关于内存泄漏的检测与防范

林丽芬

(福建信息职业技术学院,福建 福州 350003)

摘 要:动态分配、回收内存是 C/C++编程语言一个最强的特点,但是最强的同时也可能是最弱的,在内存处理出错的地方通常就是 BUGS 产生的地方。一个最敏感和难检测的 BUGS 就是内存泄漏——没有把前边分配的内存成功释放,一个小的内存泄漏可能不会引起人的注意,但是程序泄漏大块内存,将可能引起复杂的内存耗尽错误。

关键词:内存泄漏;动态分配;释放内存

中图分类号: TP311 **文献标识码**: A **文章编号**: 1673-9884(2007)07-114-04

1 引言

在程序设计中,经常会碰到一个重要的问题,那就是内存泄漏,特别是在一个大型的服务器系统中,若存在内存泄漏,将会影响到系统的长期稳定运行。所以,内存泄漏困扰着应用软件的顺利开发与正常使用,下面将对这个方面的内容展开论述。

2 操作系统的管理

在操作系统中,一般提供给用户两种的内存使用方式:

2.1 栈

在系统管理中,由操作系统负责栈内存的分配,提交缺页处理。在内存不足的情况下,栈会超出分配给它的限制,就会发生缺页异常,该异常并不会进入 debug,系统将会判断该异常是由于栈空间超出了初始分配的大小,便向系统申请更多的内存,而用户程序并不会察觉此现象,仍在发生异常的地方继续运行,导致用户程序继续使用动态的内存空间。

2.2 用户堆

在应用程序中,由用户程序向系统申请所需内存,使用结束后,返回给系统,在此过程中,由于程序的复杂和庞大,使得已经分配了的内存没有及时由系统回收,这样很容易导致可用的内存愈来愈小,严重时系统会出现内存不足,而发生泄漏的现象。

3 内存泄漏

3.1 内存泄漏的定义

内存泄漏常指堆内存的泄漏[1]。堆内存是指程序从堆中分配的、大小任意的并且在使用结束后必须释放的内存。当我们使用 C/C++、delphi或 java等开发工具编写应用程序时,一般在构造函数中使用 malloc、realloc、new等函数从堆中分配到一块内存,该内存空间使用结束后,程序必须相应地调用 free或 delete等函数释放该内存块,以便操作系统重新分配与再次使用此内存。若在使用结束后,没有及时在程序的全部执行路径或析构函数中释放此内存块,并且无合适的指针指向该块内存,使得该部分内存失去重用性,这就称为内存泄漏。

3.2 内存泄露的危害

从应用程序的角度来看,作为一般的用户,根本感觉不到内存泄漏的存在,因为一次性内存泄漏本身并没有什么危害,它不会堆积,当程序结束后内存会自动释放,内存泄漏自行消失。但是随着内存泄漏的堆积,最终将消耗系统所有的内存,影响计算机的正常使用。对于服务器而言,如果出现内存泄漏,则将导致服务器系统的崩溃。

3.3 几种常见的内存泄漏

内存泄漏对于程序员来讲,是个容易忽略的关键问题。它常出现在静态全局变量、动态分配内存空间等处。

收稿日期: 2007-04-30

作者简介:林丽芬(1967-),女,福建莆田人,福建信息职业技术学院软件工程系高级讲师。

3.3.1 在使用动态分配内存空间中的内存泄漏

例如：

```
void MyFt (int n)
{
   char *   p = new char[nSize]; //定义一个指针变量并分配一块内存供 P使用
    if(! GetStringFrom(p,n)){
     MessageBox("Error");
      return;
    }
    …
    delete p; //释放 P所指向的内存块
}
```

这是一个简单而又典型的内存泄漏示例，当函数 GetStringFrom()调用错误时，函数 MyFt结束而指针 p 指向的内存却没有被释放，此时便出现了内存泄漏。在程序段入口处分配内存，在出口处释放内存，但是 c 函数可以在任何地方退出，所以一旦有某个出口处没有释放应该释放的内存，就很容易发生内存泄漏[2]。

但简单的释放有时也会造成内存泄漏，例如：

```
PDataItem = ^TDataItem;
TDataItem = Record
   ……
   stuData: String;
end;
lstDataItems: TList; pItem: PDataItem;
lstDataItems: = TList.Create;
new(pItem);
lstDataItems.Add(pItem);
for i: = 0 to lstDataItems.Count - 1 do
  Dispose(lstDataItems.Items[i]);
```

以上代码中，Dispose(lstDataItems.Items[i]) 语句发生了内存泄漏，原因是 lstDataItems.Items[i]返回的是一个 Pointer类型的数据，Dispose函数对 Pointer 类型数据释放内存时，只会释放该指针所指向的内存块。

