有效实现内存管理的方法

潘立登　李　婷
(北京化工大学自动化系, 北京　100029)

摘　要: 分析开发过程中常遇的内存泄露和内存越界等问题,提出一种有效的内存管理方案。方案主要包括优化原有的内存管理函数和引入保存内存信息的信息链表两方面。文中还给出基于该方案的具体 C 语言实现和应用例程。

关键词: 内存;指针;分配;释放
中图分类号: TP 392

随着计算机应用需求的日益增加,应用程序的设计与开发也相应的日趋复杂,开发人员在程序实现的过程中处理的变量也大量增加,如何有效进行内存分配和释放,防止内存泄露的问题变得越来越突出。特别是服务器应用软件,需要长时间的运行,不断的处理由客户端发来的请求,如果没有有效的内存管理,每处理一次请求信息就有一定的内存泄露。这样不仅影响到服务器的性能,还可能造成整个系统的崩溃。因此,内存管理成为软件设计开发人员在设计中考虑的主要方面。本文提出了一种有效实现内存管理的方案,通过对系统本身内存管理优化,解决应用软件开发中的内存管理问题,特别是解决服务器应用软件开发的内存管理问题。同时,给出该方案 C 语言的程序实现例程。

1　基本概念

在 C 语言中,从变量存在的时间(生命周期)角度上,把变量分为静态存储变量和动态存储变量两类。静态存储变量是指在程序运行期间分配了固定存储空间的变量。而动态存储变量是指在程序运行期间根据实际需要进行动态地分配存储空间的变量。在内存中供用户使用的内存空间分为三部分:(1)程序存储区;(2)静态存储区;(3)动态存储区。程序中所用的数据分别存放在静态存储区和动态存储区中。静态存储区数据在程序的开始就分配好内存区,在整个程序执行过程中它们所占的存储单元是固定的,在程序结束时就释放,因此静态存储区数据一般为全局变量。动态存储区数据则是在程序执行过程中根据需要动态分配和动态释放的存储单元,动态存储区数据有三类:函数形参变量、局部变量和函数调用时的现场保护与返回地址。由于动态存储变量可以根据函数调用的需要,动态地分配和释放存储空间,大大提高了内存的使用效率,使得动态存储变量在程序中被广泛使用。

开发人员进行程序开发的过程使用动态存储变量时,不可避免地面对内存管理的问题。程序中动态分配的存储空间,在程序执行完毕后需要进行释放。没有释放动态分配的存储空间而造成内存泄露,是使用动态存储变量的主要问题。一般情况下,开发人员使用系统提供的内存管理基本函数,malloc、recalloc、calloc、free 等,完成动态存储变量存储空间的分配和释放。但是,当开发程序中使用动态存储变量较多和频繁使用函数调用时,就会经常发生内存管理错误,例如:(1)分配一个内存块并使用其中未经初始化的内容;(2)释放一个内存块,但继续引用其中的内容;(3)子函数中分配的内存空间在主函数出现异常中断时、或主函数对子函数返回的信息使用结束时,没有对分配的内存进行释放;(4)程序实现过程中分配的临时内存在程序结束时,没有释放临时内存。内存错误一般是不可再现的,开发人员不易在程序调试和测试阶段发现,既使花费了很多精力和时间,也无法彻底消除。

2　有效的内存管理方案

针对上述的内存管理问题,提出一种有效进行内存管理的方案。主要包括以下两方面:

收稿日期: 1999-11-13
基金项目: 信息产业部资助项目
第一作者: 男,1938 年生,教授,博士生导师

(1)优化系统提供的动态分配和释放存储空间的函数,构造新的用于动态分配和释放存储空间的函数。a)构造新的动态分配内存函数。调用系统函数动态分配存储空间,并对分配的内存做初始化。b)构造新的动态释放内存函数。调用系统函数释放存储空间,在释放之前对要释放的空间置空。使用新的动态分配和释放存储空间函数,就可以完全避免发生使用未初始化的内存空间,或使用已释放的内存空间等错误。

(2)将每一次动态分配存储空间返回的地址指针,集中保存在一个内存信息链表中。同时,该链表中还保存着每一次动态分配存储空间的大小。在程序执行过程中,这个内存链表记录了所有动态分配内存的信息,包括子函数中分配的内存空间。最后在程序结束时,将链表置空并释放,可以将程序运行过程中动态分配的存储空间全部释放。程序中为临时变量分配的内存空间、需要由主函数释放的内存空间的信息等,都可以保存在这个链表中,开发人员只需释放链表,就可以轻松地完成以前逐步释放存储空间的烦琐工作。

