文章编号:1001-5132（2008）01-0049-06

基于嵌入式系统的线程池研究

王让定，李　霏，徐　霁

（宁波大学 纵横智能软件研究所，浙江 宁波 315211）

摘要：服务器响应客户请求一般采用并发式服务，但普遍存在着开销大及资源不足等问题，而线程池技术是解决这些问题的一个良好解决方案. 在 POSIX 线程库研究的基础上，提出了以 ARM9 为硬件平台进行嵌入式服务器线程池设计，并优化了线程池设计的几个关键问题.

关键词：并发式服务；线程池；嵌入式系统；POSIX 线程库

中图分类号：TP309　　　　文献标识码：A

计算机技术在经历了几十年的发展后，互联网已经深入到千家万户，各种基于服务器/客户端系统的分布式应用比比皆是，如P2P网络视频直播和网络游戏服务等，但这些类型的应用对系统的服务质量和实时性要求都很高. 在一般情况的下，服务器有 2 种服务模式——循环模式和并发模式. 由于考虑到循环模式响应速度慢，所以一般服务器都采用并发模式，即每当收到 1 个客户请求时，服务器就自动产生 1 个子线程来处理该请求，而主线程继续进行其服务器端的工作. 线程虽然属于轻量级进程，具有占用资源小、共享内存及线程间任务切换快等优点[1,2]，但是当存在大量客户请求时，开辟、销毁线程将浪费大量系统时间和资源，以至于在创建及销毁子线程上花费的时间和资源要比花在完成用户请求的时间和资源更多，显然是得不偿失的.

在此情况下，研究者们提出了线程池(Thread Pool)的概念[3]. 线程池技术为线程创建、销毁的开销问题和系统资源不足问题提供了很好的解决方案，能有效地提高系统实时性和整体性能. 其主要思想是在服务器启动时，自动创建 1 个对象池，并在这个对象池中创建 N 个空闲线程. 当有客户请求时，服务器自动分配给该客户 1 个空闲线程. 当池中所有线程都处于繁忙状态时，则该请求自动进入等待队列.

线程池主要有工作组模型、主从模型和管道模型 3 种. 文献[3]指出了主从模型具有易于管理、可移植性高、易于开发等特点，这些特点更符合嵌入式环境下系统开发的要求. 主从模型是指线程池中有 1 个或多个线程处于管理者的地位(Master)，而其他线程处于被管理的地位(Slavers). Master 接受用户的请求，并将其指派到其他 Slavers 线程执行，模型如图 1 所示.

对于主从模型而言，1 个基本的线程池至少应包含：(1)线程池管理模块(主线程)；(2)工作线程模块(从线程)；(3)任务等待队列. 采用该模型来进行服务器的并发服务控制，相对于传统的并发服务模式，有利于延长任务实际执行时间，减少服务器的

收稿日期：2007-07-18. 宁波大学学报（理工版）网址：http://3xb.nbu.edu.cn
基金项目：国家自然科学基金（60672070）；宁波市科技攻关项目（2006B100067）.
第一作者：王让定（1962－），男，甘肃天水人，博士/教授，主要研究方向：数字水印及语音识别. E-mail: wangrangding@nbu.edu.cn

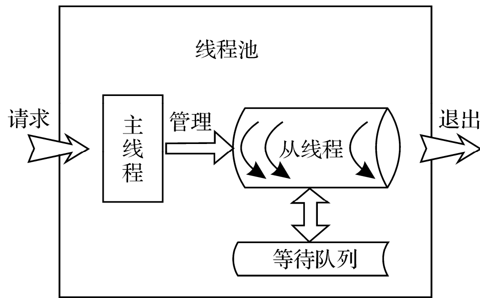

图 1 主从模型

负荷，提高服务质量. 要使主从模型线程池的工作效率达到最优化，需要解决如下几个关键问题：(1)线程池的初始大小如何选择；(2)线程池如何根据实际情况进行动态调整；(3)线程池如何解决线程死锁和线程泄露的问题.

