文章编号：1008-0562(2011)04-0584-04　DOI: CNKI:21-1379/N.20110826.1645.001

基于 linux 用户态可自控缓冲区管理设计与实现

刘青昆，王　佳
（辽宁师范大学 计算机与信息技术学院，辽宁 大连 116081）

摘　要：为了避免创建缓冲区的过程中，由于多次释放和重新分配内存而导致可能的内存泄露和内存浪费等弊端，采用一种基于用户态的可自行调控的缓冲区管理机制，该管理机制是基于抽象缓冲区虚拟接口而设计的，具有自适应性，并且能够支持动态的内存分配、回收，缓冲区重用机制。研究结果表明：在基于零拷贝的集群并行通信系统中，采用该缓冲区管理机制的设计满足了网络通信系统的应用需求，该方法是一种高效的、可靠的、具有实时性的可兼容的缓冲区管理机制。

关键词：通讯优化；零拷贝；缓冲区重用；抽象缓冲区；内存分配
中图分类号：TP 393　　**文献标志码**：A

Design and implementation for an automatic buffer management based on LINUX

LIU Qingkun，WANG Jia
(School of Computer and Information Technology, Liaoning Normal University, Dalian 116081, China)

Abstract: To avoid the problem of memory leak and memory waste, which is caused by freeing and allocating memory, this paper proposes an automatic buffer management mechanism. The buffer management mechanism based on the abstract buffer has a good adaptive ability and supports dynamic memory allocation, recovery and reuse. The case study on the communication system with zero copy technology shows that the buffer management mechanism designed meets the requirements of a network communication system. In addition, it is highly efficient, reliable, and real-time compatible with the buffer management mechanism.

Key words: communication optimization; zero copy; packet buffer recycling; abstract buffer; memory allocation

0　引　言

随着网络功能的发展，网络中的数据包在复杂的网络环境中，对内存的有着不同的需求。原有的内存分配方法存在一定的缺陷性，仅仅依靠预先申请分配的一块大的缓冲区资源已满足不了系统通信性能的实时性要求，包长度不同会相应产生资源和内存的浪费，造成网络延迟。为了满足长度不一的即时数据的需求，考虑缓冲区的管理问题，分配和释放如果不在相同代码位置进行操作，则必须在某内存块不再使用时对其进行一次释放，否则容易导致内存泄露。考虑如何确定缓冲块大小问题，传统做法是保守的估计最大的数据尺寸，但这种预分配的内存大小比实际需求大出了很多，也没有一定的分配标准，造成系统资源的极大浪费[1]。

1　相关工作

许多缓冲区优化方法，都集中在 TCP/IP 等底层协议。针对数据复制等耗时操作，可采用全新通用硬件设施[2]。针对减少协议处理中数据复制问题，可采取零拷贝思想进行优化。针对提高缓冲区处理灵活性和效率问题，P.Druschel 提出 fast buffer 方法[3]。实现多层协议的高效处理，可采用 ILP 方法[4]。但是这两种方法都存在一定的缺陷，不是依赖特殊硬件或操作系统支持，就是不适合现有协议[5]。

为有效利用资源，提高系统的数据传输效率，使现有的零拷贝技术更具有实用价值和应用性能。针对复杂的应用层协议，提出一种不依赖操作系统或特殊硬件支持的，且完全可以在用户空间实现的可自控的缓冲区管理方法。该方法是一种基于内存

收稿日期：2010-11-22　**网络出版时间**：2011-8-26　**网络出版地址**：http://www.cnki.net/kcms/detail/21.1379.N.20110826.1645.001.html
基金项目：辽宁省博士科研启动基金资助项目（20051058）；辽宁省教育厅科研项目（20060479）
作者简介：刘青昆（1971-），男，河北 清苑人，博士，副教授，主要从事并行计算与分布式处理等方向研究；通讯作者：王佳，E-mail:wangluv@gmail.com。
本文编校：朱艳华

保护模式的，可配置的，支持动态分配、回收和重用的可自控的、不用预先判断数据大小的缓冲区管理方法。该方法要达到的目标要求，在有限的内存空间可以存放不同长度的数据，减少分配、回收和重用时遍历的次数，减少对预分配的内存进行长度估计，减少数据拷贝操作，对不再使用的缓冲块进行重用操作[6]以便下次数据到来时直接重新使用该数据块，避免了不断的重复申请新的内存空间。