若上面的代码中，Pointer指向的是一个结构体 TDataItem，那么，该结构体本身所占的内存块就被释放了。但是，如果该结构体内部的数据成员为 String 类型、PChar或对象时，这些数据成员自身所占用的内存得不到释放，就会引起内存泄漏。由此可见，在释放一个指针的内存时，有可能会出现内存泄漏。

3.3.2 在使用局部指针变量或使用静态指针变量中的内存泄漏：

请看以下这段代码

```
int main()
{
int * pi;
pi = new int[100];
return 0;
}
```

这是常见的使用局部变量时出现的内存泄漏，若使用 valgrind工具检查，报告会显示“definitely lost”，即内存泄漏 (memory leak)了。

下面的例子虽然是用静态指针，同样会造成内存泄漏。

```
int main()
{
static int * pi = 0;
for (int i=0; i<10; i++)
     pi = new int;
return 0;
}
```

程序退出后，分配的 10个 int只有最后一个是 reachable的，而前面 9个 int则泄漏了。

4 内存泄漏的检测

所谓泄漏检测，就是通过一定的手段检测所运行的系统中是否存在内存泄漏，及时发现错误并纠正错误的一种方法。

4.1 用户堆泄漏的检测

用户堆的程序是以进程的方式完成一定的功能，当系统起用一个进程时，会分配给该进程所需要的资源，当进程退出前将所有的资源回收，这里所讨论的泄漏便是发生在系统分配资源后和回收前这两者之间的事情。

一般系统对用户内存的分配，是通过一个方法来实现的，比如假定__malloc()是基本的分配函数，那么 C/C++的 malloc()方法和 new关键字方法都是在它的基础上实现的。同样认为系统中存在__free()方法，在它的基础上才实现了 C/C++的 free()方法和 delete()方法。这样系统在分配所有内存和释放所有内存时，malloc都会返回一个已分配的内存首地

址,释放时凭该首地址将该内存释放。

现在定义如下结构体:

```
Struct memnode{
int    index;
addr_t   addr;
};
```

该结构体是一个链表的节点,该链表代表用户进程在运行过程中分配内存记录。每分配一次系统就产生一个节点,index成员表示该次分配是第几次分配,是顺序号。Addr成员是内存的首地址。每调用一次__malloc()函数,就表明有一次分配,每调用一次__free()方法,就表明有一次释放,在链表中根据内存首地址,找到相应的节点,给予删除。等整个进程退出时,查看链表,如果还有节点,那这些节点就是存在内存泄漏的节点,有几处这样的节点,就存在几处内存的泄漏。通过以上方法可以完成用户程序的内存检测,从而防止内存泄漏。

这里的链表节点可以包含更多的信息,让用户很容易找到问题的所在,比如分配内存的大小、线程 id等信息,当然链表也可以改为哈希表,或平衡二差树以加快查找。

4.2内核堆泄漏的检测

内核如果存在内存泄漏,那么检验方法要复杂一些,因为内核是一直在系统的整个工作周期中。它所拥有的资源,通常不需要释放。这样就很难断定"哪些是系统还会使用到的?哪些是可以释放了的?"。但可以换个角度,重新审视这个问题,内存泄漏在什么情况下会造成巨大的危害呢?那就是程序在运行时内存被不断的需求,只增不减,同样的程序运行次数愈多,内存的消耗就愈大。也可以说和程序的运行次数成正比。只要能避免对内存不合理的反复需求,就说明控制住了内存泄漏。

解决此问题的方法如下:

在内核的关键部位,设置检查点 checkpoint,该检查点记录内核内存的使用情况信息,比如内核的缓存有多大,使用了多少页,实际物理内存有多少,释放了多少等数据。刚开始可能系统对内存的需求没有达到稳态,但是运行同样的程序多次后,观察同一个checkpoint的信息是否有内存正比增加,如果有则认为有内存泄漏,需要加以处理。

4.3动态内存泄漏信息的检测方法

内存泄漏的检测能够在程序结束时,显示出在程序运行过程中已经在堆上分配但是没有释放的内存分配信息,由此可以找到程序中"显式"的内存泄漏点并加以改正。此外,可能还存在另外两种危害性更大的"隐式"内存泄漏,其表现就是在程序退出时,没有任何显示内存泄漏的现象,但在程序运行过程中,内存占用量却不断增加,致使整个系统崩溃。其一是程序的一个线程不断分配内存,并将指向内存的指针保存在一个数据存储中(如 list),但是在程序运行过程中,一直没有任何线程进行内存释放。当程序退出的时候,该数据存储中的指针值所指向的内存块被依次释放。其二是程序的 N个线程进行内存分配,并将指针传递给一个数据存储,由 M个线程从数据存储进行数据处理和内存释放。由于 N远大于 M,或者 M个线程数据处理的时间过长,导致内存分配的速度远大于内存被释放的速度,但是在程序退出的时候,数据存储中的指针值所指向的内存块被依次释放。而这又是很不容易将这种问题找出来,程序可能连续运行几个十几个小时没有问题,从而通过了不严密的系统测试。但是如果在实际环境中运行时,系统将不定时的崩溃,而且崩溃的原因从 log和程序表象上都查不出。

