一种有效的动态内存泄漏检测技术的研究与实现

张晓明 刘建君 李树江
(沈阳工业大学信息科学与工程学院 辽宁 沈阳 110870)

摘 要 内存泄漏故障是一个程序员所必须关心的问题之一。通过对内存泄漏及其相关检测技术的研究,提出面向类型的动态内存泄漏检测的概念,使泄漏检测具有较高的实时性,大大降低由于内存泄漏检测而导致 CPU 占用时间的急剧变化值,并在 Linux 下验证了算法的真实有效性。

关键词 内存泄漏 内存分配 检测技术

中图分类号 TP306 **文献标识码** A

RESEARCH ON AN EFFICIENT DYNAMIC MEMORY LEAK DETECTION TECHNOLOGY AND ITS IMPLEMENTATION

Zhang Xiaoming Liu Jianjun Li Shujiang
(*College of Information Science and Engineering, Shenyang University of Technology, Shenyang 110870, Liaoning, China*)

Abstract Memory leak failure is one of the problems that a programmer must be concerned about. In this paper, based on the study of memory leak and the related detection, the notion of type-oriented dynamic memory leak detection is proposed, which makes the leak detection technology be highly timely, and greatly reduces the sharp change value of CPU occupancy time caused by the memory leak detection. The reality and validity of this algorithm is verified in Linux operating system.

Keywords Memory leak Memory allocation Detection

0 引 言

随着科学技术的发展,人类逐渐进入信息技术社会。而计算机作为信息社会的主力军,在世界经济和社会发展中起着无法替代的作用。计算机科学的重要研究领域——内存管理,因其在计算机中的关键作用,受到全世界计算机工作者的普遍关注。一旦程序员疏忽,对内存管理错误,就会引起一些内存故障。因此,检测内存故障便成了一个无法避免的问题。检测内存故障是非常困难的,也难以准确识别出程序中的故障源[1]。内存泄漏便是其中的一种。

所谓内存泄漏是指程序已在堆里动态分配了一块内存,在使用完毕后,由于某种原因一直未释放或者因丢失了访问路径而无法释放,这样的内存就是泄漏内存。这种内存故障在数量较少时,对系统的正常运行没有明显的影响,但随着泄漏逐渐积累到一定程度时,将会导致程序运行速度减慢,甚至系统崩溃。如何有效地进行内存分配和释放,防止内存泄漏问题变得越来越突出。因此,使用一款内存泄漏检测工具来检查泄漏故障,便成为程序调试阶段所必不可少的。

1 相关技术

为了适应各种应用环境,许多公司及个人开发了不同的内存检测工具。这些检测工具根据是否要执行代码,可以分为静态工具和动态工具。静态工具效率高,但存在较高的漏报率和误报率,而动态工具主要是对目标代码进行插装,准确率高。

LCLink[2]是通过对源代码及添加到源代码中特定格式的注释进行静态分析的程序理解和检错工具。Purify 在目标代码中插入特殊的检查指令来实现对内存的检测。Insure ++[3]是运行时检测工具,能检测 C/C ++ 中的编程和运行时错误,检验静态(全局)和堆栈以及动态分配内存的操作的有效性,但对发现第三方库函数中的代码泄漏无能为力。文献[4]阐述了一种通过静态分析指针映射关系的方法来检测内存泄漏故障。此方法与运行时检测相比,其内存故障的代价要小得多。本文基于内存泄漏检测的有效性以及程序执行的高效性,提出了一种有效的内存泄漏检测算法,在一定程度上提高了泄漏检测的实时性,大大提升了检测程序的执行效率。

2 程序插装

程序插装的概念是由 J. C. Huang 教授首次提出的[5],也称软件打点技术。其目的是通过探针来捕获当前程序状态。插桩技术是目前大多数软件测试采用的关键技术。

程序插装根据插桩的代码形式分为两种:源代码插装和目标代码插装。源代码插装主要是向被测程序中植入相应类型的探针函数,通过运行此被测程序而获取程序运行的动态数据。目标代码插装实现较为复杂,它的主要工作是围绕断点而进行,

