基于 C++的动态内存实时监测器①

陈 楠

(中国石油化工股份有限公司 石油物探技术研究院, 南京 211103)

摘 要: 通过对地球物理软件研发过程中比较常见内存错误的调研与总结, 设计并实现了一个基于 C/C++的动态内存检测工具, 采用内嵌与关键函数截获方式, 对编译器开放接口进行扩展与改进. 该工具通过对软件运行过程中堆内存使用情况的实时收集、分类统计与分析, 达到动态的监控与检测内存堆栈错误的目的. 以中国石化石油物探技术研究院自主研发的油气综合解释系统 NEWS 子系统-叠前叠后联合解释模块主要流程为例, 用该检测工具对其进行全面的测试与应用. 实践表明, 嵌入监测器的应用软件在开发过程中大幅降低了内存泄漏现象, 运行时减少了内存错误导致的异常崩溃现象, 提高了应用软件的稳定性, 并能够对开发以及测试人员快速定位与分析软件错误起到较强的指导作用.

关键词: 内存泄漏; 重载; 检测; 调用栈; NEWS

Real-Time Monitor of Dynamic Memory Based on C++

CHEN Nan

(Sinopec Geophysical Research Institute, Nanjing 211103, China)

Abstract: Through the summary of the common memory errors research in the geophysical software development process, we design and implement a dynamic memory detection tool based on C/C++ by using the embedded and key function interception, which can expand and improve the compiler open interface. Through the real-time collection, classification, statistic and analysis of the heap memory usage in the running process of software, the tool achieves dynamic monitoring and detection of false memory stack. We take the NEWS software subsystem of prestack and poststack joint interpretation module process as an example to test and apply it comprehensively. The practice shows that the monitor, embedded in the software, can greatly reduce the memory leak phenomenon during the development process, and decrease the runtime memory errors caused by abnormal collapse phenomenon, and improve the stability of the application software. It also plays a strong role in the development to help tester's in rapid location and analysis of software errors.

Key words: memory leaks; overload; detection; call stack; NEWS

1 引言

随着野外油气勘探技术的发展与进步, 采集数据信息量的增多也使得地震数据变得日趋庞大, 大数据量任务处理一直是地球物理软件研发人员不断研究与探索的技术课题, 计算机物理内存的管理也越来越复杂[1], 操作系统从 32 位升级到 64 位, 理论上将 64 位 CPU 的寻址空间从 4GB 扩展到了无穷大(远远超出物理内存大小), 解决了程序设计中物理内存分配的上限问题, 不过这样往往导致了另外一个问题: 程序员们对手动释放物理内存的意识逐渐变得淡薄, 以至于在系统性能变得低下甚至频临崩溃时才意识到问题严重, 但此时的开发规模也使得排查工作变得复杂无序, 笔者虽具备多年一线软件研发经验, 但当出现内存问题累积引发的系统性能问题方面, 也感到维护工作非常

① 基金项目:国家科技重大专项(2011ZX05035)

收稿时间:2016-03-14;收到修改稿时间:2016-04-19 [doi: 10.15888/j.cnki.csa.005459]

困难, 轻者按程序流程分支逐一排查测试, 重则推倒重来, 新手遇到类似问题更是感到无从下手, 往往严重影响项目研发进度. 针对这些现象, 第三方解决方案其实已经存在, Windows 系统下有 MFC 内嵌的内存监测工具可以实时跟踪应用软件内存情况并提示定位内存泄漏信息[2,3], 但该工具不支持 MingW 编译器, Linux 系统下相对来说就没有那么容易使用的解决方案, 而 mpatrol 之类的现有工具, 在易用性、附加开销和性能等方面都不是很理想[4-6].

一个成熟的软件产品往往都已经对外发布试用与销售, 随着应用范围的扩大, 不可避免地也会存在一定数量的 bug, 如果不具备远程调试的条件, 开发人员仅仅从用户的反馈信息中很难准确定位软件在使用过程中出现的问题.