将上述的两方面结合使用,用封装后的内存分配、释放函数来确保内存空间的正确使用,用内存信息链表来确保无内存泄露,彻底克服了使用动态存储变量易犯的错误,实现了有效无误的内存管理。该解决方案是基于系统的内存管理之上的,是对系统内存管理的优化,因此与系统内存管理函数在使用上没有任何冲突。

3　实现设计

为了便于使用和理解,采用 C 语言实现该内存管理方案。从上述的解决方案中可以看出,用于保存内存信息的链表是整个解决方案的核心,它将零散分配的内存空间的信息集中在一起,实现了内存的集中释放和无泄露。实现设计中内存信息链表采用双向链表,这样可以独立的插入或删除指向某一个指针的内存。该链表具体信息如下:

```
struct cell {
  struct cell  *prev;
  struct cell  *next;
  int          size;
  union {
    void  *p;
    char c[1];
  } memory;
};
```

其中,prev 指向上一级链表;next 指向下一级链表;size 为本节点分配的内存的大小;联合体 memory 保存分配内存后返回的地址。新内存分配函数就是将每次内存分配返回的地址和大小存入该信息链表,最后释放时从该信息链表中找到曾经分配的内存地址和大小逐一释放。为了对使用者屏蔽该信息链表,将一个内存信息链表句柄提供给使用者。内存信息链表句柄结构如下:

```
typedef struct {
  struct cell  *alloc_list;
} _POOL_;
```

这个内存信息链表像一个内存管理的堆,可称为"内存堆"。使用者在使用"内存堆"之前,需要创建一个_POOL_类型的变量。该"内存堆"变量作为新动态分配内存的函数的接口函数,函数内部通过调用相关函数,把每次分配的内存信息保存到"内存堆"中。

4　实现程序

具体程序实现中涉及的函数如下:

```
int   insert_cell(_POOL_  *pool,struct cell  *c);
                    /*向内存堆中插入新节点*/
int   free_cell(struct cell  *c)
                    /*从内存堆中释放一个节点*/
int   LF_create_heap(LT_CTX *ctx)
                    /*分配一个指向内存堆的指针*/
int   LF_delete_heap(LT_CTX *ctx)
                    /*释放指向内存堆的指针*/
void *   LF_malloc(UINU4 size,LT_CTX ctx)
                    /*新封装的 malloc 函数*/
void *   LF_calloc(UINT4 nelem, UINT4 size,LT_CTX ctx)
                    /*新封装的 calloc 函数*/
```

```
int insert_cell(_POOL_  *pool,struct cell  *c)
```

函数功能:将内存信息节点 c 插入指定的内存堆中,用于给一个新指针分配内存时,将其相关信息插入到指定的内存堆中。参数:pool 是指向内存堆的指针,c 是要插入的内存信息链表指针。返回值:返回一整型数值,成功为 0,失败为小于 0。

```
int insert_cell(_POOL_  *pool,struct cell  *c)
```

```
{
  if (pool)- > alloc _ list ! = 0{
    c- > next = pool)- > alloc _ list ;
    pool)- > alloc _ list- > prev = c ;
  }
  pool)- > alloc _ list = c ;
  return 0 ;
}
int free _ cell (struct cell *c)
```

函数功能:释放内存堆中的一个节点。**参数**:c 是要释放的内存信息链表。**返回值**:返回一整型数值,成功为 0,失败为小于 0。

```
int free _ cell (struct cell *c)
{
  (void) memset ((void *) c ,0 ,c- > size)
  (void) free (c) ;
  return 0 ;
}
```

以上两个函数是在新的内存分配函数中使用的内部函数,使用者可以不必了解其具体实现。在有关内存堆的操作中,为了使用方便定义一个 void * 指针,保存内存堆的指针,具体使用见例程。

```
typedef void * L T _ CTX ;
int L F _ create _ heap (L T _ CTX *ctx)
```

函数功能:分配一个内存堆,用于存放内存信息链表。**参数**:ctx 是要创建的内存堆的指针。**返回值**:返回一整型数值,成功为 0,失败为小于 0。

```
int L F _ create _ heap (L T _ CTX *ctx)
{
  if (ctx = = 0) return - 1 ;
   *ctx = calloc (1 ,sizeof (_ POOL _)) ;
  if ( *ctx = = 0) return - 2 ;
  return 0 ;}

int L F _ delete _ heap (L T _ CTX *ctx)
```