收稿日期:2010 - 09 - 16。国家自然科学基金项目(60772055)。张晓明,副教授,主研领域:基于 Linux 的嵌入式软件。

其前提是对目标代码进行必要的分析以确定插装的地点和内容。

3　内存泄漏的原因及分类

导致内存泄漏的情况有很多,其原因实质上是申请的内存没有释放。例如程序员编写程序时对内存管理考虑不周导致的内存泄漏;没有在程序的全部执行路径中释放内存,函数返回值的不处理或不适当处理;内存指针被重新赋值导致原内存泄漏等。一个程序在崩溃之前可运行的时间越长,则导致崩溃的原因与内存泄漏的关系越大。

图 1 给出程序中内存泄漏的不同表现形式,其中②、③为泄漏单元。

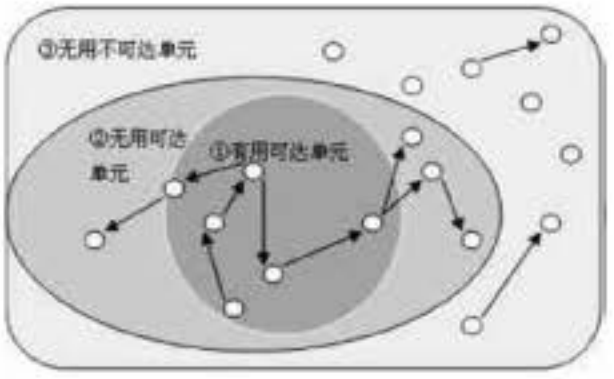

图 1　内存泄漏的不同表现形式

内存泄漏是由于用户向系统动态申请的内存没有得到及时、正确的释放而导致的。因其发生泄漏原因的不同有显示与隐式之分。显式内存泄漏是指分配的内存没有指针指向它或是在程序的运行过程中指向该内存的指针被赋予其他值或被销毁,而使得该内存块丢失。隐式内存泄漏是指系统分配的内存在程序的运行过程中并没有及时调用相关的函数进行释放,只能在程序运行结束后才统一被系统回收,这样,在系统运行过程中,内存使用就会不断累加,甚至耗尽,而导致系统崩溃。

4　内存泄漏检测技术设计与实现

4.1　内存泄漏检测算法

由于在 Linux 运行环境下的自由软件中,通常的内存泄漏调试工具 dmalloc、memwatch 等,大都是在程序运行时记录下内存的操作信息,等到程序运行结束后,才统一检测并释放内存。这样,不但提升了由于内存泄漏检测而导致 CPU 占用时间的急剧变化值,而且更缺少实时性。这给软件的测试,尤其是长时间运行软件(如服务器软件)的测试带来了极大的不便。这里我们采用面向类型的内存泄漏检测算法来实现对程序中分配的 UB(User Buffer)块层次性地进行内存泄漏检测。

