|  |  |
| --- | --- |
|  |  |
|  |  |

程序调试实践

Program Debug Practice

王浩

（软件工程师）

程序调试实践

中文摘要

随着计算机行业的不断发展和信息技术的继续迭代，越来越多的行业在计算机和互联网的帮助下来提升自身的效率。这对于软件工程来说意味着会有更复杂更多样的需求，而且这些需求也会使得代码变得复杂，程序更加庞大。而对于程序员们来说，这些变化意味着他们需要有更优秀的编程能力和代码调试能力。

本文主要介绍当程序出了不符合预期的表现时，也就是当程序出了BUG时，程序员应当如何去找到出现这个问题的原因。当然为了解释清楚BUG的来龙去脉，本文来将会介绍与之有关的计算机的必要知识，这些知识主要集中在操作系统和程序计算机中的如何编译运行的。

由于目前开发的环境大多集中于UNIX体系的操作系统中，所以本文着重介绍的还是在Linux的操作系统下程序调试相关内容，其中最主要涉及的就是Linux下关于调试方法的研究，比如GDB等调试工具的使用和在调试过程的中应用等。

**关键词：软件调试、Linux、GDB、BUG、DEBUG**

**Program Debug Practice**

**Abstract**

With the development of computer industry and information technology grows, more and more industries are making themselves more efficient with the help of internet and computer. it means that there are more complex and diversified requirement for software engineering, which will make code become sophisticated and make program become huge. Meanwhile, these changes will require programmer with perfect coding skills and excellent code debug ability to cover these solutions.

This paper mainly introduces how to find out the cause of problem for programmer when the program appears non-expected performance, in other words, the program appears with a bug. Of course, in order to better explain the context of the bug, this paper also will introduce some related necessary knowledge about computer and program, and this knowledge mainly center on operation system and how does the program compile and run in the computer.

So far, because of the development environment mainly fasten-on class of the UNIX operation system, so this article focuses on the introduction of the relevant content of program debug technique under Linux operation system, in which the most of important of all is about research on debugging methods in Linux, such as the use of debugging tools and application during debugging process.

**Keywords:** **software、Linux、GDB、BUG、DEBUG**

Table of Contents

[第一章 绪 论 1](#_Toc131495497)

[1.1 程序调试研究的原因 1](#_Toc131495498)

[1.2 程序调试研究的目的 1](#_Toc131495499)

[1.3 程序调试研究的意义 2](#_Toc131495500)

[第二章 程序调试的介绍 3](#_Toc131495501)

[2.1 程序调试的背景 3](#_Toc131495502)

[2.1.1 什么是程序调试 3](#_Toc131495503)

[2.1.2 什么是BUG 3](#_Toc131495504)

[2.1.3 程序调试和程序测试的区别 4](#_Toc131495505)

[2.2 程序调试的目的 5](#_Toc131495506)

[2.2.1 为什么要进行程序调试 5](#_Toc131495507)

[2.2.2 什么时候进行程序调试 5](#_Toc131495508)

[2.2.3 程序调试的步骤 6](#_Toc131495509)

[第三章 程序的知识准备 7](#_Toc131495510)

[3.1 操作系统相关知识 7](#_Toc131495511)

[3.1.1 字节序 7](#_Toc131495512)

[3.1.2 寄存器 7](#_Toc131495513)

[3.1.3 地址空间 9](#_Toc131495514)

[3.1.4 数据类型 9](#_Toc131495515)

[3.1.5 堆栈 10](#_Toc131495516)

[3.2 程序运行相关知识 10](#_Toc131495517)

[3.2.1 程序的编译链接过程 10](#_Toc131495518)

[3.2.2 进程的虚拟地址 13](#_Toc131495519)

[3.2.3 程序的加载运行过程 15](#_Toc131495520)

[3.3 汇编语言相关知识 16](#_Toc131495521)

[3.3.1 汇编语言语法 16](#_Toc131495522)

[3.3.2 汇编语言实战 18](#_Toc131495523)

[第四章 调试的知识准备 31](#_Toc131495524)

[4.1 GDB基础命令 31](#_Toc131495525)

[4.1.1 断点 31](#_Toc131495526)

[4.1.2 命令行 32](#_Toc131495527)

[4.1.3 程序栈 33](#_Toc131495528)

[4.1.4 多线程、多进程 33](#_Toc131495529)

[4.1.5 打印输出 34](#_Toc131495530)

[4.1.6 函数跳转 36](#_Toc131495531)

[4.1.7 其他 36](#_Toc131495532)

[4.2 GDB调试步骤 37](#_Toc131495533)

[4.2.1 生成可调试的执行程序 37](#_Toc131495534)

[4.2.2 设置生成可调试的core文件 37](#_Toc131495535)

[4.2.3 调试程序 38](#_Toc131495536)

[第五章 简单的程序调试实践 39](#_Toc131495537)

[5.1程序调试实践之调试main函数 39](#_Toc131495538)

[5.1.1 GDB启动程序调试过程 39](#_Toc131495539)

[5.1.2 breakpoint断点和打印调试过程 42](#_Toc131495540)

[5.1.3 watchpoint 断点调试过程 45](#_Toc131495541)

[5.1.4 catchpoint 断点调试过程 46](#_Toc131495542)

[5.1.5 输出打印调试过程 50](#_Toc131495543)

[5.2程序调试实践之调试函数 52](#_Toc131495544)

[5.2.1 对程序的函数进行调试 52](#_Toc131495545)

[5.3程序调试实践之调试静态库 56](#_Toc131495546)

[5.3.1 静态库的生成 56](#_Toc131495547)

[5.3.2 静态库的调试 57](#_Toc131495548)

[5.4程序调试实践之调试动态库 60](#_Toc131495549)

[1. 动态库的生成 60](#_Toc131495550)

[2. 动态库的调试 61](#_Toc131495551)

[第六章 多进程程序调试实践 64](#_Toc131495552)

[6.1 多进程程序编译运行 64](#_Toc131495553)

[6.1.1. Linux 多进程介绍 64](#_Toc131495554)

[6.1.2. Linux 多进程编程 64](#_Toc131495555)

[6.2 多进程程序调试实践 65](#_Toc131495556)

[6.2.1. 简单版多进程实例 65](#_Toc131495557)

[6.2.2. 进阶版多进程实例 68](#_Toc131495558)

[6.2.3. 升级版多进程实例 69](#_Toc131495559)

[第七章 多线程程序调试实践 72](#_Toc131495560)

[7.1 多线程程序编译运行 72](#_Toc131495561)

[7.1.1 线程的基本概念 72](#_Toc131495562)

[7.1.2 线程的相关函数 72](#_Toc131495563)

[7.1.3 线程的简单实例 74](#_Toc131495564)

[7.2 多线程程序调试实践 75](#_Toc131495565)

[7.2.1 多线程基础程序调试 75](#_Toc131495566)

[7.2.2 多线程线程锁程序调试 77](#_Toc131495567)

[第八章 程序异常有关的调试实践 81](#_Toc131495568)

[8.1 与程序死锁有关的调试实践 81](#_Toc131495569)

[8.1.1 死锁进程的调试 81](#_Toc131495570)

[8.1.2 转储文件的死锁调试 88](#_Toc131495571)

[8.2 与内存检查有关的调试实践 92](#_Toc131495572)

[8.2.1 内存泄露 92](#_Toc131495573)

[8.2.2 堆溢出 94](#_Toc131495574)

[8.2.3 栈溢出 97](#_Toc131495575)

[8.2.4 全局内存溢出 100](#_Toc131495576)

[8.2.5 使用释放的内存 102](#_Toc131495577)

[8.3 release版本程序调试实践 105](#_Toc131495578)

[8.3.1 从调试版中提取调试符号 105](#_Toc131495579)

[8.3.2 直接使用调试版作为符号源 107](#_Toc131495580)

[参考文献 109](#_Toc131495581)

[致谢 109](#_Toc131495582)

# 绪 论

## 1.1 程序调试研究的原因

随着社会科学技术的发展，各行各业对于计算机的需求也越来越旺盛，随之而来的就是计算机的从业的人员也越来越多，计算机行业也在快速的迭代和发展。计算机行业中对于编码方面的书籍和论文数不胜数，但是对于程序调试方法的研究却是寥寥无几。

在一个程序的实现过程中，程序的编码无疑是占了一个很重要的地位，但是程序调试的地位也丝毫不逊色于程序编码的重要性。甚至程序就是在写代码和改代码的过程中，逐步的去满足最终的需求。程序的调试过程甚至于要比编码在项目周期中占用的时间更多。所以对程序调试进行研究这一工作是一件很有必要的事。

虽然程序调试在程序开发过程中占有很重要的位置，但是目前来说，程序员大多只能靠经验来支撑这一部分的工作，大多数人只能通过不断在工作中踩坑来积累经验，并且市面上对于调试部分的研究论文和书籍也是寥寥无几。并且各种语言和项目又千差万别，不能一概而论的现象又加剧了这种情况的延续。

所以这篇文章就是基于以上的理由，开始研究关于在Linux系统下关于C/C++程序的调试的一些方法和步骤，特别是一些Linux自带的工具甚至也可能涉及到一些第三方工具的使用，都是为了更好的对调试做一些介绍和研究。