2　动态缓冲区管理机制的设计

缓冲区的分配策略一般分为静态分配和动态分配两种[7]。静态分配是预先分配固定大小的一个缓冲块组，它不会分配失败，速度快，避免了接收数据产生的瓶颈，具有一定的实时性，但会因包长度的不同相应产生资源和内存的浪费。动态分配则是根据用户指定的大小分配空间，用户申请空间的建立在数据处理过程中，支持动态分配、释放和回收，提高了内存利用率，避免了静态分配空间浪费现象。综合分析比较，与静态分配相比，动态分配在资源利用方面更具有优势，它能够迅速分配数据空间，方便变长缓存操作，达到数据复制的最少化。

2.1　缓冲区回收机制

传统使用的动态分配内存方法，为缓冲区管理提供了灵活性，但相应的也带来了一定的影响。为了避免内存泄露，必须在使用完内存块之后对其进行释放，但是随着数据的不断流动，内存块的多次分配和释放在所难免。对于 CPU 资源来说，这又是一种资源浪费的策略。那么，缓冲区重用机制成为避免资源浪费的一种有效的手段。

缓冲区回收机制设计如下：

（1）回收操作　在回收操作中，分为全回收和半回收。全回收是指使用 used 标识的未使用的内存块全部挂入 free_q 中。半回收是指 frblock 标识为 0 时，释放的自身内存块挂入到 free_q 队列中，挂入 free_q 队列中的块可供以后继续使用，且不用再进行重新分配。因此节省了 CPU 分配内存所耗时间，提高了系统性能。

（2）重用操作　在接收过程中，首先判断缓冲区队列是否有被回收的数据块。如果有，则直接从 free_q 队列中获取该块以备使用，否则，从系统内存管理模块中分配新的数据块。

（3）碎片管理　对系统进行分配、回收等的不断操作过程中，必然会产生内存碎片。如何将内存碎片有效利用起来也是亟待解决的问题之一。对于动态分配内存产生的碎片，采用半回收和重用操作进行再回收利用。

2.2　内存管理设计

根据应用对象对网络通信等的需求，参考 4.4BSD MBUF 和 linux 中的 sk_buffer 设计，提出静态与动态分配相结合的一种缓冲区可自控管理优化策略[7]。该策略具有高效性、可靠性及兼容性良好等优点。

数据缓冲区的分配操作是由数据缓冲区实体定义和实际内存分配共同完成[1]。在实际应用时，数据变为可用数据以前是不需要进行分配实际内存的。由此可将定义操作和内存分配操作进行分离。为缓冲区定义一组属性标识码，标记方法通过定义属性是否可写、是否需要释放等属性来实现[5]。

本文设计的内存管理结构见图 1。

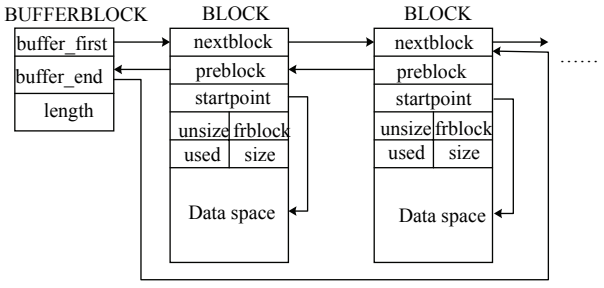

图 1　内存管理块结构

Fig.1　structure diagram of memory management block

BUFFERBLOCK 包含关于创建的内存管理的信息，它还管理动态存储内存块的一个链表。length 是块的个数，即缓冲区链表的长度信息，BLOCK *buffer_first 指向第一个块的指针，BLOCK *buffer_end 指向最后一个块的指针。

如果分配了内存，size 表示分配块的大小，每个块存放一个帧格式的以太包,且大小为 2K 字节。这样分配依据是因为路由器的 MTU 是 1 500 字节，加上以太包的 14 字节，一共不超过 1 514 字节。2K 字节大小刚好满足包的大小要求。在 LIUNX 2.4 内核中，内存每个页面大小为 4K 字节，这样块大小正好为页面大小的一半，方便后面的页操作。BLOCK *nextblock 是指向下一个块的指针，BLOCK *preblock 是指向上一个块的指针。startpoint 是未被使用部分的头指针，unsize 表示每

个块中未被使用的大小。used 是当前分配好的内存块是否被使用的标志位，当 used 的某一个标志位置 0，used 后面的所有内存块都不被使用，都可以释放。frblock 是当前内存块是否可以被释放的标志位，当 frblock 为 0 时，只释放它本身的内存。这样设置之后，不被使用的内存资源就会被释放出来，以待其它申请资源时使用，充分利用了现有资源，避免了 CPU 资源浪费，实现半回收机制。