内存泄漏检测技术原理(单链表实现 UB 分类算法)如图 2 所示。

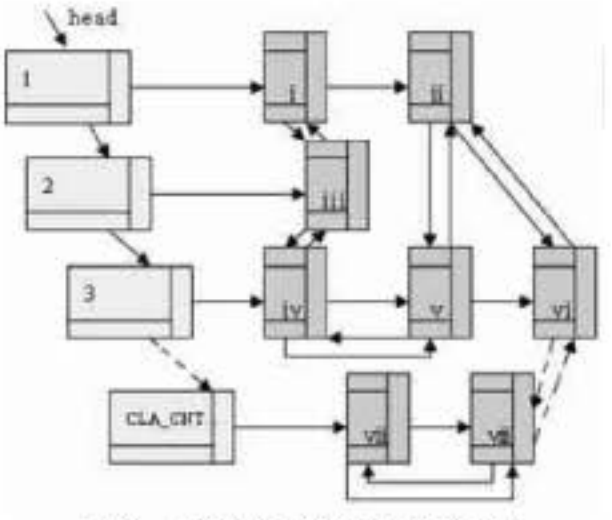

图 2　内存泄漏检测技术原理图

系统构造一个单向链表(head 为链表头指针),用于存储用户申请分配的内存块类型信息,每个结点均指向该类的内存块信息结点链,链表长度 CLA_CNT 随着 UB 块类型数量的增加而递增。假设系统中需要使用的最大内存类型数为 N 个,则设置 N 个内存块类型信息链元素。用户每动态申请一个内存块时,系统就捕捉该类 UB 块的泄漏检测积聚值 LDAV(Leak detection of the accumulation value),当该积聚值超过系统规定的泄漏检测限度值 LVLD(Limit values of Leak detection)时,便自动调用内存泄漏检测程序对该类内存进行泄漏检测,同时将 LDAV 清零。

4.2　内存泄漏检测算法数据结构

在本文所述的内存泄漏检测算法中,采用单向链表和双向链表来存储信息结点,内存块描述信息结点由双向链表存储,内存类型结点存放在类型描述信息单向链表中。这样就构成了一个按 UB 块分类的彼此相联系的层次有序队列,而这个类型信息链表的每个结点又是一个单向的有序队列。内存块类型信息结点结构描述如下:

```
Typedef struct _class_node_block_
{
Size_t size;                                    //该类内存块大小
Int ClaNum;                                     //系统中该类的编号
COUNT MaCnt;                                    //该类 UB 块分配次数
COUNT NCnt;                                     //该类中的元素个数
Struct _Dyna_Mem_Info_Block_ * DyInfoPoin;
                                                //该类指向 UB 块信息结点指针
Struct _class_node_block_ * Next;               //指向下一个类结点的指针
} ClaNodeBlo;
```

内存信息块指向指针域 MemPoinZoneBlo 结点用于存储指向本内存块指针。其数据结构描述如下:

```
Typedef struct _Mem_pointer_zone_Block_
{
Void * * MemPoinAddr;                           //指向本 UB 块的指针
Struct _Mem_pointer_zone_Block_ * next;         //指向下一个结点指针
} MemPoinZoneBlo;
```

内存信息块结构 DynaMemInfoBlo 记录已分配 UB 块的详细信息,在系统为用户分配一个 UB 块的同时,也为其分配一个 DynaMemInfoBlo 结点。根据用户申请分配的时间,将 DynaMemInfoBlo 结点安放到双向链表的相应位置,同时与相应的 ClaNodeBlo 结点相挂接。内存信息块结构如下所示:

```
Typedef struct _Base_Info_Block_
{
Size_t size;
Char * file;                                    //申请 UB 块所在的文件
Char * func;                                    //申请 UB 块所在的函数
Unsigned line;                                  //申请 UB 块所在的行号
} BaInfoBlo;                                    //基础信息结构单元
Typedef struct _Dyna_Mem_Info_Block_
{
Void * addr;                                    //用户申请 UB 块的起始地址
BaInfoBlo DyBaInfo;                             //UB 块的基础信息
COUNT index;                                    //申请 UB 块的序列
Struct _Mem_pointer_zone_Block_ * MemPoinZone;
                                                //本内存信息块结点的指向指针域指针
Struct _Dyna_Mem_Info_Block_ * NodePoinNext;
                                                //指向此 UB 类的下一个内存信息块结点
```

```
Union lf {
Struct {struct _Dyna_Mem_Info_Block_ *prev, *next ;} list;
                                    //内存信息块结点的前指针及后指针
Struct {char *file; char *func; unsigned line ;} free;
                                    //本 UB 块的基本信息存储器
} u;
} DynaMemInfoBlo;
```

4.3 内存泄漏检测技术实现

该内存检测算法实现的本质就是:为每个已分配的 UB 块配备一个指向指针域指针,并把每个 UB 块按类型分类,加入相应的类。内存泄漏检测启动时,检测该类结点的指向指针域指针,如果有结点的指针为空,并且指针存储器中也无指向该 UB 块的指针,则此内存块泄漏。