函数功能:释放一个内存堆,释放堆中的所有内存信息链表。**参数**:ctx 是要释放的内存堆的指针。**返回值**:返回一整型数值,成功为 0,失败为小于 0

```
itn L F _ delete _ heap (L T _ CTX *ctx)
{
  struct cell *c = 0 ;
  struct cell *hold _ next = 0 ;
  int status = 0 ;
  if (ctx = = 0) return - 1 ;
  if ( *clx ! = 0)
    c = ( *ctx)- > alloc _ list ;
    while (c ! = 0) {
      hold _ next = c- > next ;
      if (free _ cell (c) ! = 0) status = - 1 ;
      c = hold _ next ;
    }
    memset ( *ctx ,0 ,sizeof (_ POOL _)) ;
    (void) free ( *ctx) ;
  }
   *ctx = 0 ;
  if (stats ! = 0) return - 1 ;
  return 0 ;
}

void *L F _ malloc (U IN T4 size ,L T _ CTX ctx)
```

函数功能:在 ctx 内存堆中分配 size 大小的内存块并块初始化。**参数**:size 是要分配的内存空间的大小;ctx 是要存放分配的内存信息的内存堆。**返回值**:返回值是一个 void 类型指针,表示所分配内存的地址。

```
void *L F _ malloc (U IN T4 size ,L T _ CTX ctx)
{
  _ POOL _  *pool = (_ POOL _ *) ctx ;
  struct cell *c = 0 ;
  void  *p = 0 ;
  int  len ;
  len = size + sizeof ( *c) ;
  c = (struct cell *) malloc (len) ;
  i (c = = 0) return ( - 1) ;
  c- > prev = c- > next = 0 ;
  c- > size = len ;
  p = (void *) ((cell)- > memory. c) ;
  memset (p ,0 ,size) ;
  insert _ cell (pool ,c) ;
  return p ;
}

void *L F _ calloc (U IN T4 nelem ,U IN T4 size ,L T _ CTX ctx)
```

函数功能:在 ctx 内存堆中分配 nelem 个 size 大小的内存块并初始化。**参数**:nelem 是要分配的元素个

数;size 是要分配元素的大小;ctx 是要存放分配的内存信息的内存堆。返回值:返回值是一个 void 类型指针,表示所分配内存的地址。

```
void *LF_calloc(UINT4 nelem,UINT4 size,LT_CTX ctx)
{
  _POOL_    *pool = (_POOL_*)ctx;
  struct cell    *c = 0;
  void    *p = 0;
  int    len;
  len = (size *nelem) + sizeof( *c);
  c = (struct cell *)calloc(len,1);
  if(c = =0)return(-1);
  c->size = len;
  p = (void *)((cell)->cell)->memory.c);
  memset(p,0,size *nelem);
  insert_cell(pool,c);
  return p;
}
```

上述程序在 IBM 的 AIX 4.3.2 版本上和 Windows 98 操作系统均可正确运行。

5　使用例程

开发人员实际需要调用的函数,仅是上述函数中的后四个接口函数 LF_create_heap、LF_delete_heap、LF_malloc、LF_calloc。以函数形参和局部两种动态存储变量为例说明如何使用这些函数:

(1)动态存储是函数形参变量时,在主函数中创建内存堆,并作为参数传入子函数,子函数在该内存堆中,为形参动态分配存储空间,当主函数中需要释放为这些形参变量分配的内存空间时,只需释放该内存堆即可;