针对上述的一些问题，研究者们也提出了一些自己的解决方案[3-8]，但是大多数研究都集中在大型服务器领域，对嵌入式系统下线程池的研究较少. 而嵌入式系统硬件相对简单，需要完成的功能也相对单一，有些算法和方案并不一定适合嵌入式系统. 本文基于POSIX线程库的主从模型线程池设计方案，研究了针对所开发的嵌入式(ARM9)视频服务器的线程池设计问题，并针对主从模型线程池设计的几个关键问题提出了相应的解决方案.

1 基于 POSIX 的嵌入式线程池系统

1.1 系统工作流程

服务器启动时，自动采集设备信息，选择好线程池的初始大小，创建管理线程. 这时候服务器就可以处理其他的服务，而将线程池交与管理线程进行管理. 这样更有利于提高服务器的响应速度和并发性. 当服务器脱管后，管理线程自动创建 N 个从线程作为工作线程. 此时线程池进入就绪状态，客户任务经过等待队列接口，向主线程发送请求. 如果此时有空闲工作线程，那么主线程就将该工作线程分配给该任务. 在任务完成后，需要子线程自动返回入池. 如果有异常，子线程没有返回入池，那么管理线程自动创建 1 个新的子线程. 线程池系统的工作流程如图 2 所示.

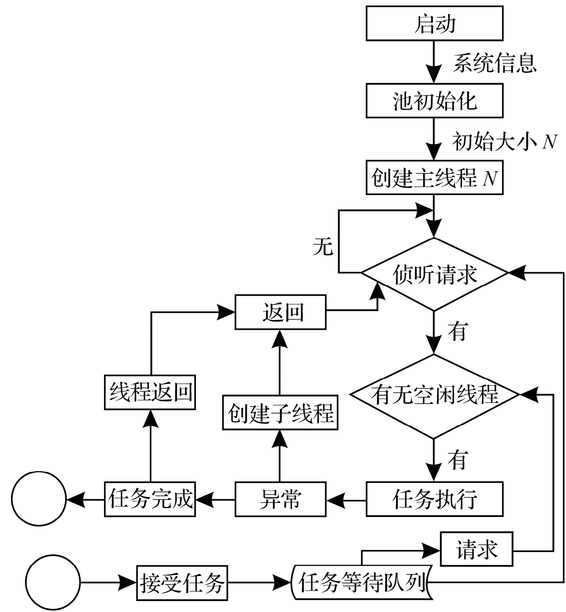

图 2 嵌入式线程池系统工作流程

1.2 线程池初始大小的选定

前已所述，线程池的初始大小是线程池设计的关键问题之一. 如果选的过小，那么在客户请求高峰期，管理线程就需要频繁创建子线程来满足客户需求，这样就花费了大量时间在创建子线程上. 但如果选的过大，那么又会浪费系统资源，这对于资源本来就比较紧张的嵌入式系统来说也是不可容忍的.

系统在选择线程池的大小时，主要考虑客户任务数和系统硬件水平这 2 个因素. 文献[5]指出线程池的最佳大小取决于可用处理器数目和客户任务的性质. 若是计算性质的任务，$N=$ 处理器数 $+1$；若是 I/O 任务，$N=$ 处理器数 $\times(1+W/S)$，其中 N 为线程池初始大小，W 为响应时间，S 为服务时间. 考虑到一般的嵌入式系统都是单 CPU，所以用处理器数来计算其线程池大小是不合适的. 目前越来越多的微处理器支持超线程技术(Hyper-Threading Tech)，利用特殊硬件指令，把处理器内部的 2 个逻辑内核模拟成 2 个物理芯片，使多个线程同时运行于 1 个超线程处理器上，效率最大提高达 30%.

因此，本文设计将处理器数改为超线程处理器的内核数，这样就更适于嵌入式系统的实际情况. 另一方面，考虑到视频监控系统功能单一，其每月每天的用户访问量服从一定的概率分布，而且每月每天的访问量相对稳定，所以可以让系统自动统计最近 30 d 的当日最高同时访问量，计算其数学期望. 最后，线程池的初始大小 N 就是上述二者的加权平均值. 综上所述，计算嵌入式线程池系统的初始大小，可引出以下公式.