## 1.2 程序调试研究的目的

这篇文章的研究目的主要有两个：

其一：旨在希望当有需要的人在读完这篇文章以后，能帮助他们快速上手关于C/C++程序在Linux系统下的调试方法。为什么要强调适用语言，主要是各个语言对于本身的调试调试支持，如果不是限制到该语言中，有些方法和工具是需要具体分析的，有些语言对于调试支持的比较好会有自己的输出逻辑和框架。此外说是在Linux系统下，其实相关的Unix下，关于C/C++语言的调试都是可以使用到的，因为我们需要用到的调试工具GDB在Linux系列的系统下大都支持，并且如果在GDB工具不支持使用的情况下，还有一些更好用的第三方工具可以选择来帮助到调试。

其二：旨在帮助学习这篇文章的人可以借着学习调试的过程中来更深入的学习计算机底层的知识，构建计算机操作系统的知识网络。因为C/C++是一门和计算机底层联系比较密切的语言，有C/C++编译出来的代码很容易就和操作系统做了很深的交互，如果对底层知识可以很熟悉的运用和了解的话，对于一些调试的方法可以起到事半功倍的效果。此外其他语言虽然可能有自己的调试模块来帮助程序员做对应的调试工作，但是这些部分最终都是需要走到计算机操作系统的调用。所以对于学习操作系统这件事来说，对于程序员提升自己的计算机素养，很有帮助且，很有必要。

## 1.3 程序调试研究的意义

进行程序调试的细致研究，主要是为了通过对程序调试的前因后果的过程探索，梳理调试过程中涉及到的知识体系，可以让程序员掌握计算机更深层次的知识，更重要的是通过此次研究过程，可以让程序员能构建出对程序调试这一过程中的知识网络。

现阶段，大部分程序员对于程序在计算机中的操作系统等知识一直都是一知半解，但是随着互联网的继续发展下去，计算机会和各行各业都进行一个赋能结合的过程，这也就意味着程序员除了需要有过硬的编程能力之外，还需要了解计算机的底层知识和对程序的Debug能力，这样才能应对在整个程序开发周期过程中面临的主要的编码和调试问题。

而通过对程序调试的研究，不仅可以帮助程序员了解和梳理计算机底层知识，还能学习在面对的程序出现不符合预期的现象时，如何去快速准确追踪溯源的找到该问题发生的原因，并且恰到好处的解决问题。

除此之外，通过研究程序调试的方法，要求程序员去掌握和了解在操作系统层次下关于程序的调试工具的使用方法，和培养当没有直接输出时如何去模拟程序运行环境下程序调试的能力。

总而言之，程序调试研究的意义主要是想让程序员掌握计算机底层的知识以及面对没有直接调试信息输出时，或者只有调试信息如何去准确的确定问题并解决问题的能力。

# 程序调试的介绍

## 2.1 程序调试的背景

### 2.1.1 什么是程序调试

每一个程序都是为了解决一个问题。所以在一个完整的，规范的项目实现过程中，程序的最初来源是需求定义，然后是根据需求，进行程序的总体设计，然后再是根据总体设计制定详细设计，再经过程序员的编码，系统测试步骤，程序测试调试等步骤，最终完成一个程序来解决开头提出的那个问题。当然这些步骤的能实现的前提是项目立项，需求确定。

当需求以文档确定好以后，程序员会拆分整个需求，构建程序架构，然后分配各个模块到每个组，或者每个人手中。程序在编码过程中会有自测过程，当一个模块完成以后会进行模块或者单元测试，验证程序是否像需求文档描述那样符合预期的运转。自测通过以后，程序会整理打包交到测试人员手中进行正式的系统测试，发现问题。通过几轮测试修改，程序运行结果如需求文档要求后，就可以正式对外发布正式版本，交付最终产品。

在这个过程中，程序员在开发时进行的自测过程，也算是程序调试。当测试人员发现整体某一点和需求文档有出入的话，会通过约定的方式通知开发人员这处错误，开发人员会继续修改这个问题，这个过程也需要进行程序调试。

程序调试的目的就是为了解决程序中不符合预期现象，而方法就是通过追踪程序中流转的数据，找到数据在哪一部分出现不符合预期的错误。而这个方法就是我们接下来需要重点讨论的话题，并且程序调试的方式多种多样，但是最根本的就是通过种种手段来跟踪程序数据的流转和处理过程。这些手段主要分为系统或者第三方工具来追踪程序堆栈或者是使用者自己使用输出日志形式观察程序运行过程的数据并且保存下来。

总结就是，运行程序检查程序是否符合预期的这一过程就叫做程序调试，其中进行程序调试的时机主要有两个，一个当程序员根据需求初步完成编码时需要进行自测，二是当整体模块编写完成并且通过时，程序被以正式版本的形式交给专业的测试人员进行系统测试。

### 2.1.2 什么是BUG

BUG原意为虫子或者臭虫，在现代的计算机行业中通常被视为在程序中的缺陷或者问题。

BUG从原意到现在的引申义之间是有一个故事的，最初的计算机都是由数量庞大的真空管组合而成，并且还需要用庞大的电力使得真空管发光。当时的计算机体积约有一间屋子大小，1945年，在计算机工作时，一只小虫子飞蛾被计算机的电和光所吸引，落在了计算机内部一个继电器的触点上，导致计算机无法继续工作。然后这种导致计算机无法正常工作的现象就被称为BUG，与之相对应的解决问题的过程则被称为DEBUG。

广义上的BUG是由电脑系统的硬件、系统软件、或者应用软件出错；而狭义的BUG指的是程序的漏洞和错误，除此之外BUG还包括由测试工程师或者用户所发现的和提出的软件可改进的细节、或者与需求文档存在差异的功能实现等。

BUG出现的原因则有1）对各种流程分支考虑的不够充分；2）对于边界处理有错误；3）编码的错误导致；任何软件在发布的时候都不可能是零BUG的状态，并且现在的软件复杂程度早就超过了一般人能控制的范围了，所以如何减少BUG的出现以及DEBUG软件开发过程中一个重要的部分。

### 2.1.3 程序调试和程序测试的区别

程序调试是由开发人员解决BUG的过程；而程序测试是测试测试人员找到BUG的过程。这是这两者的根本区别，前者的目标是解决问题，而后者的目标是发现问题，通过不停的对程序进行测试和DEBUG的过程，使程序能如需求文档那样顺利流畅的运行，最后进行软件发布。所以程序调试和程序测试的最终目的就是为了使开发的软件能正常顺利的运行发布。

一般来说，非正式测试以后（确认程序可以正常的编译运行）就是测试和调试过程，但是BUG发现的时机不仅仅是在正式的测试阶段由测试工程师测试出来，它也会在平常的使用过程中，被开发人员，或者其他人员发现，而发现的方式也是多种多样。

而无论是由谁，以任何方式方式发现的问题，最总都会转回到负责该模块的开发人员手中，由开发人员通过程序调试的方式来找到这个问题出现的原因，并且解决掉这个问题，然后该问题的修复结果会以相同的方式返回到发现人的手里进行验证。如果这个问题是由测试人员发现的，他们会以一个正式的项目管理的方式将这个问题通知给开发人员，而开发人员在解决掉该问题以后放在正式的版本中，然后再通知测试人员验证该问题是否继续存在。

这就是程序测试和程序调试的联系，程序测试如果出现问题，则紧接着就会有程序调试来解决问题。

## 2.2 程序调试的目的

### 2.2.1 为什么要进行程序调试

程序员在完成一个项目时，大致会经历需求定义、软件设计、程序编码、程序测试、和程序调试这几个阶段，如图：



程序调试的根本原因就是为了找到程序出现的错误，然后解决掉。而程序出现错误的原因有很多，可能是主观上程序员没有注意到某个细节；可能是程序过于复杂，程序没有通盘考虑到这个情况；也可能是外界的环境不同导致；也可能是代码版本管理出了问题；凡此种种有客观的，主观的原因，都会导致问题的出现。所以想要解决这些问题，就需要根据程序表现的现象，一步步追踪到根本原因处，然后采用合适的方案来解决。

为什么要程序调试，就是为了准确的找到问题出现的原因，并且修正。BUG在整个软件开发的过程中都是一直被伴随着的状态，所以程序调试也是贯穿整个开发周期的始终。

### 2.2.2 什么时候进行程序调试

发现BUG的时机，下一步就是进行程序调试的时候。根据上图所示，BUG发现的时机是在程序可以正确启动以后，和程序测试阶段，然后接下来就会转回到程序的开发人员手中进行程序调试过程。但是BUG发现的时机不单单只有上述的两个阶段才会发生，只有该程序是在使用的状态过程中，都有可能被自己人，或者是其他任何人，以各种各样的方式发现。

但是在项目的正式发布之前，使用该软件的只有相应的开发人员和测试人员，此时程序调试的时机就在开发与测试过程中发现BUG后进行程序调试。

### 2.2.3 程序调试的步骤

如果开发人员被告知有一个问题，那么他首先应该做些什么去解决这些BUG。做些什么的这个过程，其实就是程序调试的步骤。

程序调试在软件开发过程中的流程大概为下图所示：

****

无论BUG是开发人员自己自测出来的，还是其他人发现的，其主要的调试步骤如下：

1. 复现BUG；
2. 程序调试
3. 确认程序的执行过程（测试）
4. 程序行为与预期相同，则停止，程序行为与预期不同，跳转到步骤2.

