基于多线程的 Linux下并发服务器的实现研究*

张志佳[1] 于立国[2] 李海滨[2] 王东署[3] 苑 葳[4]

([1]沈阳工业大学 沈阳 110023 [2]中国科学院沈阳自动化研究所 沈阳 110016
[3]东北大学人工智能与机器人研究所 沈阳 110004 [4]中国联通辽宁分公司 沈阳 110003)

摘 要: 探讨了 Linux平台下多线程技术和套接字网络通讯问题,在此基础上利用互斥锁和条件变量技术设计了一个面向连接的多线程并发服务器的详细算法。最后给出了基于 Posix线程库的 Linux系统下用 C++实现多线程并发服务器的基本程序框架。实际项目应用表明这种基于多线程的并发服务器结构在完善程序功能的同时还可以有效提高其服务性能。
关键词: 多线程 并发服务器 套接字 Linux系统

The Implement of Multithread Concurrent Server in Linux

ZHANG Zhijia[1], YU Liguo[2], LI Haibin[2], WANG Dongshu[3], YUAN Wei[4]

([1] Shenyang University of Thechnology, Shenyang 110023, China
[2] ShenYang Institute of Automation, Chinese Academy of Sciences, Shenyang 110016, China
[3] Institute of Artificial Intelligence and Robotics, Northeastern University, Shenyang 110004, China
[4] China Unicom Liaonong Branch, Shenyang 110003, China)

Abstracts: The paper discusses the techniques of multithread and the socket communication under the Linux platform. Then, based on mutual exclusion and condition variables, a detailed algorithm to implement the connection-oriented multithread concurrent server is given. Finally, the basic program frame to implement multithread concurrent server with C++ language under Linux system based on Posix thread library is presented. The application shows the proposed frame can improve the service performance while expand the program's functions effectively.
Keywords: Multithread, Concurrent Server, Socket, Linux

1 引言

面向连接的并发服务器是目前 Linux网络服务器的主流形式。它采用主、从服务器的工作方式,较好地解决了网络中客户进程的并发请求问题。但从其工作方式和过程分析,还存在不足,主要在于并发服务器的父、子进程之间缺乏一种有效的进程间通信机制,使得父进程不能动态有效地管理子进程[1]。

进程(Process)与线程(Thread)是现代操作系统进行多任务处理的核心内容。进程是计算机为所有在其上运行程序进行资源管理的最小单位,一个进程内部可以有一个或多个线程。相对于进程,线程就是指进程的一条执行路径。

Linux的内核本身并不涉及到线程处理,而是纯粹以进程为处理器调度单位。但是 Linux提供了 Linuxthreads库,它实现了符合 POSIX1003.1c标准的多线程支持[2]。多线程的实现方式分为用户级、内核级、用户级与内核级相结合三种[3]。和进程相比,线程的最大优点之一是线程间数据的共享性,各个线程共享父进程处沿袭的数据段,可以方便的获得、修改数据。位于同一进程内的线程共享进程的执行代码和大部分

本文于 2005-12-06收到, 2006-05-23收到修改稿。
* 基金资助: 国家 863计划资助项目(2001AA422270)。

数据, 共享数据被一个线程修改之后可以被进程内的其他线程见到。但同时这也给多线程编程带来了许多问题尤其是线程间的同步问题。正确处理进程内多个线程同时访问相同变量的情况必须要解决线程间的同步问题。互斥锁和条件变量是实现同步的基本架构模块[4]。

基于多进程的并发服务器对于每客户连接需要启动一个进程, 而多线程的并发服务器处理每一客户连接是用同一进程内的不同线程。使用多线程技术, 在系统的机制、对客户事件的响应速度等方面能够有效的提高系统的性能并能显著改善程序结构。本文针对在并发服务器中使用多进程进行管理的缺点, 设计了一个基于多线程的 Linux平台下并发服务器的实现框架并给出了一个开发实例。

2　多线程并发服务器框架及实现

2 1　socket套接字技术

套接字 (socket)是网络在传输层上提供给应用程序的接口之一, 其目的主要是网络进程通信[5]。而网间网进程的首要问题是进程标识。在同一主机上, 可以用进程号唯一来标识进程。但在网络环境中, 各主机独立分配的进程号是不能作为进程标识的, 所以网间网进程标识需要全网唯一的主机地址来参与。为了在主机范围内唯一标识进程, Socket采用比进程号更低级的端口号作为进程标识。

使用套接字技术, 有面向连接和无连接两种方式。面向连接方式采用的是 TCP协议, 能提供可靠的通讯流服务并能保持数据的发送顺序; 无连接的方式采用的高层协议是 UDP, 它不像 TCP协议那样维护一个连接, 而是把目标 IP地址包含在数据包内一起发送, 不能保证数据传输的有序性和可靠性。本文采用面向连接的套接字编程技术。