若是计算任务，则：

$$N=\left\lceil (kernel+1)\times 0.5+0.5\sum_{i=1}^{30}x_i p(x_i)\right\rceil .$$

若是 I/O 任务，则：

$$N=\left\lceil (kernel\times(1+w/s))\times 0.5+0.5\sum_{i=1}^{30}x_i p(x_i)\right\rceil ,$$

其中，$kernel$ 为处理器内核数；w 为响应时间；s 为服务时间；x_i 为当日最高同时访问量；$p(x_i)$ 为 30 d 中出现 x_i 的天数的概率，$\lceil\ \rceil$代表向上取整. 考虑到视频监控系统中计算任务和 I/O 任务的比例，分配计算任务、I/O 任务和访问量数学期望的权值分别为 0.3, 0.4, 0.3 . 最后线程池初始大小的选定由以下公式决定：

$$N=\left\lceil (kernel+1)\times 0.3+(kernel\times(1+w/s))\times 0.4+0.3\sum_{i=1}^{30}x_i p(x_i)\right\rceil .$$

1.3　线程池的动态调整

线程池的动态调整一般都采取反馈的办法[3,6]. 通过采集的一些系统信息，比如响应时间及吞吐量等，以判断当前线程池的规模是否不足或过剩，然后通过反馈调整，来使线程池维持在一个最经济、最有效的状态. 由于受到嵌入式设备硬件条件的制约，为减轻系统负担，本系统线程池的动态调整主要由管理线程(主从模型主线程)单独完成，其收集的信息主要有如下几种：

(1) 空闲线程(从线程)数；

(2) 最大等待队列时间；

(3) 等待队列任务数；

(4) 吞吐量.

其中，吞吐量 = 完成任务数/T，T为采样周期. 这 4 种信息的后 3 种都可以直接从系统中读取或计算出来. 但是对于第 1 种信息——空闲线程数来说，本文采用的则是空闲线程最大堆算法[3]，其为每个子线程分配 1 个计数器，每次管理线程优先将计数器最大的空闲线程交给任务，这样负载就会集中于部分线程，更有利于发现长期空闲的线程，缩小线程池的规模.

根据收集的后 3 种信息，线程池的反馈管理思想可设计如下：

(1) 在线程池初始化时，创建线程数有上阈值 $Thread_{\mathrm{top}}$；

(2) 当发现最空闲线程时，就暂时将其休眠，并将其存入 1 个休眠队列，而该休眠队列中的线程可随时被唤醒，再被加入到工作线程中. 当休眠队列满时，可采用 FirstIn-FirstOut 策略销毁. 此种作法是防止频繁的销毁线程，反而增加了线程的销毁时间，违背使用线程池的初衷.

(3) 针对后 3 个信息，本文设计了 3 个参数作为系统管理员对线程池运行效果的期望，分别为：$WaitTime_{\mathrm{hope}}$，$WaitJob_{\mathrm{hope}}$和$ThroughPut_{\mathrm{hope}}$. 然后分别用最大等待队列时间、等待队列任务数和吞吐量与它们相比，得出比率分别为$WaitTime$，$WaitJob$和$ThroughPut$. 最后可引出如下调整计算公式：

$$p=-(0.3\sqrt[3]{1-WaitTime}+0.3\sqrt[3]{1-WaitJob}-0.4\sqrt[3]{1-ThroughPut}) .$$

当 $p>0$ 时，表示线程池的处理能力不足，应在当前线程池的规模上增加 p 个百分比. 当 $p<0$ 时，则不需作出调整.

1.4　线程死锁和线程泄露的处理

本文对死锁问题采取先预防后检测的方法. 对于一般性死锁来说，首先编写程序时要注意程序逻辑，当必须同时锁定 2 个资源时，尽量保证在任何时刻都应该按照相同的顺序来锁定资源. 假设 A、

B 想同时锁定资源 a、b，那么 A 如果以 a → b 的顺序锁定，B 也必须从 a → b 的顺序锁定，则不可能出现 A 占有 a 申请 b，B 占有 b 申请 a 的情况，死锁也就不会发生. 因此可以证明这样可大大减少发生死锁的概率