综上所述, 在软件工程开发的过程中, 一个具备全程动态跟踪功能的、跨平台的错误定位与内存检测工具是很有必要的.

2 开发需求

一般情况下, 实时检测程序应该具备以下几个基本功能: ①全程跟踪并记录程序中内存分配和释放情况[7]; ②监控分配的内存是否为有效数据; ③在发生异常崩溃时进行跳转处理并回溯崩溃点的函数调用栈, 提示开发人员定位崩溃点位置找出错误; ④支持线程安全.

3 关键技术研发

3.1 程序调用栈(call stack)原理分析

功能: 主要用于程序异常退出时寻找错误点, 回溯堆栈, 列出当前错误序列函数的调用关系.

原理: 通过对当前堆栈的分析, 找到其上层函数在栈中的帧地址, 再分析上层函数的堆栈, 再找再上层的帧地址, 一直找到最顶层为止, 帧地址指的是一块在栈上存放局部变量, 上层返回地址, 及寄存器值的空间.

要了解调用栈, 首先需要了解函数的调用过程, 下面用一段简单代码作为例子:

```
#include <stdio.h>
int add(int a, int b)
{
    int result = 0;
    result = a + b;
    return result;
}
int main(int argc, char *argv[ ])
{
    int result = 0;
    result = test(1, 2);
    printf("result = %d \r\n", result);
    return 0;
}
```

通过对代码的反汇编可以看到(图 1), 进入 add 函数后, 首先进行的操作是将当前的栈基址 ebp 压栈(此栈基址是调用者 main 函数的), 然后将 ebp 指向栈顶 esp, 接下来再进行函数内的处理流程. 函数结束前, 会将函数调用者的栈基址恢复, 然后返回准备执行下一指令. 这个过程中, 栈上的空间展布如图 2 所示.

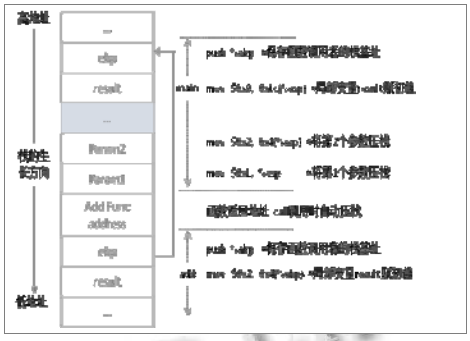

图 1 反汇编代码解析

可以发现, 每调用一次函数, 都会对调用者的栈基址(ebp)进行压栈操作(图 2), 并且由于栈基址是由当时栈顶指针(esp)而来, 会发现, 各层函数的栈基址很巧妙的构成了一个链, 即当前的栈基址指向下一层函数栈基址所在的位置, 如图 2 所示.

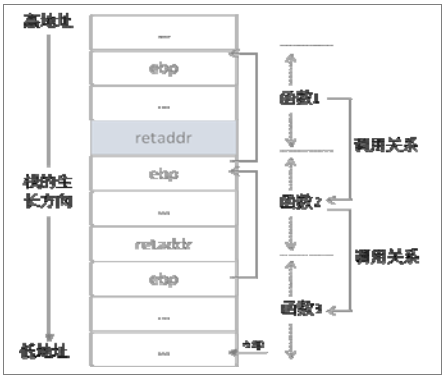

图 2 调用栈关系

了解函数的调用过程, 想要回溯调用栈也就容易了, 首先获取当前函数的栈基址(寄存器 ebp)的值, 然后获取该地址所指向的栈的值, 该值也就是下层函数的栈基址, 找到下层函数的栈基址后, 重复刚才的动作, 即可以将每一层函数的栈基址都找出来, 这也就是我们所需要的调用栈了,前面描述的是函数栈的内存结构关系和调用逻辑, 在不同编译器条件下可能会出现内存高低位置的互换, 但并不影响对栈结构的分析和研究[8,9].