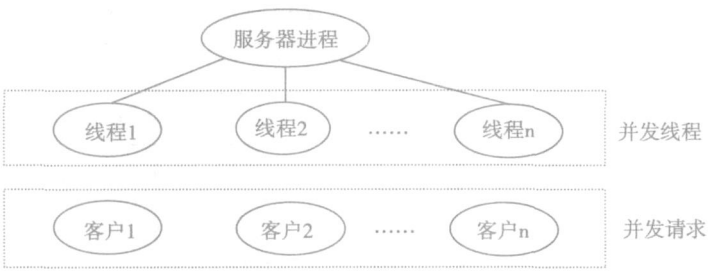

图 1　并发服务器模块

2 2　多线程并发服务器基本框架

并发服务器面向不定长时间才能处理完的请求, 对每个请求由服务器的线程处理。线程被用来建立请求驱动的服务程序, 每个客户一个线程, 多个线程可以并发执行。线程使得服务器的进程数目并不随客户数目的增加而线性增加, 这样减少了服务器进程的压力, 降低了开销。并发服务器的结构如图 1所示, 在某一个时刻, 由同一服务器进程所产生的多个并发线程同时对多个客户的并发请求进行处理, 从而解决了并发请求问题。各个线程既能独立操作, 又可以协同作业, 实现了简单而高效的服务器结构。

在并发系统中, 互斥锁 (Mutual exclusion)是最基本的同步方式, 用于保护临界区的数据。多个线程或进程共享一个互斥锁的时候, 在任何时刻这个互斥锁只能被一个线程或进程所获得, 其他线程或进程试图获得被其他线程或进程锁定的互斥锁都将被阻塞或返回一个 EBUSY 错误, 这样的机制能保证在任何时刻只有一个线程或进程执行临界区中的指令, 从而达到保护共享数据的目的。互斥锁有一个明显的缺点, 那就是它只有两种状态: 锁定和非锁定。而条件变量通过允许线程阻塞和等到另一个线程发送信号的方法弥补了互斥锁的不足。条件变量在使用时一般都要和一个互斥锁进行关联。本文中通过互斥锁和条件变量的协调使用来解决并发服务器中的同步通信问题。

并发服务器的工作流程按照三个步骤建立: ¹ 建立套接字并在某一约定端口上等待接收客户请求; º 当接收到来自客户端的服务请求后: 建立一新线程来处理, 同时主线程继续等待其他客户连接。当新线程处理完成后, 关闭新线程与客户的通信链路并终止新线程。 » 关闭服务器。

图 2详细描述了采用面向连接方式的并发服务器和客户端的算法。

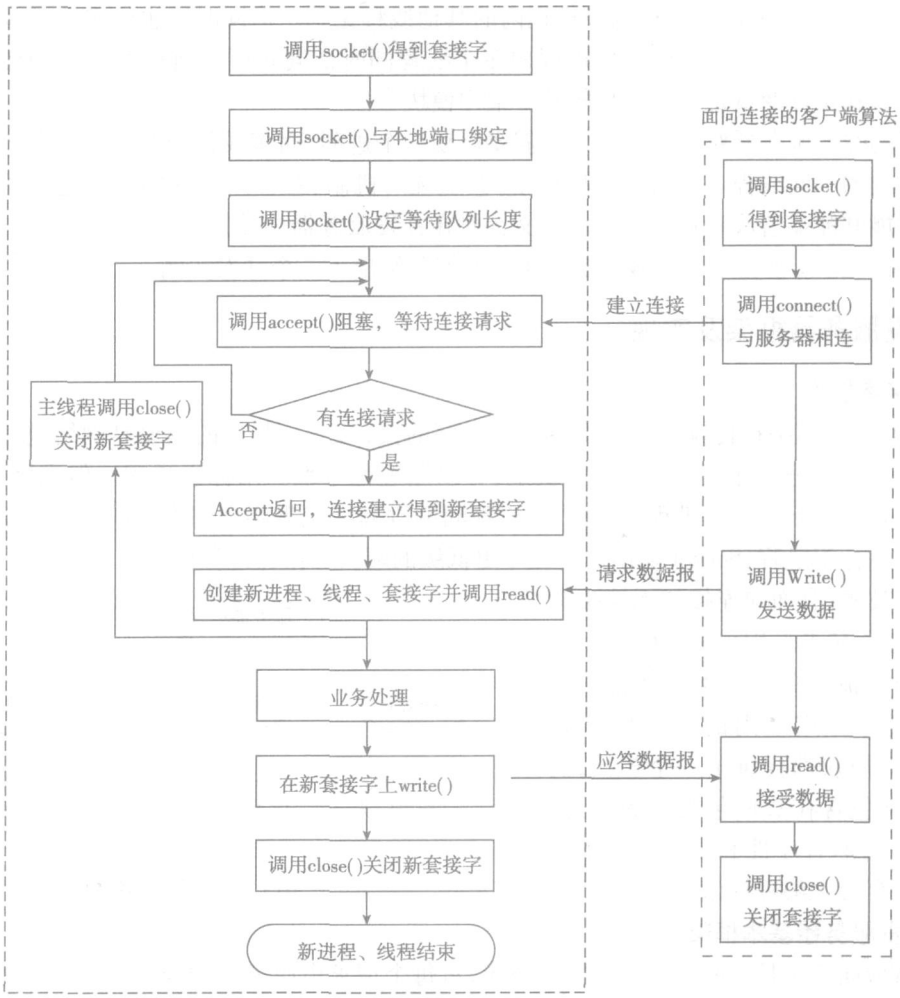

图 2 面向连接的并发服务器算法

基于多线程的 Linux系统下并发服务器能够同时并有效地运行多个任务。采用这种并发系统结构,既增加了服务器程序的功能,又提高了其性能。只有一个处理器时,多个线程不能真正地并行执行。然而,经过固定的时间间隔或当一个线程处于等待状态时,就可从一个线程切换到另一线程,实现多个线程重叠执行。在有多个 CPU的计算机上,如果底层系统支持,则多个线程可真正地被并行执行,服务的并行性潜力得以开发。