而本文就是来介绍步骤2中的各种调试方法和经验。

# 程序的知识准备

## 3.1 操作系统相关知识

### 3.1.1 字节序

字节序，顾名思义就是字节的排序顺序。

首先解释一下什么叫字节。大多数计算机都使用8位的块来作为最小的可寻址单位，这个8位的块，就被称为字节。从C语言角度来说，就是每8位来表示一个ASCII码，然后以此为单位组成更多的内容。

如果一个多字节的对象保存在机器的内存中，机器如何判断哪个字节在前哪个字节在后，这个问题产生了字节序。比如在32bit的操作系统中，int类型的数据占了4字节，假设一个值为0x12345678的int型变量x地址为 0x100，那么x的值就会被存储到地址为 0x100,0x101,0x102,0x103 这四个字节的内存中，但是从哪里开始放呢，是将 12、34、56、78 依次放到 0x100、0x101、0x102、0x103 地址的内存中还是依次放到 0x103、0x102、0x101、0x100 的四个地址中呢。

以上两种存储多字节对象的方式，以最低有效位（0x78）在前（低地址-0x100）为小端法，在后（高地址：0x103）大端法。

对于大多数程序员来说，机器使用的字节顺序其实是无需关心的。只要写入的字节规则和读取的字节规则保持一致，换句话说，只要同一台机器或者是使用相同字节序的机器都不会因为字节序的不同产生问题。但是当两台字节序不同的机器进行数据传输，那么对于同样的内存的数据就会因为字节序的原因产生错误。一个最常见的问题就是当一个采用小端法的机器发送的数据被大端法的机器收到时，就会出现字节反序的错误。所以为了避免这类问题，网络应用程序都需要按照已建立好的字节顺序进行转换。

### 3.1.2 寄存器

寄存器是中央处理单元内部能够存储参与运算的数据和结果的时序逻辑电路。计算机本质是一个存储数据并且计算处理数据的机器。其中存储数据的功能就在磁盘、内存和寄存器中体现。当CPU需要进行数据计算时，会预先将数据从硬盘读取到内存中，然后再把即将要使用的数据从内存读取到寄存器中。寄存器，又被称为缓存，是容量最小，读写速度最快的存储数据的设备，但是它的造价是最昂贵的。内存和磁盘对应的读写速度和造价依次下降，容量依次增大。

寄存器分为通用寄存器组、指针和变址寄存器、段寄存器、指令指针寄存器、标志寄存器。

**3.1.2.1 通用寄存器**

通用寄存器组包括AX、BX、CX、DX 4 个寄存器，其中每个寄存器根据CPU迭代的版本，%ax 是16位的寄存器，%eax 是32位的寄存器，%rax 是64位的寄存器；BX、CX、DX系列的寄存器同理也有16、32和64位的寄存器迭代版本。

AX：累加器，主要存储返回值；16位的是%ax，有AH、AL的高8位和低8位区分；32位的是%eax；64位的是%rax；

BX：基址存储器，主要被调用者保存；16位的是%bx，有BH、BL的高8位和低8位区分；32位的是%ebx；64位的是%rbx；

CX：基数寄存器，保存函数调用的第四个参数，16位的%cx；32位的%ecx;64位的%rcx；

DX：数据寄存器，保存函数调用的第三个参数；16位的%dx；32位的%edx;64位的%rdx；

**3.1.2.2 指针和变址寄存器**

BP（BASE Pointer Register）：基址指针寄存器，被调用者保存

SP（Stack Pointer Register）：堆栈指针寄存器，保存栈指针；

SI（Source Index Register）：源变址寄存器，保存函数调用的第一个参数；

DI（Destination Index Register）：目的变址寄存器，保存函数调用的第二个参数。

**3.1.2.3 段寄存器**

CS：代码段寄存器

DS：数据段寄存器

SS：堆栈段寄存器

ES：附加段寄存器

**3.1.2.4 指令指针寄存器**

%rip：程序计数器，通常被称为PC，记录的是将要执行的下一条指令在内存中的地址；

**3.1.2.5 标志寄存器**

标志位寄存器也被称为程序状态位寄存器；标志位寄存器是16bit的寄存器，其中用来存放状态标志和控制标志的位数有9位。6位状态标志，分别是CF、PF、AF、ZF、SF和OF，用来记录程序运行的状态信息。控制位有3位，IF、DF、TF，用来CPU的控制。

**3.1.2.6 工作原理**

一个触发器能存放一个二进制代码， 若干个触发器和门电路组成一个寄存器。其中若干个的数量可以分为8，16，32，64。通常一个 x86-64 的中央处理单元包含一组16个能存储64位值的通用目的寄存器，而寄存器主要是用来存储整数数据和指针。

寄存器具有接受数据、存放数据和输出数据的功能。当寄存器收到“存入脉冲”时接受数据，得到“读出”指令时，将保存的数据输出。

寄存器存取数据有并行和串行两种方式，将n位二进制一次写入或者读出的方式被称为并行方式；将n位二进制以每次1位的分为n次存入寄存器或者从寄存器中读出的方式被称为串行方式。并行方式只需要一个时钟脉冲就可以完成工作，但是需要n条输入和输出数据线，穿行方式需要n个时钟才能完成n次的写入或者读取操作，但是只需要一根输入或者输出数据线，传输线少，适用于远距离传输。

### 3.1.3 地址空间

操作系统将内存视为一个非常大的字节数组，这个字节数组就被称为虚拟内存。内存的每一个字节都是由唯一的一个数字，该标识被称为地址。

地址是一个非负整数地址的游戏集合。在一个带虚拟内存的系统中，如果该系统是64bit，那么CPU就从一个有N=2^64个地址的地址空间中生成虚拟地址。这个地址空间被称为虚拟地址空间。一个系统还有一个物理地址空间，对应的大小是物理空间的字节数。

操作系统会将虚拟地址空间分割为虚拟页，作为磁盘内容拷贝到内存中的存储单元，这样每次从磁盘拷贝数据到内存中，直接拷贝一整页内容，而无需每次需要什么就去磁盘找然后拷贝。因为去磁盘拷贝数据的过程是非常耗费时间的。为了实现这个拷贝一个虚拟页的想法，物理空间地址也会遵循虚拟页的大小，分割成一个个物理页。操作系统通过内存管理单元将物理页和虚拟页进行映射，进行页表替换等操作。

### 3.1.4 数据类型

基本的数据类型包括字节（8bit）、字（16bit）、双字（32bit）、四字（64bit）。

**3.1.4.1 整数数据类型**

在C/C++中，整型的数据有：一个字节的char类型，一个字大小的short类型，一个双字大小的int类型，双字大小的long 类型（long类型具体的长度需要根据位数具体决定）；

上述的所有整型类型都支持无符号整数和有符号整数。无符号整数从0 ~ 2^（所占位数）-1,有符号整数表示的范围为 -2^（所占位数-1） ~ 2^（所占位数-1）-1，比如1字节的无符号 char，unsigned char 能表示的范围就是 0 ~ 2^8-1；有符号位的char， signed char 能表示的范围为 -2^7 ~ 2^7-1

**3.1.4.2 浮点数据类型**

单精度浮点数 float 占双字大小，双精度浮点数 double 占四字大小。这些数据类型的数据格式对应于 IEEE754 标准。

单精度的浮点数（32 bit）的精度是 24 bit，双精度的浮点数（64 bit）的精度为 53 bit。

**3.1.4.3 其余数据类型**

除了整型数据类型和浮点数据类型以外，CPU还会有一些其他的基础类型，比如指针数据类型，字符串数据类型等等。具体的数据类型需要根据不同的操作系统和编译器区别决定。

### 3.1.5 堆栈

这一小节介绍的堆和栈的知识主要是程序在计算机中数据存储的堆区和栈区。栈区和堆区都是程序用来存放数据的内存之一。

**3.1.5.1 堆区**

堆区是用来给程序员自主的分配空间使用；并且申请和释放都是由程序员控制，内存泄露的大部分原因也是在堆区中申请了空间后忘记释放导致；

**3.1.5.2 栈区**

栈区是由操作系统分配和释放，主要是用来存储和程序的函数调用有关的一系列的参数。包括函数参数，跳转地址，返回地址和一系列局部变量等。栈区的数据是先进后出的方式。

## 3.2 程序运行相关知识

### 3.2.1 程序的编译链接过程

从源码的.c或者.cpp文件到执行程序中，需要经过编译和链接两个过程，然后可执行程序被加载到计算机内存中等待CPU分配资源运行。其中编译整个过程的shell命令如下：



**3.2.1.1 编译过程**

编译过程可以分为两个阶段，编译和汇编。

编译阶段就是读取源程序的字符流，然后对其进行词法和语法分析，再将高级语言指令转换成对应的汇编代码，最终生成目标代码。

编译器完成两部分工作，一是编译预处理，读取源码后对#开头的伪指令和特殊符号进行预处理。二是优化处理，不仅对编译技术进行优化，还需要根据不同的硬件条件对中间代码进行优化，目的是减少对内存的访问次数。

预处理的伪指令包括宏定义指令如#define、#undef；条件编译指令如#ifdef、#ifndef、#else、#elif、#endif；头文件包括用户头文件和系统头文件，特殊符号如LINE、FILE等。预编译阶段主要完成的工作是对源程序的替代生成一个没有宏定义、条件编译指令、没有特殊符号的输出文件。