内存泄漏通常较难发现,定位比较困难。因此,为了能准确地定位内存泄漏的位置,本文采用预编译阶段代码插装技术,利用宏定义功能,通过编译,把 LibC 的内存管理函数调用替换为本检测系统中重新实现的相应管理函数,同时还利用三个预定义宏_FILE_、_LINE_和_FUNC_,来获得动态分配函数所在的源文件名、行号和函数名,并作为参数传入到重新实现的内存管理函数中。

为了能动态、深入地跟踪内存的分配情况,本检测技术重新实现的动态分配函数,除了分配用户申请的内存外,还要生成内存信息块及相应的类信息块,用以填充相关的内存信息,并把内存信息块插入到内存信息结点双向链表中,且与相应的类信息结点链相挂连,以对该内存块进行跟踪。其实现流程如图 3 所示。

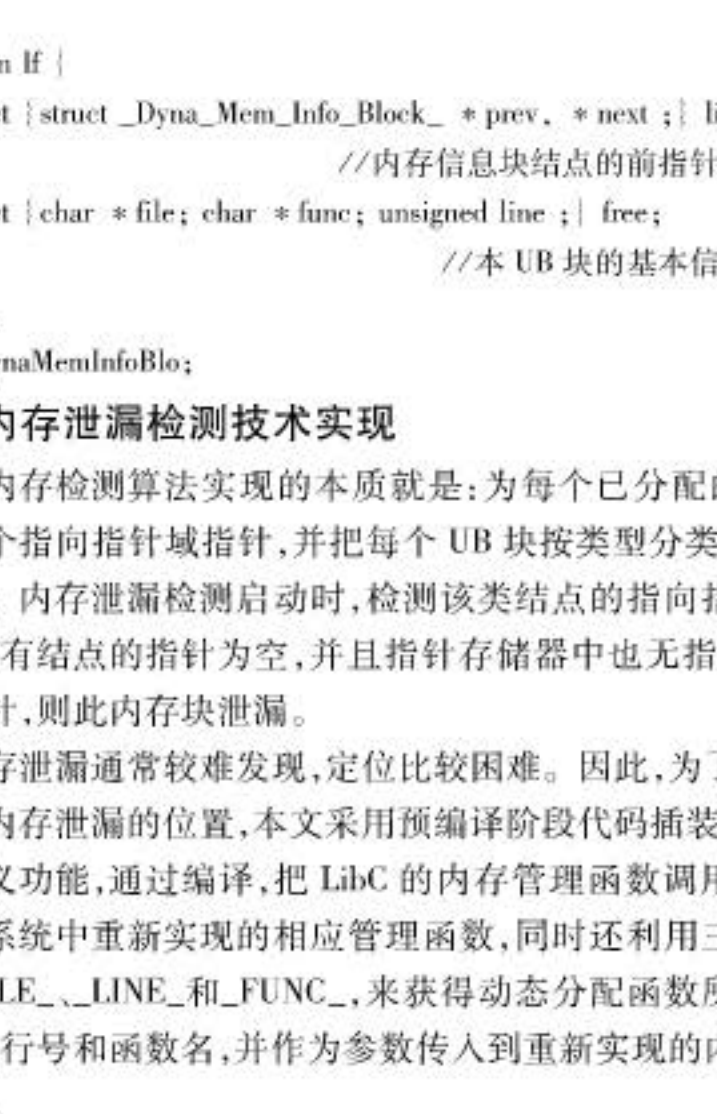

图 3 动态分配函数实现流程图

系统周期性地调用内存泄漏检测程序,按类分层扫描内存信息结点双向链表是该算法的核心;LDAV 对每类 UB 块的分配次数进行计数,并存储在各类相应 ClaNodeBlo 结点的 MaCnt 元素中。其实现流程见图 4 所示。