3 多线程并发服务器的应用实例

按照上节给出的算法思想,下面给出了一个面向连接的多线程并发服务器的一个应用实例框架,其中线程管理部分用到了线程间同步的方法。

```
# include线程库文件和 socket的库文件
……
pSTRTHREAD g_pHead, pThreadChainTail
                                    //线程链表的头、尾指针
STRUCTSYNC g_struSync;
                    //定义一个实现同步用的数据结构对象
/* main函数实现了一个并发服务器,主线程创建的管理线程和临时交易线程之间进行数据* /
intmain( int argc, char* argv[] )
{ ……
  CManageThread* pManageThread = new CManageThread ();
                                    //生成管理线程的一个实例
  pManageThread - > Start();
  //调用 socket()、bind()、listen(),建立 socke套接字描述符
  for(; ; ) {
    //调用 accept(),循环接受客户的连接请求
    CBusinessThread * pBusinessThread = new CBusiness-
        Thread();
```

```
                    //生成一个临时交易线程的实例
    pBusinessThread->Start();    //运行交易线程
  }
  //调用 close();关闭 socket套接字
  ......
}
```

CManageThread和 CBusinessThread派生自同一个线程基类 CthreadBase。CthreadBase类定义了一个静态成员函数即调用 pthread_create建立线程时运行的函数指针,在派生用户线程类时只需实现各自的 Run函数。

下面给出 CBusinessThread和 CManageThread各自实现的 Run方法,CBusinessThread工作线程和 CManageThread管理线程都操作一个共享的数据结构,CBusinessThread与 CManageThread形成一对提供/消费者的关系,CBusinessThread在线程退出时增加共享数据结构中的记数并增加链表指针,CManageThread每收到 CBusinessThread的信号后使共享数据结构记数减一并减少链表指针,演示了线程间同步的用法。

```
void CBusinessThread::Run()
{ ……
  //线程退出时进行以下操作
  pthread_mutex_lock(&g_struSync.mutex);
   //获得互斥锁,进入临界区
    //链表操作,把线程自身添加到链表头
    g_struSync.iCount++;    //增加统计记数
    pthread_cond_signal(&g_struSync.cond);
  //通知管理线程
  pthread_mutex_unlock(&g_struSync.mutex);
  //出临界区,释放互斥锁
}
void CManageThread::Run()
{ ……
  while(线程运行条件满足){
    pthread_mutex_lock(&g_struSync.mutex);
    //获得互斥锁,进入临界区
      if(g_struSync.iCount==0)
      pthread_cond_wait(&g_struSync.cond,&g_struSync.mutex);
    if(g_struSync.iCount>0){
      if(g_pHead!=NULL)
      //链表操作,取出 g_pHead执行线程相关清理
      g_struSync.iCount--;    //减少统计记数
      pthread_mutex_unlock(&g_struSync.mutex);    //出临界区,释放互斥锁
    }else
      pthread_mutex_unlock(&g_struSync.mutex);    //出临界区,释放互斥锁
    }
}
```

4 结束语

并发服务器是目前应用较为广泛的网络服务器的实现形式。本文在探讨了进程和线程之间的联系和区别并详细说明了 Linux下的多线程运行机制和同步问题,并且建立了一个面向连接并发服务器实现框架。基于多线程的并发服务器在简化程序结构和机制的同时能有效地提高服务器的运行效率以及系统性能。根据该方案的基本思想加以扩展深化,可以解决具体的网络通信问题,这对提高网络服务器的效率具有一定的现实意义。

参 考 文 献

1 彭岚,周启海. UNIX并发服务器及其几种优化改进方案. 计算机应用, 2005, 24(4): 47~49

2 郑燕飞,余海燕. Linux的多线程机制探讨与实践. 计算机应用, 2001, 21(1): 81~83

3 张炯. UNIX网络编程实用技术与实例分析. 北京:清华大学出版社, 2002

4 W. Richard Stevens UNIX网络编程. 北京:清华大学出版社, 1999

5 宋燕红,马礼. 多线程并发服务器的实现. 华北工学院学报, 1998, 19(2): 124~126

6 喻志虎. UNIX平台下 C语音编程. 北京:清华大学出版社, 2001

作者简介

张志佳,男,(1974-),博士,研究方向为模式识别、软件工程等。

于立国,男,(1978-),工程师,研究方向为软件工程、数据通信等。