二是编译优化阶段，编译阶段只要是对预编译的输出文件做词法分析和语法分析，经过分析后确认所有的指令都符合语法规则后，生成等价的中间代码或者汇编代码。优化阶段涉及到两部分，一部分针对的是编译技术有关，做的工作是删除公共表达式、循环优化、复写传播、以及无用赋值删除等，这种优化不依赖与具体的计算机；另一部分则是针对机器的硬件结构优化，主要是考虑如何充分利用机器的各个硬件寄存器以达到尽可能少的访问内存以及更高更快的执行目标代码的效果。

经过编译器生成的汇编代码是以.s结尾的中间文件形式保存的。

汇编过程是就是将汇编代码语言翻译成目标机器指令的过程，目标的机器语言代码被存放在以.o为后缀的目标文件中。

目标文件是由段组成的，一个目标文件至少包含两个段，代码段包含的主要是程序的指令，属性是只读；数据段存放了程序的各种全局变量和静态数据，属性是可读可写可执行的。其中可重定位文件的内部数据组织格式如下所示：

A picture containing night sky

Description automatically generated

在Unix环境下目标文件主要分为三种：

1. 可重定位文件：包含适合其它目标文件链接来创建一个可执行或者可共享的目标文件的代码和数据。
2. 共享的目标文件：该文件存放了适合上下文链接的代码和数据。
3. 可执行文件：包含了一个可以被操作系统创建进程被执行的文件。

**3.2.1.2 链接过程**

汇编程序生成的目标文件，也就是上述介绍的第一种目标文件，它实际上是不能立即执行的。因为可能某个源文件的某个函数引用了另一个源文件中定义的函数或者变量；亦或者是在程序的某个地方调用了某个库文件中的函数等等问题。这些都需要经过链接程序的处理才能被执行。

根据目标文件中调用的函数或者变量所在的文件的类型不同，链接处理可分为两种：

1. 静态链接：静态链接的过程会将静态库中被程序调用的函数代码直接拷贝到最终的可执行程序中。这样程序在执行时所有的代码都会被加载到该进程的虚拟地址空间中。静态库实际上相当于一个目标文件的集合，其中每个文件含有库的一个或者一组相关函数的代码。
2. 动态链接：动态链接阶段链接程序会记录下需要目标程序会使用到的动态链接库的共享对象和其他少量信息。在可执行程序被执行时，动态链接库的所有内容都会被映射到该程序的虚拟地址空间中，动态链接程序根据可执行程序中相关的记录去找调用对应的函数代码。

最终生成的经过链接过程后会将可重定位的文件转换成可执行程序的目标文件，其内部的结构数据保存结构如下：



### 3.2.2 进程的虚拟地址

一个程序在计算机上运行时，最重要的就是两部分资源，CPU-执行指令做对应指令的运算，内存-保存要计算的数据和计算后的数据。现代操作系统在组织分配整个计算的内存资源时，使用了一个虚拟内存地址。虚拟内存是计算机系统管理内存的一种技术，通过使用一个虚拟地址和一个物理地址的映射关系的方法屏蔽物理内存碎片，对应用程序提供连续可用的内存。

操作系统会为每一个需要运行的程序建立一个进程来管理系统为该程序运行所需要的资源，包括但不限于CPU、运行内存。其中分配的运行内存以虚拟内存的方式存在，每个进程都拥有一个和操心系统位数有关的独立的虚拟内存空间。其中虚拟内存空间的分布大概如下：

A picture containing night sky

Description automatically generated

**用户空间：**

1. 指令放在.text段，只读数据放在 .rodata 段，该部分数据为只读；
2. .data 段用来存放初始化切不为 0 的数据；
3. .bss 段用来存放未初始化的以及初始化数据为 0 的数据；
4. .heap 用来存放程序中使用的分配内存的头指针；
5. Shared\_Library 用来存放程序用来链接的动态库；
6. 程序中的函数调用相关的数据都是存放在 stack 中，从高地址向低地址增长；

**内核空间：**

1. ZONE\_DMA：该区域大小为 0~16M，该区域的物理内存主要是为了专门为 I/O 设置的 DMA 使用。
2. ZONE\_NORMAL：大小约为 800M，进程空间的PCB（进程控制块）和内核空间的线程以及内核函数在运行时所依赖的栈空间存放在里面。
3. ZONE\_HIGHMEM：高端地址，做地址映射时使用。

### 3.2.3 程序的加载运行过程

经过编译链接过程以后，高级语言的代码就变成了计算机可以识别并进行运行的汇编指令集合的文本。当一个可执行程序被双击（被允许执行）时，首先会被操作系统进行加载。

**3.2.2.1 创建进程，分配虚拟内存空间**

程序被操作系统进行加载的过程，其实就是操作系统创建一个来执行该程序的进程的过程。操作系统会创建一个进程，然后读取可执行文件的文件头，将程序运行的必要信息保存到该进程的相关结构中：段表位置，程序入口，程序类型等；并且操作系统还会为根据该执行程序文件分配分配对应的虚拟地址空间；

**3.2.2.2 建立映射关系**

早期计算机的运行内存很小，不足以将一个程序加载进来，所以就采用将程序需要执行的指令一段一段的加载到计算机内存中运行的方法。于是就有了虚拟地址空间这个概念。虚拟地址空间和实际的物理地址需要有一个映射关系：即通过进程中保存的某一段的虚拟地址找到对应的物理地址保存的运行指令。通过不断的更替虚拟地址中的运行指令将内存无法一下子装载完的程序执行完成。

这个不断更替的过程就是虚拟地址到物理地址的映射关系来完成，而这个映射的关系的建立时间就是当操作系统将分配完虚拟内存空间以后就需要做的事。

**3.2.2.3 初始化程序参数**

完成虚拟地址和物理地址的映射以后，操作系统就可以载入可执行文件的剩下的数据段和代码段的数据了，比如：main函数中形参：char argc， char\* argv[]; .data,.rodata,.text段中的内容;程序运行的环境变量；对进程的栈空间和对空间进行初始：也就是初始化静态数据和全局变量部分；

**3.2.2.4 程序运行**

最后，将程序执行寄存器指向程序开始的地址，然后开始从 main 函数开始执行整个程序。程序运行时操作系统会创建一个进程，然后以进程控制块来表示该进城运行的一些必要信息，其中进程控制块主要包含以下信息：

1. 进程id，进程在操作系统中的唯一标识，以一个非负整数表示；
2. 进程的运行状态，就绪，执行，挂起，停止等状态
3. 进程切换时需要保存和恢复的一些CPU寄存器
4. 描述虚拟地址空间的信息
5. 描述控制终端的信息
6. 当前进程工作的目录
7. umask 掩码信息
8. 文件描述符，主要是一个指向文件结构体的指针数组
9. 信号有关的信息
10. 进程所属的用户id和组id
11. 会话和进程组

进程控制块在操作系统中是以一个数据结构实现的，该结构名为 task\_struct，保存在 **/usr/src/linux-headers-4.4.0-96/include/linux/sched.h 文件中**

## 3.3 汇编语言相关知识

本章主要是介绍了汇编语言基础的命令用法，以便于我们在调试时如果遇到了需要看汇编语言的部分可以直接理解。此外因为不同的平台汇编语言也不尽相同，我们使用的环境基本以 x86\_64 平台下的汇编（以最常使用的例子）。

### 3.3.1 汇编语言语法

#### 3.3.1.1 寄存器

Table

Description automatically generated

%rax：保存函数返回值

%rbx：被调用者保存

%rcx：保存函数参数中第四个参数值

%rdx：保存函数参数中第三个参数值

%rsi：保存函数参数中第二个参数值

%rdi：保存函数参数中第一个参数值

%rbp：被调用者保存

%rsp：栈指针

%r8：保存函数参数中第五个参数值

%r9：保存函数参数中第六个参数值

%r10~r13：被调用者保存

以 %rax 寄存器为例，如果返回值是64bit的话，汇编语言会显示该寄存器的名字为 %rax，但是如果返回的类型为 8 bit 的话，则会显示 %al。其他寄存器的名字也可以以此类推。

#### 3.1.1.2 栈指针

栈是一种先进后出的数据结构，该结构在函数调用中发挥着重要的作用：当调用一个函数时，会先将返回地址入栈，再将函数参数入栈，再存储临时变量，完成函数的调用。

在程序运行中，上述的函数调用过程的实现是由一个栈顶指针实现的，也就是 %rsp 指针。栈有两个操作，分别是进栈和出栈。进栈：将栈顶指针向下移动指定字节，使得栈容量增大，然后再将对应的数据保存到栈顶。出栈：向上移动指定的直接即可。

为什么栈顶指针向下移动就是进栈操作：因为栈从增长方向是从大都小的。所以程序运行的栈空间是大小限制的；

#### 3.1.1.3 PC寄存器

PC（Program counter）寄存器，保存的是CPU下一个要执行的指令的地址，是最重要的寄存器之一，通常使用 %rip 来表示该寄存器。调整指令和函数调用就是通过PC寄存器来实现非顺序执行指令的功能的。

PC寄存器的值不会显示的出现在汇编代码中。

#### 3.1.1.4 指令

计算机本质就是计算并保存计算的结果。其中计算的实现主体是CPU，而实现的方法是通过将CPU的操作转化为执行指令。

指令的格式是由一个操作码和 0~2 个操作数。操作码指的是要将操作数做什么操作，比如无操作数的地址跳转、一个操作数的自增自减、两个操作数的相加减等；而操作数就是要对什么值进行操作。指令的长度是不固定的，短则一两个字节，长则十五个字节。