2.3 内存管理主要实现例程

分开考虑缓冲区定义操作和内存分配操作。在缓冲区出现数据之前，必须先创建一个只用于分配一个包含内存管理的内存块，且初始化它的结构体的功能的抽象缓冲区接口。下面是用伪代码实现函数创建例程：

```
int initBlock( struct BLOCK * mem, struct
BUFFERBLOCK *data){
    while(i!=(data->length)){
        struct BLOCK *mem_new;
        add *mem_new in the mem;
        mem_new->nextblock=NULL; and i++;}}
```

创建抽象接口后，通过追加数据操作可以对到来的数据进行存储，使其保存到抽象缓冲区中。使用下面伪代码实现数据追加过程：

```
int inputBlock(FILE *file, struct BLOCK *mblock,){
    while(FILE *file!=NULL)  {
        struct BLOCK *ram;
        ram read the data of file;
        *ram=*(mblock->nextblock);
        if(there is no block) {
          dynamic buffer allocation;
          ram read the data of file;}}
}
```

如果要实现从抽象缓冲区逐段读取数据可以使用下面的伪代码，该函数销毁性地 buffer_block 指向的抽象缓冲区进行数据的读取操作，若内存块变为空，则将它们从链表中删除，然后返回成功读取的字节数目。

在使用抽象缓冲区之后通过调用下面的 freebuffer()函数进行销毁该缓冲区中的内存块：

```
int freebuffer(struct BLOCK *memblock) {
    while(memblock!=NULL)  {
        if(memblock->used==0)  {
            remove memblock and all the blocks
after memblock to the free_q;  and  break; }
        if(memblock->frblock==1)  {
            remove the memblock to the free_q; }
        memblock= memblock->nextblock;  }
}
```

利用内存块的一个空链表完成抽象缓冲区的创建。分配内存操作只在实际中数据变得可用时，同时还担负着释放内存的责任，这样就把内存管理操作进行了集中。它有如下几个优点：

（1）通过调用预定义的 API 函数可以构造或销毁内核和驱动的数据缓冲区。

（2）内存的使用将持续接近最优状态，内存泄露问题被最大化的解决了。

（3）只有存在额外数据空间时缓冲区才会进行数据复制，事实证明缓冲区溢出也不可能会发生。

2.4 数据传输分析与零拷贝

网络协议栈对内核的缓冲区管理能力要求包括以下内容：方便操作可变长缓存、将数据添加在缓存头部和尾部、将数据从缓存中移走，减少这些操作所引起的数据复制。

零拷贝[8-10]虚拟接口的使用可以有效的减少数据传输的时间。在零拷贝虚拟接口之上加入动态缓冲区管理机制，将缓冲区的管理操作集中起来。

在用户空间需要创建发送接收的数据包，数据包可以通过静态与动态分配实现，但是静态分配容易造成资源浪费，而由于不断分配空间，导致动态分配 CPU 利用率高。所以，在用户空间中利用本文提出的内存管理能够解决资源浪费以及 CPU 利用率高的问题，同时内存管理也能够提高数据读写速度，进一步减少了数据传输时间。

3 性能测试分析

基于上述方法，我们对零拷贝通信系统进行了缓冲区优化。测试的硬件环境为 Intel Pentium IV1.7GHz,128MB 内存，RealTek 8139 网卡。软件环境为 RedHat Linux9.0 操作系统。网络协议栈部分作为模块加载并在应用层编写测试程序进行测试。首先服务器端建立连接，等待客户端连接；客户端与服务器建立连接，接收服务器端发送的文件 FILE，同时记录时间 TIME1，调用函数 initBlock()、inputBlock(FILE)、freebuffer()，分别进行内存管理