3.2 动态内存泄漏检测

功能: 实时监控内存分配与释放情况.

原理: 重载与替换系统内存分配释放函数.

栈内存在函数结束时由系统自动回收释放, 所以不会造成泄漏现象, 一般我们常说的内存泄漏是指堆内存的分配与释放不匹配导致[10-12]. 堆内存是程序从堆中分配的, 大小任意的(理论上内存块的大小可以在程序运行期决定), 使用完后必须显示释放的内存. 应用程序一般使用 malloc, realloc, new 等函数从堆中分配到一块内存, 使用后, C++程序员必须手动调用相应的 free 或 delete 释放该内存块, 否则, 这块内存就不能被系统回收再次使用, 从而造成泄漏. 内存泄漏不仅仅包含堆内存的泄漏, 还可以包含系统资源的泄漏(resource leak), 比如核心态指针如 HANDLE, GDI Object, SOCKET, Interface 等, 从根本上说这些由操作系统分配的对象也消耗内存, 如果这些对象发生泄漏最终也会导致物理内存的丢失. 而且, 某些对象消耗的是核心态内存, 这些对象严重泄漏时会导致整个操作系统不稳定. 所以相比之下, 系统资源的泄漏比堆内存的泄漏更为严重[13]. 以发生的方式来分类, 内存泄漏可以分为 4 类:

① 常发性内存泄漏. 发生内存泄漏的代码会被多次执行到, 每次被执行的时候都会导致一块内存泄漏, 这种泄漏在地球物理软件批量处理模块中比较常见.

② 偶发性内存泄漏. 发生内存泄漏的代码只有在某些特定环境或操作过程下才会发生, 比如在某分支条件下的用户交互操作. 对于特定的环境, 偶发性的也许就变成了常发性的. 所以测试环境和测试方法对检测内存泄漏也是至关重要.

③ 一次性内存泄漏. 发生内存泄漏的代码只会被执行一次, 或者由于算法上的缺陷, 导致总会有一块仅且一块内存发生泄漏. 比如, 在类的构造函数中分配内存, 在析构函数中却没有释放该内存, 所以内存泄漏只会发生一次, 或者是一些全局静态指针等等.

④ 隐式内存泄漏. 程序在运行过程中不停的分配内存, 但是直到结束的时候才释放内存. 严格的说这里并没有发生内存泄漏, 因为最终程序释放了所有申请的内存. 但是对于一个服务器程序, 需要运行几天, 几周甚至几个月, 不及时释放内存也可能导致最终耗尽系统的所有内存. 所以, 我们称这类内存泄漏为隐式内存泄漏, 作为一般的用户, 通常感觉不到内存泄漏的存在. 真正有危害的是内存泄漏的堆积[14], 这会最终消耗尽系统所有的内存. 从这个角度来说, 一次性内存泄漏并没有什么危害, 因为它不会堆积, 而隐式内存泄漏危害性则较大, 因为它更难被检测到, 所以这方面的问题需要从软件程序的优化方面考虑解决.

检测内存泄漏的关键是要能截获住对分配内存和释放内存的函数的调用[15]. 截获住这两个函数, 我们就能跟踪每一块内存的生命周期, 比如, 每当成功的分配一块内存后, 就把它的指针加入一个全局的 list 中; 每当释放一块内存, 再把它的指针从 list 中删除. 这样, 当程序结束的时候, list 中剩余的指针就是指向那些没有被释放的内存. 这里只是简单的描述了检测内存泄漏的基本原理[16], 文章后面再详述讨论. 在 new/delete 操作中, C++为开发人员产生了对 operator new 和 operator delete 的调用. 这是不能改变的. 对于"new int", 编译器会产生一个调用"operator new(sizeof(int))", 而对于"new char[10]", 编译器会产生"operator new[](sizeof(char) * 10)"(如果 new 后面跟的是一个类名的话, 编译器还要调用该类的构造函数). 类似地, 对于"delete ptr"和"delete[] ptr", 编译器会产生"operator delete(ptr)"调用和"operator delete[](ptr)"调用(如果 ptr 的类型是指向对象的指针的话, 那在 operator delete 之前还要调用对象的析构函数). 当用户没有提供这些操作符时, 编译系统自动提供其定义; 而当用户自己提供了这些操作符时, 就覆盖了编译器提供的函数版本, 从而找到了可获得对动态内存分配操作的精确跟踪和控制的入口. 基于此, 利用自定义宏在用户程序中进行替换, 重载操作符 operator new, 格式如下所示.

```
void* operator new(size_t nsize, const char* cfile,
```