为了解决这个问题,可增加一个动态检测模块MemSnapShot,用于在程序运行过程中,每隔一定的时间间隔就对程序当前的内存使用情况和内存分配情况进行统计,以使用户能够对程序的动态内存分配状况进行监视。

当用户使用 MemSnapShot监视一个运行中的进程时,被监视进程的内存子系统将把内存分配和释放的信息实时传送给 MemSnapShot。MemSnapShot则每隔一定的时间间隔就对所接收到的信息进行统计,计算该进程总的内存使用量,同时调用 new方法进行内存分配,以它的文件名和行号作为索引值,计算每个内存分配动作所分配而未释放的内存总量。如果在连续多个时间间隔的统计结果中,某文件的某行所分配的内存总量不断增长而始终没有到达一个平衡点甚至回落,那它一定是我们上面所说到的两种问题之一。

5用户内存泄漏的防范与调试

首先,要养成良好的编程规范。如在上例中,把代码改为:

```
for i: = 0 to lstDataItems Count - 1 do
```

```
begin
  pItem := lstDataItems.Items[i];
  Dispose(pItem);
end;
```

则可以有效地释放内存而避免内存泄漏[3]。

当然,如果结构体中包含有其它指针或其类似的数据成员,则必须在释放该结构指针之前先释放这些成员的私有内存,否则也会造成内存泄漏。

其次,一旦泄漏发生,要在已发现内存泄漏的基础上如何找到内存泄漏的现场。如在上面结构体 Struct memnode中,成员 index记录的是内存分配的先后顺序,在假定多次运行内存分配的顺序不会发生颠倒的情况下,就可轻松找到是哪里分配的内存泄漏了,并得到了控制,下面看如何做到这一点。

在第一遍运行程序,知道了第几个 index的节点出现了内存泄漏,第二次在运行同样的程序,这时当在__malloc()函数中发生的内存分配次数与上次记录的 index值一致时,则认为就是这次发生的内存泄漏,设置断点,就可轻松找到用户程序内存的真实泄漏处。

另外,在 Windows平台下,检测内存泄漏的工具也是调试与防范内存泄漏的常用方法,常用的检测工具一般有三种:MSC - Runtime Library通过函数调用栈,内建的检测功能;外挂式的检测工具,诸如, Purify, BoundsChecker等;利用 Windows NT自带的 Performance Monitor。这三种工具各有优缺点,MSC - Runtime Library功能上较之外挂式的工具要弱; Performance Monitor虽然无法标示出发生问题的代码,但是它能检测出隐式的内存泄漏的存在。

6总　结

内存是一个系统程序员应该关注的核心,内存泄漏会留下很多隐患,导致内存泄漏的原因很多:忘记释放内存;构造函数失败;存在内存泄漏的析构函数;多个返回语句;使用错误形式的 delete等等。内存泄漏往往会导致系统资源的泄漏,因为动态分配内存往往不仅仅代表一块存储区域,还代表了某些类型的系统资源,如文件、窗口、设备上下文、GDI对象等。编写应用程序时,要养成一个良好的编程习惯,注意对于动态分配的内存不用时要及时释放,合理的使用异常处理方法,在技术上避免内存泄漏的发生,在调试程序时,一旦发现内存泄漏,可以使用相应的监测工具找到泄漏处并消除。

参考文献:

[1](美) R. ALEXANDER, G. BENSLEY. 王峰,史金虎译. 内存与性能优化 [M]. 北京:中国电力出版社, 2005.

[2] Stanley B. Lippman. 候捷译. 深度探索 C + +对象模型 [M]. 武汉:华中科技大学出版社, 2001.

[3]刘艺. 面向对象编程思想 [M]. 北京:机械工业出版社, 2003.

[4]W. Richard Stevens. 尤晋元等译. UNIX环境高级编程 [M]. 北京:机械工业出版社, 2000.

The Detection and Prevention of Memory Leakage

L N Li - fen

(Fujian Polytechnic of Information Technology, Fuzhou 350003, China)

Abstract: Dynamic allocation and memory retrieve are two of the strong features in the C/C + + program. However while being the two strongest points, they could also be the weakness. Usually where errors are found during the memory treatment, this proves to be where the bugs have appeared. One of the most sensitive and difficult bugs to detect is memory leakage, where the previously allocated memory has not been successfully released. We may not notice a small - sized memory leakage, but for a large - sized memory leakage, it will cause some complicated errors which may lead to the memory exhaustion.

key wordS: memory leakage; dynamic allocation; memory retrieve