图 4 内存泄漏检测程序执行流程图

重新实现的内存归还函数除了释放用户申请的内存外,还要从内存信息结点双向链表中移除相应的结点,减去相应的内存空间,并从类信息结点链中去除。

此外,系统还设置了一个用户备用信息栈,用于暂存最近正常释放的内存结点单元。当正常释放内存结点单元时,将该结点的基本信息存入到本结点的基本信息存储器中,并将其复制到用户备用信息栈中,用于日后备查。

在整个检测程序的运行过程中,系统定次、自动地对申请的内存进行层次性地检测,并将检测出的内存泄漏故障记录到跟踪日志文件中,方便日后分析程序中的内存故障。

5 实 例

这里给出了具体实例,在 Linux 平台下,验证本内存泄漏检测算法的真实有效性。为了节省篇幅,这里删除了实例中的头文件部分。

```
Int func01 (STRING p1)
{
  p1 = malloc (MALLOC_SIZE); Return 0;
}
STRING func02 ()
{
Return malloc (MALLOC_SIZE);
}
Void func03 (STRING p1)
{
Malloc (MALLOC_SIZE); func01 (p1);
}
Int main (int argc, char * * argv)
{
STRING p1, p2, p3; FML_Init (10);
p2 = (char *) malloc (MALLOC_SIZE);
p3 = (char *) malloc (MALLOC_SIZE);
p2 = p3; func03 (p1); p2 = func02 ();
Return 0;
}
```

以上 C 代码中,FML_Init( )为检测程序初始化函数,主要是对用户备用信息栈的初始化。MALLOC_SIZE 为笔者宏定义的分配内存类型大小。内存泄漏信息日志文件显示如下:

<MEM RT DETECT > Size of Leakage: 156, Position test. c:func03 ():22: Malloc Index: 3

<MEM RT DETECT > Size of Leakage: 156, Position test. c: main ():33: Malloc Index: 1

<FML_FDetect > Size of Leakage: 156, Position test. c: main ():34: Malloc Index: 2

<FML_FDetect > Size of Leakage: 156, Position test. c:func01 ():11: Malloc Index: 4

<FML_FDetect > Size of Leakage: 156, Position test. c:func02 ():17: Malloc Index: 5

由文件显示的内容来看,此实例存在的 5 处内存泄漏均被成功捕获,并准确定位至发生泄漏的文件名、函数名和行号。从结果看出,检测实例程序时,自动、实时捕获的泄漏内存块为两处(index 1 和 3),均为显示泄漏类型。

6 结　语

本文提出了面向类型对 UB 块层次性地进行内存泄漏检测的概念,实现了一种新的内存泄漏检测技术,并给出了在 Linux 下实现该检测技术的实现方法及具体实例。经试验该算法能够有效、准确、自动地检测出系统的内存泄漏,并具有较高的实时性。此外,该算法目前还只用于处理内存泄漏故障,如何检测其他内存故障,还需在日后的实践中,不断改进,不断完善。

参 考 文 献

[1] Hastings R, Joyce B. Purify: Fast Detection of Memory Leaks and Access Errors [C]//Proceedings of the Winter USENIX Conference, 1999:125 - 136.

[2] Rational Software Corporation. LCLink Product Information, LCLink Detection of Memory Leaks and Access Error ACM outran Forum, August 2003, Feb22.

[3] Para Soft Corporation. Insure ++ Manuals, 1998.

[4] 张威,卢庆龄,李梅,等. 基于指针分析的内存泄漏故障测试方法研究[J]. 计算机应用研究,2006(10):22 - 24.

[5] Huang J C. Program instrumentation and software testing[J]. Computer, 1978, 11(4):25 - 32.

