定线程的范围之内。例如:一个多线程程序,每一线程对应写入一个单独的文件(即需要单独的文件描述符),此时采用线程局部存储就能十分方便地完成程序的需求。每个线程专用的数据是用进程中唯一的关键字(KEY)来标识,通过该关键字,每一个线程就可以来存取对应的线程局部存储数据了(其函数接口见第三部分)。

三、多线程的同步机制与函数接口

进程中的多线程存在共享数据,如果对这些数据不加特殊的处理,会造成程序数据的混乱和错误,有可能还会造成程序的死锁。为了控制多线程对共享数据处理的问题,即多线程同步问题,Unix 提供了多种同步对象供用户使用,包括互斥锁(Mutual Exclusion Lock)、条件变量(Condition Variable)、计数信号量(Counting Semaphore)和读写锁(Readers/Writer Lock)。由于介绍 Unix 多线程同步机制的书籍已经较多,这里便不赘述了。

用户级的多线程接口函数是由系统共享对象库 libthread.so 实现的。由于 Solaris 对 POSIX(Portable Operating System Interface for Computer Environment)标准的

支持,还提供 POSIX 线程实现的共享对象库 libpthread.so。根据其功能进行分类,多线程函数可用下表表示。

功能	函数名	POSIX 函数名
线程创建	thr_create()	pthread_create()
线程终止	thr_exit()	pthread_exit()
线程同时性控制	thr_getconcurrency() thr_setconcurrency()	无
向线程发送信号	thr_kill()	pthead_kill()
设置线程信号掩码	thr_sigsetmask()	pthread_sigmask
线程号获得	thr_self()	pthread_self()
主线程判断	thr_main()	无
线程放弃执行	thr_yield()	无
线程挂起与继续执行	thr_suspend() thr_continue()	无
线程终止等待	thr_join()	pthread_join()
线程优先级控制	thr_getprio() thr_setprio()	pthread_getschedparam() pthread_setschedparam()
线程局部存储数据控制	thr_keycreate() thr_getspecific() thr_setspecific()	pthread_key_create() pthread_getspecific() pthread_setspecific() pthread_key_delete()
线程号比较	无	pthread_equal()
取消线程执行	无	pthread_cancel()

四、多线程举例

以下是多线程的一个实例。

```
#define _REENTRANT
#include <stdio.h>
#include <thread.h>
#define NUM _ THREADS 12
void * change _ global _ data(void);
main(int argc,char * argv[]) {
  int i=0;
  for (i=0; i< NUM _ THREADS; i++) {
    thr _ create(NULL, 0, change _ global _ data, NULL, 0, NULL);
  }
  while ((thr _ join(NULL, NULL, NULL) == 0));
}

void * change _ global _ data(void * null) {
static mutex _ t Global _ mutex;
static int Global _ data = 0;
mutex _ lock(&Global _ mutex);
Global _ data++;
printf("%d is global data \ n",Global _ data);
mutex _ unlock(&Global _ mutex);
return NULL;
}
```

该实例十分简单,除主线程外,另有十二个子线程执行 change _ global _ data()函数。该程序主要是通过互斥锁函数 mutex _ lock()和 mutex _ unlolck()完成多线程环境下对共享数据 Global _ data 的保护。如果不进行数据保护,则可能出现以下情况:线程 1 执行完 Global _ data ++后(设此时 Global _ data 等于 1)被线程 2 中断,线程 2 也执行到语句 Global _ data++结束(这时 Global _ data 等于 2),线程 1 恢复执行,打印 Global _ data,其期望值为 1,但实际输出值为 2,说明该共享数据被破坏了。由这个小例子可以说明:在多线程编程中,一定要特别注意线程间的同步问题,保护共享对象。

多线程是 Unix 新增加的功能,具有很多优点,它的引入对程序设计结构、编程的方法思路都会产生较大的变化,只有通过不断的实践和学习才可能真正地掌握和利用它,更好地为我们服务。

(收稿时间:1999 年 9 月)