对于线程池存在的特殊死锁，产生的情况比较复杂，举个简单的例子：假设线程池有 2 个子线程，某一时刻有 2 个用户想要访问 Web Server，那么将 2 个子线程分配给这 2 个用户. 子线程首先要建立 socket 连接，假设它还需要自己开启子线程来传输数据，这时线程池已经没有多余的子线程了，于是 2 个子线程都无法运行下去，线程池便死锁了. 因此可以得出的结论为：当子线程需要靠其他子线程来帮助自己完成任务时，这种死锁便会发生. 针对这种情况，则必须为每个线程设定 1 个定时器，如果线程运行时间超时，则调用死锁检测算法，其算法描述如下：

(1) 初始化 $\mathrm{Work}[i]=\mathrm{Available}[i]$；

(2) 如果 $\mathrm{Need}[j,i]=0$，则 $\mathrm{Finish}[j]=\mathrm{true}$，否则 $\mathrm{Finish}[j]=\mathrm{false}$；

(3) 找 1 个能满足如下条件的线程 thread$[j]$：$(\mathrm{Finish}[j])=\mathrm{false})\|(\mathrm{Request}[j,i]\le \mathrm{Work}[i])$，如果存在，则转至(4)，否则转至(5)；

(4) $\mathrm{Work}[i]=\mathrm{Work}[i]+\mathrm{Allocation}[j,i]$；$\mathrm{Finish}[j]=\mathrm{true}$；转至(3)；

(5) 如果所有的 $\mathrm{Finish}[j]=\mathrm{true}$ 不成立，则系统出现死锁；而且 $\mathrm{Finish}[j]=\mathrm{false}$ 的线程 thread$[j]$ 出现死锁.

其中，Available$[i]$ 代表 i 类资源是否被占有；Need$[j,i]$ 表示线程 j 还需要资源 i 的个数；Request$[j,i]$ 表示线程 j 申请的资源 i 的个数，Allocation$[j,i]$ 表示线程 j 占有资源 i 的个数，Finish$[j]$ 表示线程 j 是否完成任务. 当检测出发生死锁的线程后(可能是 1 个或多个)，则在这些线程中消除超时线程的任务，然后分配空闲线程给任务等待队列，任务等待队列采取 FIFO 策略.

线程泄露大部分是由于线程运行时发生异常，没有进行及时地回退和捕捉. 因此为防止线程泄露，当子线程完成任务后，应主动返回到线程池中，其部分代码如下：

```
save_thread(_threadpool*pool, _thread*thread)
    //让工作完成的子线自动回归线程池，
{
  pthread_mutex_lock(&pool→tp_mutex);
  if (pool→tp_index<pool→tp_max_index)
  {
    pool→tp_list[pool→tp_index]=thread;
    //将子线程 ID 交给线程池，
    pool→tp_index++;
    pthread_cond_signal(&pool→tp_idle);
    //将该子线程状态设为空闲，
    ……
  }
  pthread_mutex_unlock(&pool→tp_mutex);
}.
```

如果线程回归时发生异常，则需要及时的进行错误状态捕捉，其部分代码如下：

```
if(!(error=pthread_join(thread,&exitcode)))
    //得到错误状态码，
  fprintf (stderr, “The exit code was%d\n”,
    *exitcode);
pthread_cond_destroy(&thread→cond);
    //销毁该线程的条件变量，
pthread_mutex_destroy(&thread→mutex);
    //销毁该线程互斥锁，
Free (thread);   //释放该线程内存空间.
```