(上接第 39 页)

本文提出的自主发育体系结构,将认知能力区分不同的等级,只需要给出基本的感知能力、运动能力和自主发育程序,所有的高级认知能力都可以通过发育得到,不需重新编程。该发育过程通过感知发育模块、认知发育模块和行为发育模块获得,每个发育模块都可以自组织增量学习、能学到什么样的知识只和机器人所获得经验有关。各模块之间互相依赖并且可以同时学习,具有实时的自主发育能力。时空经验模块基于知觉经验的直接表示,代表了基于状态而不是基于符号来进行组织的原理,其中不需要抽象的过程,既可以实现反应式行为和慎思式行为的无缝衔接,又可以实现直接的推理过程。

每个模块的功能虽然是相互独立的,但各模块知识可以相互交流转化。如果时空经验模块中与某一任务相关的行动序列被经常调用,逐渐强化,则产生反应式行为;相应的反应式行为的学习也可以来更新时空经验模块;时空经验模块可以抽象出符号化知识模型,实现多机器人共享。一旦利用符号化知识进行一次规划决策后,就获得了自身经验,可以用来更新时空经验模块。所有的模块可以同时发育,一旦出现了新的感知信息,相应的感知发育模块、认知发育模块和行为发育模块以及符号知识模块都被同步更新。

每个发育模块都存在相应的学习和记忆机制。感知发育模块实现了环境特征的自主分类,是一个自组织、增量的学习过程并且需要满足实时性的要求。时空经验模块的知识表示和学习问题,以往的研究不多,是当前自主发育机器人研究的重点问题。行为发育一般利用强化学习获得,但是传统的强化学习方法不适合发育机器人的多任务学习过程,发育机器人的强化学习方法研究也将是以后发育机器人研究的重点问题。

4 结　语

本文深入分析了传统机器人范式分类的思想以及存在的问题,从认知、发育的角度重新给出新的机器人范式分类,新的范式体系完善了智能机器人的认知层次,明确了自主发育在机器人范式中的地位。遵循自主发育的思想,提出了自主发育智能机器人体系结构。该结构将感知-行动作为认知的基元,实现了自主感知分类、时空经验知识、反应式行为的逐层发育。其中各发育模块之间互相依赖并且可以实现同时发育,具有实时的自主发育能力。最后简单分析了适合不同发育模块的知识表示和学习方法,并给出了以后需要深入的研究内容和研究思路。

参 考 文 献

[1] Murphy R R. Introduction to AI robotics[M]. MIT Press, 2003:57 - 63.

[2] Matarić M J, Cliff D. Challenges in evolving controllers for physical robots[J]. Robotics and Autonomous Systems, 1996, 19(1):67 - 83.

[3] Holland J H. Adaptation in natural and artificial systems[M]. University of Michigan Press, 1975:34 - 45.

[4] Weng J, Mcclelland J, Pentaland A, et al. Autonomous mental development by robots and animals[J]. Science, 2001, 291:599 - 600.

[5] Weng J. A theory for mentally developing robots[C]//Proceedings of the 2nd Internatinal Conference on Development and Learning, 2002, MIT, Cambridge, MA, 2002:131 - 140.

[6] Weng J, Zeng S. A theory of developmental mental architecture and the dav architecture design[J]. Humanoid Robotics, 2005, 2(2):145 - 179.

[7] Weng J. On developmental mental architectures[J]. Neurocomputing, 2007, 70(13/15):2303 - 2323.

[8] Weng J, Zhang Y. Candid covariance-free incremental principal component analysis[J]. IEEE Trans Pattern Analysis and Machine Intelligence, 2003, 25(8):1034 - 1040.

[9] Weng J T, Lu W H, Xue X. Multilayer in-place learning networks for modeling functional layers in the laminar cortex[J]. Neural Networks, 2008, 21:150 - 159.

[10] Weng J, Hwang W. Incremental hierarchical discriminant regression [J]. IEEE Transactions on Neural Networks, 2007, 18(2):397 - 415.

[11] Weng J. Developmental robotics: theory and experiments[J]. International Journal of Humanoid Robotics, 2004, 1(2):199 - 236.

[12] Weng J, Hwang W S, Zhang Y, et al. Developmental humanoids: humanoids that develop skills automatically[C]//Proceedings of IEEE International Conference on Humanoid Robots. MIT, Cambridge, MA, 2000.