#### 3.1.1.5 操作数

操作数可以是立即数、寄存器、和内存地址。这三种类型的操作数本质会指向对应的类型所保存的数值。

$5：立即数

%rax：寄存器

0xf7：内存地址

(%rax)：内存地址，内存地址存放在寄存器里

0xf7(%rax,%rbp,4)：内存地址

#### 3.1.1.6 操作码

操作码分为算数逻辑类、数据传输类、控制类等；

1. 算术和逻辑指令操作码：对数值进行计算操作的类  
   addq $3,%rdi //add 表示相加，$3为第一个操作数，并且第一个操作数是源操作数，%rdi，是第二个操作数，也是目的数。该行汇编代码的意思是将立即数加到寄存器%rdi中。  
   add 后面跟的 q 表示操作数大小，表示是个四字的；此外还有b-表示字节（8bit）、w-字（两字节，16bit）、l（双字-两字大小，32bit）  
   算数逻辑的操作码还有减、乘、异或、按位与、按位或、自增、自加、取反、非等；
2. 数据传输类  
   mov $bl，%al //表示把 %bl 寄存器中的值复制给寄存器 %al  
   pushq %rbp //将 %rbq 的值压入栈中；即将栈顶指针寄存器的 %rsp 的值减少 8，然后再将 %rbp 的值赋值给 %rsp 所指的内存单元；  
   popq %rsi // 表示将栈顶的 8 个字节的值弹出，并且赋值给寄存器 %rsi
3. 控制类  
   控制类的操作码通常是通过修改PC的内容值来控制程序执行的流向，包括无条件转移指令、条件转移指令、比较转移指令、循环转移指令、子程序调用和返回指令、空操作指令；

### 3.3.2 汇编语言实战

#### 3.3.2.1 汇编语言之简单main函数

##### 1）编写如下的 hello.c 文件



##### 2）使用汇编生成命令



##### 3）查看生成的汇编文件



##### 4）分析步骤 3 的汇编代码

①：%rbp 寄存器值入栈

②：%rsp寄存器值赋值给%rbp

③：main 函数的第二个参数入栈；

④：main 函数的第一个参数入栈；

⑤：调用printf函数

⑥：将返回值保存到%eax寄存器中；

⑦：栈顶元素出栈赋值给%rbp；和①对应

##### 5）关于调用者保存和被调用者保存

CPU的寄存器一般只有16个，除了几个特别用处外，比如 %rsp 保存栈指针、%rax 保存返回值、%rdi、%rsi、%rdx、%rcx、%r8、%r9 用来保存函数参数外，其他八个寄存器可以保存任何值。但是当函数调用时，父函数和子函数都可以使用这些寄存器，这样就就会导致后面的赋值覆盖掉前面的赋值，导致数据丢失的问题出现。而解决方案就是在子函数在调用之前将这些寄存器的值保存在栈中，待子函数使用完成后，再将栈中保存的值返还给寄存器。根据谁保存这些寄存器，可以将共同使用的寄存器分为“调用者保存”和“被调用者保存”。

“调用者保存”：即在调用子函数之前将父函数使用过的寄存器值压栈；“被调用者保存”：即在子函数内，对即将使用的寄存器压栈。

其中“调用者保存寄存器”有 %r10，%r11；“被调用者保存寄存器”有 %rbq、%rbp、%12-15；

#### 3.3.2.2 汇编语言之函数返回值

在 C/C++ 中，函数只能有一个确定的返回值，这个值可以是一个数，也可以是一个地址。这个返回值被保存在 %eax 寄存器中。

##### 1）代码



##### 2）汇编结果



#### 3.2.2.3 汇编语言之函数参数

函数参数指的是调用函数的参数数量，涉及到的寄存器主要包括 %rcx、%rdx、%rsi、%rdi 、%edx、%ecx、%r8d、%r9d 等寄存器。函数调用还涉及到调用者保存和被调用者保存使用相关的寄存器。

##### 代码



##### 汇编文件







##### 汇编代码解释

1. 当函数参数的个数不大于6个时，是使用寄存器保存函数参数值的；
2. 如果函数参数个数超过6个，那么多余的函数参数会入栈保持，然后使用的时候是通过将栈指针增加16的bit开始访问多余的函数参数；
3. 函数参数是从参数列表的右侧入栈，所以函数的栈指针(%rbp+0x10)访问的最右侧的参数，(%rbp+0x18)访问的第二右侧的函数参数。
4. 为什么是最右侧的函数参数距离函数指针的位置差16bit，相差的这个16bit分别存储了函数返回地址和栈指针(%rsp);

函数调用总结：

1. 函数调用之前，调用者需要将所需的参数保存到寄存器中，如果函数参数多6个，则多出来的参数按照从后往前依次压栈。注意参数大小要扩充为8的倍数；
2. 将返回地址压栈；
3. 将被调用函数地址传递给PC寄存器（%rip）；
4. 将调用者保存寄存器压栈；
5. 为局部变量分配内存空间；
6. 执行被调用者的指令；
7. 将返回值保存到 %eax 寄存器中；
8. 增加 %rsp 的值销毁当前的函数帧栈；
9. 执行 retq 指令，将返回地址赋值给 PC。

#### 3.2.2.4 汇编语言之逻辑运算

逻辑运算包括加减乘除位运算等，还有if，switch等跳转的汇编指令；

##### 代码





##### 汇编代码





##### 汇编代码解释：

1 ：add 两数想加

2 ：sub 两数相减（从目的寄存器中操作源目的数）

3 ：imul 两数相乘

4 ：cmpl 比较 (%rbp偏移 -0x8 的元素和 0 的关系)

5 ：jne 条件转移指令，转移的条件是 ne（not equal）不相等转移。当零标志 Z=0 时跳转，否则 Z=1时顺序执行下一条指令；该行表示的意思是，当cmpl 的结果为 0 时，即，比较的两个数不相等，跳转到 4e 这条指令，若相等，则顺序执行下一条指令。

6：jump 跳转指令，跳转到 55 这条指令执行。

7：idivl 表示有符号除法，除法之前会做一个 cltd 的操作，将 %eax 寄存器的值符号扩展到 32bit 到 %edx 寄存器。有符号除法将寄存器 %edx（高 32 bit）和 %eax（低 32bit）共 64 位数作为被除数，除数以指令的操作数给出。且会将商保存到寄存器 %eax 中，余数保存到 %edx 中。

8：and 表示与操作；

9：or 表示或操作

10：xor 表示异或操作

11：jle 表示小于等于，如果为真，则跳转，反之顺序执行；

# 调试的知识准备

高级语言会屏蔽平台的差异，但是对于调试工具来说，一般需要操作系统的工具来支持，这也就意味着不同的操作系统，所使用的调试工具是不同的。对于一般的上层应用程序，调试可能用到调试工具机会不太多，因为以日志输出的形式就能满足大多数程序的调试需要。但是如果涉及到操作系统底层的数据调试，或者当日志输出出现问题时，就需要这些调试程序的登场来帮助我们跟踪数据，定位问题了。

在 Unix 家族中，以 Liunx 为例，主要使用的调试工具是 GDB,本篇文章的所以测试用例都是基于此环境。

## 4.1 GDB基础命令

### 4.1.1 断点

GDB 中关于断点的命令主要有三大类，breakinfo、watchpoint、catchpoint；

**4.1.1.1 breakinfo**

该命令侧重于根据一定的条件设置断点

|  |  |
| --- | --- |
| 命令 | 作用 |
| break | 下一个指令处设置断点 |
| break [file]:function | 在 file 文件的 function 函数入口设置断点 |
| break [file]:line | 在 file 文件的第 line 行设置断点 |
| break [+-] offset | 在当前偏移量 [+-] offset 处设置断点 |
| break \*addr | 在地址为 addr 处设置断点 |
| break ... if expr | 设置条件断点，条件表达式 expr 为真时 |
| info breakpoints | 查看断点 |
| ignore n count | 对编号为 n 的断点忽略 count 次 |
| clear | 清除所有断点 |
| clear function | 清除所有位于 function 内的断点 |
| clear [file]:line | 清除文件 file 的 line 行断点 |
| clear n | 删除编号为 n 的断点 |
| enable n | 启用编号为 n 的断点 |
| disable n | 禁用编号为 n 的断点 |
| save breakpoints file | 保存断点信号到 file 文件中 |
| source file | 从 file 文件中导入断点信息 |

**4.1.1.2 watchpoint**

break 类型断点的效果是，当程序执行到断点处会被 GDB 暂停住。而 watchpoint 的效果是当监控的表达式发生变化时则程序停止运行；

watchpoint 的实现有硬件、软件两种方式；硬件实现需要硬件部分支持，软件部分的时间则每一步都需要进行变量检查。

watchpoint 的命令如下：

|  |  |
| --- | --- |
| 命令 | 作用 |
| watch variable | 监控 variable 变量 |
| watch var1+var2 | 监控表达式值变化 |
| rwatch variable | 监控数据读变化，仅支持硬件实现 |
| awatch variable | 监控数据读写变化，仅支持硬件实现 |
| info watchpoints | 查看监控断点列表 |
| set can-use-hw-watchpoints 0 | 强制 watch 使用软件实现 |
| print &variable | 查看变量的内存地址 |
| watch \*(type\*)address | 监控类型为 type 的 address 地址 |
| watch -l variable | 指定 location 参数 |
| watch variable thread 1 | 当线程 1 修改 variable 值时中断 |