int nline);

void* operator new[](size_t nsize, const char* cfile, int nline);

其他内存分配操作符如表 1 所示, 重载后的 operator new 可以跟踪覆盖应用程序内部所有的内存分配调用, 并在指定的检查点上对不匹配的 new 和 delete 操作进行自定义处理.

表 1 其他内存分配操作符

序号	操作符
1	new T
2	new T[]
3	new()T
4	new() T[]
5	operator new(size_t)
6	operator new[](size_t)
7	operator new(size_t size, Type value)
8	operator new[](size_t size, Type value)

3.3 功能扩展

3.3.1 监测记录

通过重载系统内存操作符,用户接替了编译器对物理内存分配的管理,灵活度大幅提高, 此时可以嵌入必要的记录机制[15,16], 达到用户对应用程序内存使用情况的全面监测. 比较常用的方法是记录分配内存的指针地址[17-20], 这样消耗的内存空间比较小, 检索的速度也比较快, 为了提高检索效率, 可以定义 STL 的散列表对象, 把申请内存的文件名、行号、对象大小信息分别存入 file、line 和 size 字段中, 然后返回(malloc 返回的指针 + sizeof(new_ptr_list_t)). 在 delete 时, 则在散列表中搜索, 如果找到的话((char*)链表指针 + sizeof(new_ptr_list_t) == 待释放的指针), 则调整链表、释放内存, 找不到的话报告删除非法指针并 abort, 过程如图 3 所示. 要得到精确的内存泄漏检测报告, 可以在文件开头包含"debug_new.h". 包含的位置应当尽可能早, 除非跟系统的头文件(典型情况是 STL 的头文件)发生了冲突. 在某些情况下, 可能会不希望 debug_new 重定义 new, 这时可以在包含 debug_new.h 之前定义 DEBUG_NEW_NO_NEW_REDEFINITION, 这样的话, 在用户应用程序中应使用 debug_new 来代替 new(顺便提一句, 没有定义 DEBUG_NEW_NO_NEW_REDEFINITION 时也可以使用 debug_new 代替 new). 在源文件中也许就该这样写:

```
#ifdef _DEBUG
#define
DEBUG_NEW_NO_NEW_REDEFINITION
#include "debug_new.h"
#else
#define debug_new new
#endif
```

并在需要追踪内存分配的时候全部使用 debug_new(考虑使用全局替换). 用户可以选择定义 DEBUG_NEW_EMULATE_MALLOC, 这样 debug_new.h 会使用 debug_new 和 delete 来模拟 malloc 和 free 操作, 使得用户程序中的 malloc 和 free 操作也可以被跟踪, 反之也可以选择性的宏定义启动开关, 在发布版本中用户可以关闭跟踪功能, 避免实际应用中效率的降低.