(2)动态存储是局部变量时,在该函数内部创建内存堆,在该内存堆中,为局部变量动态分配存储空间,在函数结束时释放该内存堆。

将基于"内存堆"的内存管理程序使用过程总结如图 1,同时,为了便于读者理解使用给出一个简单例程,在主函数中调用一个子函数完成字符串拷贝。

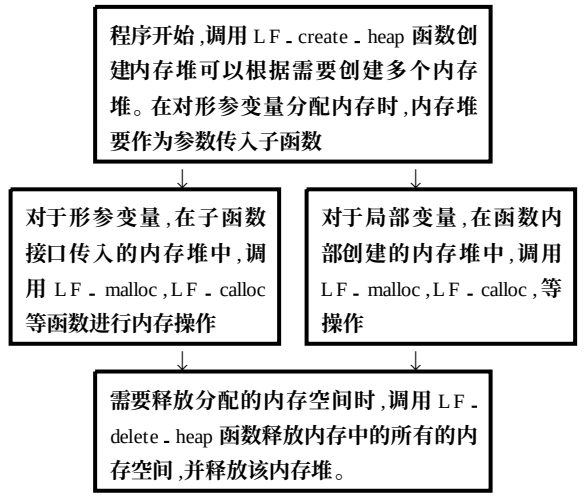

图 1　新内存管理函数的使用流图

Fig. 1　Flow diagram of new memory management function utilization

```
int main()      /*主函数*/
{
  char      *aa;
  LT_CTX   heap;
  int      ret;
  if(LF_create_heap(&heap) ! =0)return -1;
  ret = func(&aa,heap);
  if(ret ! =0){
  LF_delete_heap(&heap);
  return -1;
  }
  printf("%s\n",aa);
  LF_delete_heap(&heap);
  return 0;
}
int func(char **str,LT_CTX heap)    /*子函数*/
{
  char  *bb;
  LT_CTX   tempheap;
  if(LF_create_heap(&tempheap) ! =0) return -1;
  bb = (char *)LF_malloc(20,tempheap); /*局部变量处理*/
  strcpy(bb,"made in china");
   *str = (char *)LF_malloc(20,heap);   /*形参变量处理*/
  if( *str = =NULL)return -1;
  strcpy( *aa,bb);
  LF_delete_heap(&tempheap);
  return 0;
}
```

实际上,使用上述函数进行内存分配,除了要创建内存堆以外,用于内存操作的函数与系统内存操作函数使用上没有区别,程序代码量基本没有增加,但却实现了程序的内存的有效管理,是一劳永逸的。

参 考 文 献

[1] Steve M. 编程精粹——Microsoft 编写优质无错 C 程序秘诀. 北京:电子工业出版社,1994

[2] Jeffrey R. Windows NT 高级编程技术. 北京:清华大学出版社,1994

[3] 埃利斯·霍罗维茨,萨尔塔·萨尼,狄尼斯·梅坦. 用 C++描述数据结构. 北京:国防工业出版社,1997

[4] 山下浩,黑羽裕章,黑岩健太郎. C++程序设计. 北京:科学出版社,1995

[5] 程文斌,王一行. Microsoft C/C++和 Visual C/C++库函数和 MFC 库类详解. 北京:北京航空航天大学出版社,1995

[6] 谭浩强. C 程序设计. 北京:清华大学出版社,1991

An efficient method of memory managerment

PAN Li-deng LI Ting

(Department of Chemical Automation, Beijing University of Chemical Technology, Beijing 100029, China)

Abstract: The paper analyses memory leakage and out-of-range problems in software development, and provides a new scheme to effectively manage memory. This scheme include optimizing the system functions of memory management and introducing the information linked list for keeping information about memory meanagement. The paper also gives a program of this scheme in the C language and some routines.

Key words: memory; pointer; allocation; freeing

(上接第 97 页)

理。坚膜剂的种类有很多,硼酸坚膜是一种较为简便,有效的方法。因为硼酸结构中的 B 含有空轨道,它可与 PVA 上的羟基发生配位反应,使 PVA 分子内和分子间通过其与羟基反应而进行交联,提高版膜的耐磨性。坚膜处理后的网版,其 12 min 磨擦余量一般都提高 3 %以上(见图 4)。

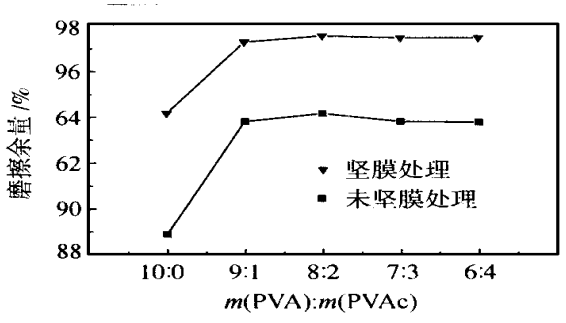

图 4 坚膜处理对丝印网版磨擦特性的影响

Fig. 4 Effect of the hardening agent on the wear curve

以上几个方面是提高网版动态耐水性的主要因素。除此之外,网版的涂布工艺,胶膜的厚度,曝光、显影的条件,助剂的选择,网版的物理强化等,均对网版的动态耐水性有一定的影响。

参 考 文 献

[1] 黄毓礼,侯海薏. 光聚合丝印胶的感光及成像性能. 北京化工大学学报,2000,27(2):35～38

[2] 邓莉莉. 非银盐感光材料. 北京:印刷工业出版社,1994. 226

Waterproof property of the photopolymer emulsion for screen printing

HOU Hai-yi HUANG Yu-li

(College of Materials Science and Engineering, Beijing University of Chemical technology, Beijing 100029, China)

Abstract: In this article, the effects of the monomer, film former, inhibitor, and heating on the waterproof property of the screen printing plate were studied in detail by wearing the wet plate and treating the plate with the hardening agent. The results indicated that the suitable monomers and film former can improve this property remarkably, and the effect of heating is not very important.

Key words: screen printing; photopolymerization; photosensitive emulsion