**4.1.1.3 catchpoint**

捕获断点，主要是监测信号的产生。

|  |  |
| --- | --- |
| 命令 | 含义 |
| catch fork | 调用fork时中断 |
| tcatch fork | 设置的断点只触发一次 |
| catch syscall ptrace | 为 ptrace 系统调用设置断点 |

### 4.1.2 命令行

|  |  |
| --- | --- |
| 命令 | 左右 |
| run arglist | 运行程序，arglist 是程序的参数列表 |
| set args arglist | 指定启动命令行参数 |
| set args | 指定空的列表参数 |
| show args | 打印命令行列表 |

### 4.1.3 程序栈

|  |  |
| --- | --- |
| 命令 | 作用 |
| backtrace [n] | 打印第 n 个帧栈 |
| frame [n] | 选择第 n 个帧栈，如果不存在则打印当前帧栈 |
| up n | 选择当前帧栈编号 +n 的帧栈 |
| down n | 选择当前帧栈编号 -n 的帧栈 |
| info frame [addr] | 打印当前选择的帧栈 |
| info args | 当前帧栈的参数列表 |
| info local | 当前帧栈的局部变量 |

### 4.1.4 多线程、多进程

**4.1.4.1 多进程**

GDB在调试多进程程序时，默认是只追踪父进程的。但是可以通过一定的参数命令去追踪和调试父进程和子进程。

|  |  |
| --- | --- |
| 命令 | 作用 |
| info inferiors | 查看进程列表 |
| attach pid | 绑定进程id |
| inferior num | 切换到指定进程进行调试 |
| print $\_exitcode | 显示程序退出时的返回值 |
| set follow-fork-mode child | 追踪子进程 |
| set follow-fortk-mode parent | 追踪父进程 |
| set detach-on-fork on | fork 调用时只追踪其中一个 |
| set detach-on-fork off | fork 调用时会同时追踪父子进程 |

在调试多进程时候，在默认情况下，除了当前调试的进程，其他进程都处于挂起状态，所以需要在调试当前进程的时候，其他进程也能正常进行，那么可以通过设置 set schedule-multiple on 即可。

**4.1.4.2 多线程**

默认调试多线程时，一旦程序中断在某点，所有线程都将暂停。如果此时再继续执行当前线程，其他线程也会同时执行。

|  |  |
| --- | --- |
| 命令 | 作用 |
| info threads | 查看线程列表 |
| print $\_thread | 显示当前正在调试的线程编号 |
| set scheduler-locking on | 调试一个线程时，其他线程暂停执行 |
| set scheduler-locking off | 调试一个线程时，其他线程同步执行 |
| set scheduler-locking step | 仅用 step 调试时其他线程不执行，用其他命令如 next 时其他线程执行 |

### 4.1.5 打印输出

通常情况下，在调试的过程中，我们需要查看某个变量的值，以分析其是否符合预期，这个时候需要打印输出变量。

|  |  |
| --- | --- |
| 命令 | 作用 |
| whatis variable | 查看变量类型 |
| ptype variable | 查看变量详细的类型信息 |
| info variables var | 查看定义该变量的文件，不支持局部变量 |

**4.1.5.1 打印字符串**

使用 x/s 命令打印 ASCII 字符串，如果是宽字符字符串，需要先看宽字符的长度 print sizeof(str)。如果长度是 2，则使用 x/hs 打印；如果长度是 4，则使用 x/ws 打印。

|  |  |
| --- | --- |
| 命令 | 作用 |
| x/s str | 打印字符串 |
| set print element 0 | 打印不限制字符串长度或不限制数组长度 |
| call printf(“%s\n”,xxx) | 打印不会含有多余转移符的字符串 |
| printf “%s\n”, xxx | 同上 |

**4.1.5.2 打印数组**

|  |  |
| --- | --- |
| 命令 | 作用 |
| print \*array@10 | 打印从数组开头连续10个元素的值 |
| print array[60]@10 | 打印 array 数组下标从 60 开始的 10 个元素，即第 60~69 个元素 |
| set print array-indexes on | 打印数组元素时，同时打印数组的下标 |

**4.1.5.3 打印指针**

|  |  |
| --- | --- |
| 命令 | 作用 |
| print ptr | 查看该指针指向的类型及指针地址 |
| print \*(struct xxx\*)ptr | 查看指向结构体的内容 |

**4.1.5.4 打印指定内存地址的值**

使用 x 命令打印内存的值，格式为 x/nfu addr，以 f 格式打印从 addr 开始的 n 个单元长度为 u 的内存值。

|  |  |
| --- | --- |
| 命令 | 作用 |
| x/8xb | 以16进制打印数组 array 的前 8 个 byte 的值 |
| x/8xw array | 以 16 进制打印数组 array 的前 16 个 word 的值 |

**4.1.5.5 打印局部变量**

|  |  |
| --- | --- |
| 命令 | 左右 |
| info locals | 打印当前函数局部变量的值 |
| backtrace full | 打印当前栈帧各个函数的局部变量值，可以缩写为 bt |
| bt full n | 从内到外显示 n 个栈帧及其局部变量 |
| bt full -n | 从外到内显示 n 个栈帧及其局部变量 |

**4.1.5.6 打印结构体**

|  |  |
| --- | --- |
| 命令 | 作用 |
| set print pretty on | 每行只显示结构体的一名成员 |
| set print null-stop | 不显示’\000’这种数据 |

### 4.1.6 函数跳转

|  |  |
| --- | --- |
| 命令 | 作用 |
| set step-mode on | 不跳过不含调试信息的函数，可以显示和调试汇编代码 |
| finish | 执行完当前函数并打印返回值，然后触发中断 |
| return 0 | 不再执行后面的指令，直接返回，可以指定返回值 |
| call printf(“%s\n”,str) | 调用printf函数，打印字符串 |
| print func() | 调用func函数（call 命令也行） |
| set var variable=xxx | 设置变量variable的值为xxx |
| set {type}address = xxx | 给存储地址为 address，类型为 type 的变量赋值 |
| info frame | 显示函数堆栈信息（堆栈帧地址、指令寄存器的值等） |

### 4.1.7 其他

**4.1.7.1 图形化**

tui 为 terminal user interface 的缩写，在启动时指定参数 -tui，或者调用时使用 ctrl+x+a 组合键，可进入或退出图形化界面。

|  |  |
| --- | --- |
| 命令 | 作用 |
| layout src | 显示源码窗口 |
| layout asm | 显示汇编窗口 |
| layout split | 显示源码和汇编窗口 |
| layout regs | 显示寄存器+源码或者汇编窗口 |
| winhegiht src+5 | 源码窗口高度增加 5 行 |
| winheight asm -5 | 汇编窗口高度减少 5 行 |
| winheight cmd + 5 | 控制台窗口高度增加 5 行 |
| winheight regs – 5 | 寄存器窗口高度减少 5 行 |

**4.1.7.2 汇编**

|  |  |
| --- | --- |
| 命令 | 作用 |
| disassemble function | 查看函数的汇编代码 |
| disassemble /mr function | 同时比较函数源代码和汇编代码 |

**4.1.7.3 调试和保存 core 文件**

|  |  |
| --- | --- |
| 命令 | 作用 |
| file exec\_file \*#\* | 加载可执行文件的符号表信息 |
| core core\_file | 加载 core-dump 文件 |
| gcore core\_file | 生成 core-dump 文件，记录当前进程的状态 |

## 4.2 GDB调试步骤

### 4.2.1 生成可调试的执行程序

如果想要对可执行程序进行编译，那么需要进行debug版本的编译。在执行编译命令的时候，需要加上 -g 参数。例如：



这样就可以对 debug 这个可执行程序进行GDB调试了。

### 4.2.2 设置生成可调试的core文件

如果程序在执行时崩溃，可以生成可调试的core文件，可以对该core文件进行gdb调试来分析程序崩溃的原因。如果想要生成core文件，需要对设置一下。

**1）检查 core 文件大小限制：**



如果第一行 core file size 是 0 的话，表示未打开程序崩溃时生成 core 的设置；

**2）打开生成 core 文件设置**



设置生成的 core 文件最大值值为 1024k，也可以设置为 unlimited。

**3）修改 core 文件生成目录**



### 4.2.3 调试程序

**1） gdb 运行程序**

1）调试可执行程序，执行 gdb program\_name，

2）需要调试 core 文件，则使用 gdb [program\_name] [corefile\_name];

3）调试已经运行的程序，ps aux | grep [program\_name] 找到运行的进程，然后使用 gdb attcach [program\_id]

**2）查看源码**

设置断点前需要知道要设置的代码行数，你可以去源码 CPP 文件去查，或者使用 list line\_number ，查看 line\_number 行附近的源码

**3）设置断点**

使用上一节的断点命令去设置断点，让程序在特定条件下暂停，然后去查看你想要了解的值；

**4）运行程序**

设置断点后就可以执行程序了，运行程序的命令有三种，分别是 start、run、continue，都可以开始执行程序，开始调试，其中：

run（r）：运行程序（如果断点会执行到断点处，如果没有断点，会按照正常程序运行下去）；

start：运行程序（到main函数的第一行中）

step（s）:单步执行程序(能执行到函数内部)；

next（n）: 单步执行，但是不会进入函数

continue（c）：执行到断点处

根据需要，选择下一步进行哪种的程序运行。

**5）输出打印，查看变量**

在断点处使用对应的命令去查看和输出你想要了解关心的数值，查看你关心的变量或者结构体是否是如你所预期的那样。如果是多线程或者多进程的程序，也可以使用上述中有关多线程和多进程的命令进行对应的操作。