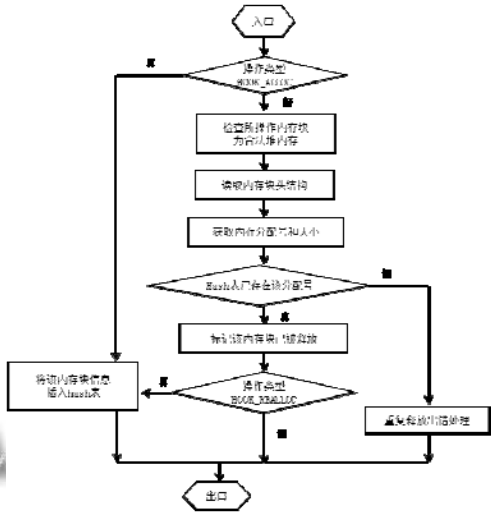

图 3 监测记录流程图

3.3.2 多线程支持

线程是否可重入是编写监测器 API 函数必须考虑的情况, 线程安全问题都是由全局变量及静态变量引起的, 内存监测器本身就是一个全局控制器[21-23], 内部有比较频繁的 I/O 操作, 所以要加入读写互斥锁用以保证线程安全性[24,25], 加入互斥锁机制后的监控器才能提供多线程支持, 这也是通用性设计的一个方面.

4 实际应用

中石化物探院 NEWS 软件子系统-“叠前叠后联合解释软件”是一套集实时叠前分析、偏移距/入射角部

分叠加、AVO 属性分析及 P 波叠前裂缝检测等功能为一体的综合性叠前数据解释工具，软件工程方面也是具备了实时联动分析、交互图形显示和批量处理功能等综合特征. 前期研发过程中较侧重于模块功能效果开发，导致了该系统自 1.0 版本起就有轻微的内存泄漏与越界现象，但并不明显，不影响研究使用，随着功能的扩充和应用范围的加大，在经过反复的交互使用与图形显示输出后，出现了内存使用明显增多的现象，从程序启动时的 80M，经过约 168 小时的方位各向异性批量处理运算后可以飙升到 800 M 左右，运行该模块的机器性能明显下降.

由于工程代码量大，程序复杂，用传统的手工逐一排查方式工作量和风险性都很大，开发团队随即在系统中嵌入了动态内存检测工具，进行了全覆盖的排查，原因分析与修正：1、交互过程中有零散的内存泄漏现象，通过检测器记录的泄漏位置逐一修正释放. 2、检测器记录显示，在进行单机多线程并行计算时，程序定义的 n 个线程并发进行内存分配，并将指针传递给一个数据存储容器，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. 这种问题在系统中较难发现，程序可能运行一段时间没有问题，从而通过了不严密的系统测试. 但是如果测网范围大，计算时间长的情况下，系统将不定时的崩溃，而且崩溃的原因从程序表象上都比较难检测. 为了解决这种问题，我们在检测器内部增加了一个动态检测模块，同时启动一个分析线程，每隔一定的时间间隔就计算一下当前的以分配而尚未释放的内存信息，并以内存的分配位置为关键字进行统计，查看在同一位置(相同文件名和行号)所分配的内存总量和其占进程总内存量的百分比. 这样在程序运行过程中，用户能够对程序的动态内存分配状况进行监视. 当客户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检测器. 检测器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，同时以调用 new 或者 malloc 进行内存分配的文件名和行号为索引值，计算每个内存分配动作所分配而未释放的内存总量. 用这种方法，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，说明该位置的内存分配方式需要调整了. 当得到 operator new 的信息时，记录内存分配信息，当收到 operator delete 消息时，删除相应的内存分配信息.

借助内嵌的 call stack 调用栈回溯机制，测试过程中出现的程序异常崩溃，可以快速准确的定位到出错文件与行号(如图 4 及图 5 所示)，使得纠错效率大幅度的提高，修正后的“叠前叠后联合解释软件”在实际生产应用中运行稳定，计算速度也得到明显提升.

```
207  JobMonitorMainWindow::~JobMonitorMainWindow()
208  {
209      int nList = m_pRsProcessList.size();
210      for( int i = 0; i < nList; i++ )
211      {
212          if( NULL != m_pRsProcessList.at(i))
213          {
214              delete m_pRsProcessList.at(i);
215          }
216      }
217      m_pRsProcessList.clear();
218      if( m_pUiTimer )
219      {
220          delete  m_pUiTimer;
221          m_pUiTimer = NULL;
222      }
223      test1();
224  }
225
226  void JobMonitorMainWindow::test1()
227  {
228      int* p = NULL;
229      *p = 20;
230  }
```