2 基于 POSIX 的线程池算法流程

本文以 ARM9 为硬件平台，以 Linux 为操作系统，采用 POSIX 线程库实现线程池的设计和算法实现. POSIX 线程库具有代码量少及创建线程快

等优点,符合嵌入式平台编程思想. 基于 POSIX 的线程池核心代码和算法流程如下：

```
//创建线程池，
Threadpool create_threadpool (int num_threads_in_pool)
{
  _threadpool *pool;
  pool=(_threadpool *) malloc (sizeof(_threadpool));    //申请线程池内存空间，
  pthread_mutex_init( &pool->tp_mutex, NULL);
  pool→tp_max_index=num_threads_in_pool;
  ……  //初始化变量，
  pool→tp_list=(_thread**)malloc(sizeof(void*)*MAXT_IN_POOL);   //申请子线程数组内存，
  for(……){
    _thread*thread=(_thread*)malloc(sizeof(_thread));    //产生子线程，
    thread→id=0;
    pthread_mutex_init(&thread->mutex, NULL);
    pthread_cond_init(&thread->cond, NULL);
    pthread_create(&thread->id, &attr, wrapper_fn, thread);}
return (threadpool) pool;
  }
//分配任务给子线程，分配时总是把最大下标空闲子线程配出，
int dispatch_threadpool(threadpool from_me, dispatch_fn dispatch_to_here, void *arg)
{
  _threadpool *pool=(_threadpool*)from_me;
  pthread_mutex_lock(&pool→tp_mutex);
  thread=pool→tp_list[pool→tp_index];
  thread→fn=dispatch_to_here;
  thread→arg=arg;
  thread→parent=pool;
  ……
  pthread_mutex_unlock(&pool->tp_mutex);
  return;
}
//任务执行，并将子线程返回到池中，
void*wrapper_fn(void*thread)
{
  _threadpool*pool=(_threadpool*)thread→parent;
  thread→fn(thread→arg);    //任务执行，
  pthread_mutex_lock(&thread→mutex);
  if(0=save_thread(thread→parent, thread)) //子线程返回到线程池，
  {
    pthread_cond_wait(&thread→cond, &thread→mutex);    //子线程进入等待队列，
    }
  pthread_mutex_unlock(&thread→mutex);
}.
```

3　结论

本文在线程池设计的关键问题基础上,研究了嵌入式环境下(ARM9)主从模型线程池的解决方案，讨论了该方案下线程池初始大小的设定，反馈管理算法以及如何解决死锁和泄漏的问题，最后借助 POSIX 线程库设计并实现了该系统核心算法和功能. 在硬件能力相对不足的嵌入式系统下，研究其线程池问题有利于促进嵌入式系统的应用推广. 下一步，将继续研究线程池的优化问题，重点着眼于优化方案的算法实现和系统性能测试，以提高所研发的视频服务器的多用户处理能力.

参考文献：

[1] Chulho Shin, Lee Seong Won, Gaudiot Jean Luc. Adaptive dynamic thread scheduling for simultaneous multith-

readed architectures with a detector thread[J]. Parallel and Distributed Computing, 2006, 66: 1 304-1 307.

[2] 周昔平, 高德远. 网络处理器的线程级并行技术研究[J]. 微电子学与计算机, 2006, 23(7):78-80.

[3] 杨刚, 周兴社, 潘惠芳. 基于反馈的自适应线程驰管理框架[J]. 计算机工程, 2006, 32(5):65-69.

[4] Yibei Ling, Tracy Mullen. Analysis of Optimal Thread Pool Size[J]. ACM SIGOPS Operating Systems Review, 2000, 34(2):42-55.

[5] 李昊, 刘志镜. 线程池技术的研究[J]. 现代电子技术, 2004(3):77-80.

[6] 吴炜荣, 梁阿磊, 吴刚. 基于 POSIX 线程库的线程池反馈算法的设计与实现[J]. 微型电脑应用, 2006, 22(5): 52-59.

[7] Irfan Pyarali, Marina Aspivak, Ron Cytron. Evaluating and Optimizing Thread Pool Strategies for Real- time CORBA[M]//ACM Special Interest Group on Programming Language, Compiler and Tool Support for Embedded Systems: Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems, 2000. New York: ACM Press, 2001:214-222.

[8] Schmidt D C, Vinoski S. Object interconnections: real-time CORBA, part 3: thread pools and Synchronizers[J]. C/C++ Users Journal, 2002, 8:30-40.

Probing Design of Thread Pool Based on Embedded System

WANG Rang-ding, LI Fei, XU Ji

(CKC Institute of Artificial Intelligence and Software Engineering, Ningbo University, Ningbo 315211, China)

Abstract: Parallel service is mostly adopted in servers to respond to clients’ requests, but this service type is at expense of system cost and in lack of sufficient resources. Thread pool turns out to be a good alternative to tackle the problems mentioned. In this paper, based on the analysis of POSIX thread library, an embedded-thread-pool system (ARM9) with ARM9 as the hardware platform is proposed. In addition, optimization is applied in solving several main problems found in thread pool design.

Key words: parallel service; thread pool; embedded system; POSIX thread library

CLC number: TP309 **Document code:** A

（责任编辑　章践立）