在使用命令调试的过程，还可以将你调试的命令和对应的输出保存到文件中，以便回头重新查看前面的调试过程。其中，设置保存操作日志的命令为：

1. 打开关闭日志文件的命令（日志默认保存到运行程序目录下的 gdb.txt中）：

set logging on/off

1. 保存日志文件为指定文件名中：

set logging file <filename>

上述是使用GDB调试的方法，具体使用GDB去调试不同的程序和问题，需要我们具体去分析。

# 简单的程序调试实践

## 5.1程序调试实践之调试main函数

### 5.1.1 GDB启动程序调试过程

C++ 程序启动时是可以支持 main 函数带参数列表的，那么在调试时，GDB就使用启动命令来支持该功能。

#### 代码

#### 2. 调试过程



#### 3. 补充说明

1) run [arglist] 以 arglist 参数列表运行程序（会重新运行程序）

2) set args [arglist] 随时修改参数列表的值，但是只有重新运行才会生效，并且重新运行时不能重新设置参数列表；

3) set args，也就是命令 set args [arglist] 中的 [arglist] 参数为空

4) show args 等同于代码里的 for 循环打印。

### 5.1.2 breakpoint断点和打印调试过程

设置断点是调试的必须步骤，通过断点让程序停留在某一行代码或者在某一条件下暂停，才能更有效率的查看程序的变量赋值是否符合预期，从而找到问题所在。断点暂停主要有三种，其一是运行到某一行暂停，其二是满足某一条件后暂停，比如监听某个变量变化，或者某个变量等于某个值，其三是捕获信号；

设置断点的目的就是为了查看变量的值，确认程序暂停时关键变量的值是否符合预期。其中可以打印的变量除了简单的内置数据类型以外，还有指针，数组，地址，字符串，结构体等，C++中的话，还需要打印一些STL容器等特定的数据结构。

#### 代码



#### 调试过程





#### 补充说明

1. 取消显示的命令是 undisplay [number]，其中 [number] 是从 info display显示出来的显示标号。
2. 如果不先指定gdb的输出文件，默认的输出文件是当前目录下的 gdb.txt

### 5.1.3 watchpoint 断点调试过程

watchpoint是一种特殊的类型的断点。触发的条件是当设置的表达式的值改变的时候，会执行暂停程序的动作，自动停留在表达式值变化后的下一行。

#### 1. 代码

#### 2. 调试过程



#### 3. 补充说明

1）监控指针 p 和监控 \*p 是两回事，需要区分一下

2）监控内容涉及到局部变量时，需要将程序运行到该有效域内；或者可以尝试监控内存地址间接设置断点： watch \*(type\*) address

### 5.1.4 catchpoint 断点调试过程

catchpoint 捕获断点，主要是检测信号的产生。主要是event 事件

其中包括有

1. throw：C++抛出异常
2. catch：C++捕获到异常
3. exec：exec被调用
4. fork：fork函数被调用
5. vfork：vfork函数被调用
6. load：加载动态库
7. unload：卸载动态库
8. load libname：加载名为 libname的动态库
9. unload libname：卸载名为 lianame的动态库
10. syscall [args]：调用系统调用，args可以指定系统调用号，也可以是系统名称

#### 1. 代码



#### 2. 调试过程



#### 3. 补充说明

1. catch throw [exception] 当程序抛出 exception 指定类型异常时，程序停止执行。expection 可以缺省，如果不指定的话，则会在所有throw处暂停执行程序

2. catch catch [exception] 同1描述，区别是程序捕获异常时就会暂停执行；

3. catch load [regexp]: regexp 表示动态库的名称，load命令是加载动态库时程序暂停；与之相对应的是 unload 命令。

4. catch 的捕获事件是上述的十种情况，用法等同于1，2，3的throw，catch，和load；

5. 在catch暂停时，可以使用 up查看当前程序暂停在哪一行代码处。

### 5.1.5 输出打印调试过程

调试过程就是输出打印查看在程序运行中变量的值是否于设计时设想的一样。

#### 1. 代码



#### 2. 调试过程



#### 3. 补充说明

1. print 的输出打印和 display 的区别，前者是当次打印当次输出；后者是只有暂停，就会自动有打印显输出；

2. 基本所有的C++类型，都可以有print打印内容，包括但不限于基础的数据类型，如int，char，float，自定义的数据类型，struct，class，C++的高级数据结构，如vector，map等数据结构

3. set print array-indexes on 是设置数组打印时显示出序号

4. print 可以缩写为 p

## 5.2程序调试实践之调试函数

### 5.2.1 对程序的函数进行调试

#### 1. 代码



#### 2. 调试过程



#### 3 补充说明

1. 调试函数，就是跟踪函数栈帧，主要看的观察的是该函数的函数实参，函数局部变量，其中栈帧使用的命令是 backtrace 和 frame；
2. bt（backtrace的缩写）显示的是函数的调用次序。一般序号为0的函数是当前函数，序号最大的函数是main。显示的内容包括栈帧序号，调用的函数名，函数实参，函数名所在的文件和行号。除此之外，非首行还显示调用的函数的函数地址。bt full 可以以信息更全，格式更工整的显示以上信息。frame 显示内容和bt大同小异，info frame会额外显示保存函数地址的寄存器名称以及一些其他信息。
3. bt 的调用显示可以理解为树的结构，从main函数开始根节点，一直调用到当前函数的叶节点。
4. finish 函数的作用是结束当前调用的函数，并且打印出返回值；
5. call 和 print 和调用函数并且打印出调用函数的返回值，并且调用的函数不需要在本函数中使用或者声明。
6. info locals 可以打印出当前函数的整个局部变量，info args 可以打印出当前函数传入的实参。
7. 需要注意的是，指针类型的打印都是一个地址，如果想要知道这个指针的值，需要解指针，或者如果是一个struct 是真的话，需要做对应的打印命令。

## 5.3程序调试实践之调试静态库

### 5.3.1 静态库的生成

#### 1. 静态库的编译命令

1）生成汇编文件：g++ -c [static\_file\_name],

g++ -c add.cpp (默认生成一个 add.o 文件)

2）生成静态库：ar -crv lib[static\_lib\_name].a [static\_file\_name].o

g++ -crv libadd.a add.o

3）使用静态库：在生成可执行成的编译命令中，加入 -L [path\_static] -l[static\_lib\_name]

g++ main main.cpp -L . -ladd

### 5.3.2 静态库的调试

#### 1. 代码



#### 2. 编译过程

1. 生成汇编文件：g++ -c static\_lib.cpp static\_lib.o -g

2. 生成静态库: ar -crvs libstatic.a static\_lib.o

3. 生成可执行文件： g++ main.cpp -o main -L . -lstatic -g

#### 3. 调试过程



#### 4. 补充说明

1） 在汇编阶段，静态库的会和汇编生成的目标文件一起打包到可执行程序中。对于调试的过程来说，静态库的数据都在可执行程序中，当调试可执行程序时，可以直接进入到静态库的程序函数中，所以调试静态库的过程和普通程序的过程以及步骤是一样的；

2） 如果想要调试程序，必须在程序中生成可调试的信息，也就是说在生成动态库的汇编文件时，就需要加一个 -g 的参数，在生成可执行程序时，-g 的参数也不能少

## 5.4程序调试实践之调试动态库

### 1. 动态库的生成

#### 1. 代码

#### 2. 编译过程

1) 生成动态库

g++ animal.cpp shared.cpp -fPIC -shared -o libshared.so

2) 生成可执行程序

g++ main.cpp -o main -L ./ -lshared -Wl,-rpath,./

3) 查看可执行程序的动态库链接文件依赖

ldd main

### 2. 动态库的调试

#### 3. 调试过程



#### 4. 补充说明

1. 动态库本质上也是一个二进制的文件，和普通的执行程序是一样的。也就说，动态库内部数据的过程和可执行程序执行的过程是一样的，区别在于可执行程序时调用动态库的这个过程是需要注意的；

2. GDB调试的过程都是需要有调试信息的，意味着如果想要调试一个二进制文件，就必须在文件生成的命令中加上 -g 参数；

3. catch load [shared] 当加载动态库时中断； info sharedlibrary 显示当前程序加载的动态库。

# 多进程程序调试实践

## 6.1 多进程程序编译运行

### 6.1.1. Linux 多进程介绍

在Linux中，多进程的实现过程如下：

当进程通过调用fork函数创建出一个子进程后，操作系统内核会将父进程的用户地址空间的内容复制一份给子进程，这样父子进程就拥有了各自独立的用户空间，当父进程修改一个变量的值时是不会影响到子进程中相同的变量的。

并且Linux为了提高效率，采用了COW(copy on write) 算法，当子进程被创建时，父子进程分享相同的地址空间，但是父子进程会在页面中有cow的标识以作区分，只有当父子进程执行写操作时，才会为子进程另外申请一个物理页，将父进程空间中对应的数据所在页的内容复制到该物理页中，然后再将该页映射到子进程用户地址空间的适当位置。