图 4　异常函数示例

```
/*************************************/
< cn : Memory Exception by Crash Filter >
0x402044 : D:\projects\Batch_app\rs_bin\BatchJobMonitor.exe : D:\projects\
.cpp
0x401e65 : D:\projects\Batch_app\rs_bin\BatchJobMonitor.exe : D:\projects\
.cpp :
(In Line : 223) , (In Func : ~JobMonitorMainWindow)
0x40393c : D:\projects\Batch_app\rs_bin\BatchJobMonitor.exe : D:\projects\
(In Line : 28) , (In Func : main)
0x401241 : D:\projects\Batch_app\rs_bin\BatchJobMonitor.exe : dyncast.cc
0x401298 : D:\projects\Batch_app\rs_bin\BatchJobMonitor.exe : dyncast.cc
Failed to init bfile from (C:\windows\syswow64\kernel32.dll)
0x75db336a : C:\windows\syswow64\kernel32.dll : BaseThreadInitThunk
Failed to init bfile from (C:\windows\SysWOW64\ntdll.dll)
0x77d99882 : C:\windows\SysWOW64\ntdll.dll : RtlInitializeExceptionChain
Failed to init bfile from (C:\windows\SysWOW64\ntdll.dll)
0x77d99855 : C:\windows\SysWOW64\ntdll.dll : RtlInitializeExceptionChain
```

图 5　异常错误定位

```
*** memory monitor v1.0-cn ***
*** about 96 memory leaks detected,total size: 3 KB ***
AppSeisBase3DSdiMainWindow.cpp, 330, size=8
AppSeisBase3DSdiMainWindow.cpp, 334, size=8
AppSeisBase3DSdiMainWindow.cpp, 338, size=8
AppSeisBase3DSdiMainWindow.cpp, 342, size=8
AppSeisBase3DSdiMainWindow.cpp, 346, size=8
AppSeisBase3DSdiMainWindow.cpp, 350, size=8
AppSeisBase3DSdiMainWindow.cpp, 364, size=8
AppSeisBase3DSdiMainWindow.cpp, 368, size=8
AppSeisBase3DSdiMainWindow.cpp, 372, size=8
AppSeisBase3DSdiMainWindow.cpp, 376, size=8
AppSeisBase3DSdiMainWindow.cpp, 380, size=8
AppSeisBase3DSdiMainWindow.cpp, 384, size=8
AppSeisBase3DSdiMainWindow.cpp, 388, size=8
AppSeisBase3DSdiMainWindow.cpp, 392, size=8
rsuibearingdialog.cpp, 15, size=160
rsaetting.cpp, 311, size=16
soft_lmap_denoising.cpp, 178, size=32
soft_lmap_denoising.cpp, 181, size=32
soft_lmap_denoising.cpp, 184, size=32
rsanisotropicoubedialog.cpp, 31, size=552
AppSeisProfileLink3DSdiMainWindow.cpp, 210, size=8
AppSeisProfileLink3DSdiMainWindow.cpp, 225, size=8
AppSeisLink3DSdiMainWindow.cpp, 101, size=8
AppSeisLink3DSdiMainWindow.cpp, 108, size=8
AppSeisLink3DSdiMainWindow.cpp, 115, size=8
AppSeisAVOLink3DSdiMainWindow.cpp, 79, size=8
AppSeisAVOLink3DSdiMainWindow.cpp, 86, size=8
RSAppSeis3DSdiMainWindow.cpp, 124, size=8
RSAppSeis3DSdiMainWindow.cpp, 128, size=8
rssetting.cpp, 37, size=8
```

图 6　内存泄漏监控记录

5 结论

开发了一套跨平台的基于 C/C++的动态内存检测工具, 集成了函数调用栈回溯、实时内存分配监测、内存使用与释放回馈等技术, 以图形或者文本的方式向开发用户提交检测预警记录(如图6所示), 在实际项目的使用中具备以下几个特征:

① 大型软件开发过程明显减少调试时间;
② 支持多平台和跨平台开发;
③ 降低维护和支持成本;
④ 实时排除算法错误;
⑤ 减少软件缺陷提高产品信誉;
⑥ 能够与项目产品的开发生命周期无缝集成.