器的初始化，读写文件，优化用户空间的缓冲区，并记录时间 TIME2；最后计算传输时间 TIME=TIEM1-TIEM2。

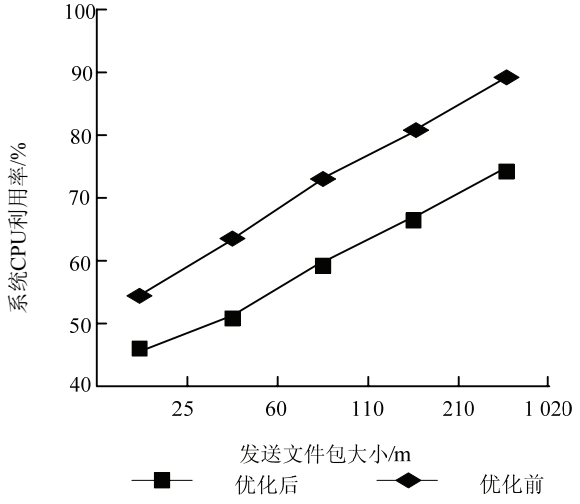

图 2 系统 CPU 利用率性能比较

Fig.2　performance comparison of System CPU

以功耗指标为标准，包括系统利用率和网络传输速率，对抽象缓冲区接口动态缓存管理机制进行网络性能测试。将优化前后的内存管理系统的 CPU 利用率进行分析比较，其性能比较结果见图 2。在高速网络环境下，网络通信系统的性能主要是由 CPU 处理能力决定，整体通信系统的稳定性由 CPU 利用率的高低说明。图 2 中实验数据显示，在一定网络速率下，优化后的内存管理机制的 CPU 功耗优化之前大大降低，适应了网络通信处理器的要求。在客户端和用户端测试中发现随着网络速率的提高，优化前的 linux 系统在数据传输中会产生丢包，而优化了内存管理的通信系统则避免了丢包现象产生。由表 1 的测试数据可以看出，优化了内存管理后的数据传输时间大大的减少了，总体上能满足用户态进程对通信系统的要求。

表 1　传输时间比较

Tab.1　time cost comparison for transmission

文件大小/M	优化前传输时间/ms	优化后传输时间/ms
25	10 083 200	1 641 600
60	17 117 760	3 395 456
110	34 696 064	4 402 944
210	99 244 160	28 298 464
1 020	124 270 215	48 779 584

经实际网络测试，结果表明在 linux 中的动态内存管理机制的应用与零拷贝技术的结合能够很好的减少 CPU 利用率和网络丢包率，提高数据传输的效率，是一种高效的协议栈内存管理机制，完全适用于通信系统，具有良好的兼容性能。

4　结　论

本文参考了 4.4BSD MBUF 以及 linux 系统中的 sk_buffer 缓冲区的管理机制，在其基础之上，又根据缓冲区的分配原理，分开考虑缓冲区的定义和内存实体的分配，提出了一种基于 linux 系统的用户态的抽象缓冲区的动态缓存管理设计。支持内存动态分配它的主要功能，避免了传统的内存分配时因确定实际的数据大小，而保守估计的预分配做法，内存资源浪费现象的减少，意味着系统资源利用率的提高。实验的性能测试数据很好的证明了使用该动态缓存管理机制提高了零拷贝模块的使用性能。在实际应用中表现出了良好的性能和可靠性，有效的提高了数据传输的效率，降低了 CPU 使用率，减少系统开销，并且可移植到其他需要缓冲区管理的通信软件中。

参考文献：

[1] 王培东,吴显伟. 一种自适应的嵌入式协议栈缓冲区管理机制[J]. 计算机应用研究,2009,26(6):2 254-2 256.

[2] Regnier G,Makineni S. TCP onloading for data center servers[J].IEEE computer magazine,2004,37(11):48-58.

[3] Druschel P,Peterson L L. Fbufs: a high-bandwidth cross-domain transfer facility[C]. Fourteenth ACM Symposium on operating systems principles,1993-12:189-202.

[4] Clark D D, Tennenhouse D L. Architectural Considerations for a New Generation of Protocols[C]. Proceedings of the SIGGOMM'90 Symposium,1990-09:200-208.

[5] 俞晓明,郭　莉. TCP/IP 协议处理中的缓冲区优化及实现[J]. 计算机工程,2006,32(8):62-63.

[6] 姚　崎,刘吉强,韩　臻,等. 面向多核处理器的 linux 网络报文缓冲区重用机制研究[J]. 通信学报,2009,30(9):102-108.

[7] 宋丽华,张晓彤,王　沁,等. 一种高效嵌入式协议栈缓冲区管理机制[J]. 小型微型计算机系统,2008,29(1):1-5.

[8] 王佰玲,方滨兴,云晓春. 零拷贝报文捕获平台的研究[J]. 计算机学报,2005,28(1):46-52.

[9] 李大斌. 并行数据库通讯组件零拷贝技术的研究与实现[D]. 哈尔滨: 哈尔滨工业大学,2005.

[10] 刘天华,陈　袅,朱宏峰,等. Linux 可加载内核模块机制的研究与应用[J]. 微计算机信息,2007,23(7-2):48-49.