参考文献

1 陈楠,祝媛媛,张光德,徐钰.基于 QT 的地震勘探可扩展平台研发与应用.华北地震科学,2013,4:1003–1375.

2 周宇.基于调用栈完整性的缓冲区溢出检测方法.计算机安全,2010,(3):16–19.

3 管晓宏,冯力,孙杰,等.Linux 环境下基于调用栈图的入侵检测方法: CN,CN 1710866 A. 2005.

4 吴多多,杨伟伟,肖力涛.利用堆栈模拟 C 语言中函数的调用过程.科技信息,2011,(15).

5 杨礼波,张志亮.在堆栈缓冲区溢出中程序调用的分析和研究.电脑知识与技术,2010,6(17):4686–4689.

6 徐建国.网络化制造系统中虚拟加工若干关键技术研究[博士学位论文].南京:南京理工大学,2007.

7 肖谦,李中升,漆锋滨.Linux 下函数栈大小的自动计算技术.计算机工程,2011,(S1).

8 梁玉,傅建明,彭国军,等.S-Tracker: 基于栈异常的 shellcode 检测方法.华中科技大学学报(自然科学版),2014,(11):39–46.

9 杨东.ARM 嵌入式系统异常调试的研究和实现[硕士学位论文].上海:上海交通大学,2009.

10 戚晓芳,周晓宇,徐晓晶,等.一种组合式基于调用栈的程序切片方法.东南大学学报(自然科学版),2011,41(6):1171–1176.

11 周国亮,朱永利,王桂兰,等.实时大数据处理技术在状态监测领域中的应用.第六届电工技术前沿问题学术论坛.2014.

12 王呈祥.基于共享内存的通用 OLE Process Control3.0 服务器的研究与设计[硕士学位论文].长春:吉林大学,2013.

13 刘黎志,刘君.空气质量实时监测系统的内存泄漏.武汉工程大学学报,2012,34(6):74–78.

14 周游弋.集群监控系统中内存数据库的设计与应用研究[硕士学位论文].上海:复旦大学,2010.

15 张飞.实时嵌入式操作系统动态内存管理研究[学位论文].合肥:中国科学技术大学,2008.

16 李明.多核路由器动态内存分配器的设计与实现[硕士学位论文].南京:南京理工大学, 2011.

17 刘晓伟.基于 Linux 的电力系统实时监测系统的开发[硕士学位论文].武汉:华中科技大学,2006.

18 何波玲,张志春.内存信息自动监控器的设计与实现.信息安全与技术,2015,(6).

19 姚志强.一个实用的自释放内存 TSR 程序的设计要素.福建师范大学学报(自然科学版),1997,(2):27–32.

20 吴民,涂奉生.Linux 下面向函数的动态内存泄漏监测.计算机工程与应用,2003,39(6):37–40.

21 孝瑞.内存动态安全监测及防范研究[硕士学位论文].保定:华北电力大学,2015.

22 王文陵.内存泄漏的处理与监测.福建信息技术教育,2005(3).

23 杨丽,郭祥昊.C 语言常见内存管理错误及内存监控功能的实现.新浪潮,1996,(8):21–22.

24 陈鸿杰.物联网在煤矿安全生产应用中监测与控制技术研究[硕士学位论文].西安:西安电子科技大学,2014.

25 寇雅楠,李增智,王建国,等.计算机软件测试研究.计算机工程与应用,2002,38(10):103–105.