子进程会继承父进程的一些进程级的资源，比如父进程已经打开的文件描述符，父进程的代码区，和一些初始数据等；不同的是fork函数返回的pid在父子进程中拥有不同的值，所以会程序代码会根据不同的pid的分支语句父子进程有不同的执行方向；

当子进程完成的是代码中某一部分功能的话，当子进程的代码执行结束后应当使用exit或者rentu函数结束子进程。否则创建子进程的函数会一直被执行。

### 6.1.2. Linux 多进程编程

#### 1. 多进程相关API

**pid\_t fork()**

+ 头文件: unistd.h

+ 功能: 创建进程。父进程调用 fork 返回的是子进程的进程ID，子进程调用该函数返回的是 0。

**void exit(int status)**

+ 头文件: stdlib.h

+ 功能: 退出进程。退出时会调用 atexit 注册的函数，先注册的后调用，exit、

函数还会按需调用 fclose 函数关闭打开的文件流

**int atexit（void(\*func)(void)）**

+ 头文件: stdlib.h

+ 功能: 为进程注册退出时调用的函数

**void \_exit(int status)**

+ 头文件: unistd.h

+ 功能: 直接退出函数

// execl 函数族的声明都位于 unistd.h 中

**int execl(const char\* path, const char\* arg, …)**

+ 功能: 在 path 目录下的 arg 中第一个参数的 文件

+ 参数:

- path: 想要执行的文件的路径或者名称

- arg: 执行可执行文件所需要的参数列表，第一个参数为执行的程序名。第二个参数为程序执行所需要的参数列表，参数以NULL结束

+ execl("./multiexec","multiexec","arg1","arg2",NULL);

**int execlp(const char\* file, const char\* arg, …)**

+ 参数:

- file: 执行的程序名/文件名

- arg: 执行文件的参数

+ 功能: 在环境变量中查到名为 file 的执行文件，找到就执行，找不到就不执行

+ execlp("ps","aux",NULL);

**int execle(const char\* path, const char\* arg, …);**

+ 功能: 到指定环境变量中找可执行程序

+ 参数

- path: 可执行程序的路径

- arg: 传入到可执行程序中的参数

+ e.g. :

- char \*envp[] = {“env1”,”env2”,NULL};

- execle("./multiexec","arg1","arg2",NULL,envp);

**int execv(const char\* path, char\* const argv[]);**

+ 功能: 将可执行文件的参数保存到数组汇总

+ e.g. :

- char \*const argv[] = {“ps”,”aux”,NULL};

- execv(“bin/ls”,argv);

## 6.2 多进程程序调试实践

### 6.2.1. 简单版多进程实例

该实例生成一个简单的子进程，主要是使用 fork() 函数使用，并且可以使用 GDB 调试父，子两个进程

#### 1. 代码



#### 2. 调试步骤

1. GDB启动后，在执行程序之前，先查看当前的调试程序，和调试模式

+ show follow-fork-mode: 查看当前调试的进程是父进程还是子进程。GDB 模式是父进程

+ set follow-fork-mode [parent/child]: 调试进程的切换，切换为父进程或者为子进程

+ show detach-on-fork: on 表示单进程调试，off表示多进程调试

+ set detach-on-fork off: 多进程调试

2. 断点设置在 fork 函数处，然后执行程序

3. 此时查看当前的运行的进程

+ info inferior

+ 根据需要，切换对应的进程进行调试

+ inferior [number\_id]

4. 然后继续打对应的位置的断点，执行 c 命令运行即可。

5. 调试步骤样例：



#### 3. 补充说明

+ 如果你的调试代码目标明确的话，调试步骤的 3 也可以直接在 1 中做好准备。直接设置好调试的进程为父进程还是子进程，然后在对应的代码段处打上断点，执行运行的命令即可。

+ 断点设置的位置最好是 fork 的下一行，确保当前的进程已经生成了子进程，否则还需要在运行 s/n 执行该处代码。

### 6.2.2. 进阶版多进程实例

该实例主要是介绍，生成一个子进程去执行另外一个函数的方法。具体实现是通过 exec 函数簇来完成的。

#### 1. 代码





#### 2. 调试步骤

因为当前我们知道要调试子进程，所以可以在刚进入GDB页面时就开始设置对应的调试属性

+ GDB multiprocess

+ set follow-fork-mode child

+ catch exec // 捕获 exec 函数调用

+ r // 执行后会直接运行到exec调用处，即子进程的执行函数处

+ b main // 此时子进程的所有参数已经被替换成 multiexec 的对应的执行内容了

+ c // 执行 c 而不是 s/n 的原因是，在exec调用到 main 函数中会有一些不知道什么的代码还在执行，会有好几步需要执行，索性就在 multiexec 的 main 处打上断点，然后直接执行到 main 函数处

+ 下面的调试步骤和命令就和普通的程序调试一样了

3）调试样例

#### 3. 补充说明

1) 编译调试程序时都得 -g

2) 使用 exec 函数簇的时候，涉及到二维字符数组时，都需要在二维数组最后加上NULL，表示二维数组的结束，比如 execl 函数的最后一个参数，环境变量数组 const char\* envp[]

### 6.2.3. 升级版多进程实例

该实例主要是介绍多进程之间的通信方法，有父子进程之间的匿名管道，两个单独进程通信的命名管道、消息队列、共享内存、信号、信号量、套接字。在这些进程通信的方法中，除了信号量以外，其他方式都是普通的两个的进程，执行方式和调试过程都和其他普通进程没有区别。而对于信号来说，GDB有特定的信号来调试。

#### 1. 代码

#### 2. 调试步骤



#### 3. 补充说明

1）raise 发送的信号和其他kill，alarm 等发送信号的方式是类似的，调试方式也是相同的

2）info signal 和info handle 作用相同，都是列出显示的信号种类

3）调试信号的命令 如下：

handle [signal\_name] [nostop/stop] [noprint/print] [pass/nopass] [ignore/noignore]

4）设置好处理信号的动作以后，可以直接运行调试程序。

# 多线程程序调试实践

## 7.1 多线程程序编译运行

### 7.1.1 线程的基本概念

1. 什么是线程

线程，被称为轻量级进程，在 Linux 操作系统中，本质还是一个进程。因为在内核中并没有线程这个概念， Linux 将线程视为一个与其他进程共享某些资源的特殊进程。

线程没有独立的地址空间，但是有程序控制块。宏观概念上一个运行的程序至少是一个进程（也可以是多个进程一起执行该程序），一个进程可以生成很多个线程去执行该程序的代码片段。

线程有自己的线程ID，栈指针，上下文信息，寄存器值，一个进程的所有线程共享该进程的进程级资源，比如文件描述符表，进程地址空间，每种信号的处理方式，用户id和组id等。

2. Linux 内核线程实现原理

创建线程的底层函数和创建进程的底层函数是一样的，都是调用内核函数 clone，复制一份当前进程的进程控制块。但是线程中的PCB和进程的PCB中关于三级页面内容是相同，意味着线程是分享进程的地址空间的内容，包括但不限于页表和物理页面。

### 7.1.2 线程的相关函数

在多线程的程序中，同一进程的线程共享进程内的绝大多数资源，当一段访问这些共享资源的代码块，有可能被多个线程执行时，那么这块代码就被称作临界区。

当有个多线程并发的在临界区执行时，程序的执行结果就会出现不确定时，这种情况就被称为竞态条件。

1. 线程自身：

原型：pthread\_t pthread\_self (void)；

功能：获取当前线程的线程id

2. 线程创建：

原型：int pthread\_create (pthread\_t \*\_\_restrict \_\_newthread,

const pthread\_attr\_t \*\_\_restrict \_\_attr,

void \*(\*\_\_start\_routine) (void \*),

void \*\_\_restrict \_\_arg);

功能：创建一个新的进程

3. 线程退出：

原型：void pthread\_exit (void \*\_\_retval)；

功能：退出单个线程

4. 线程回收：

原型：int pthread\_join (pthread\_t \_\_th, void \*\*\_\_thread\_return);

功能：阻塞当前线程，等待指定的线程退出并获取返回状态

5. 线程分离：

原型：int pthread\_detach (pthread\_t \_\_th)

功能：将指定线程与当前线程分离，系统会在指定线程结束时自动回收其资源；

6. 线程取消：

原型：int pthread\_cancel (pthread\_t \_\_th);

功能：立刻取消指定线程

7. 线程同步：

原型：int pthread\_cond\_init (pthread\_cond\_t \*\_\_restrict \_\_cond,

const pthread\_condattr\_t \*\_\_restrict \_\_cond\_attr);

功能：使用 ATTR 属性初始化线程，如果是 NULL 的话，使用默认的值初始化

原型：int pthread\_cond\_destroy (pthread\_cond\_t \*\_\_cond);

功能：销毁条件变量

原型：int pthread\_cond\_signal (pthread\_cond\_t \*\_\_cond);

功能：唤醒一个等待条件变量的线程

原型：int pthread\_cond\_broadcast (pthread\_cond\_t \*\_\_cond);

功能：唤醒所有等待条件变量的线程

原型：int pthread\_cond\_wait (pthread\_cond\_t \*\_\_restrict \_\_cond,

pthread\_mutex\_t \*\_\_restrict \_\_mutex);

功能：阻塞线程，等待条件变量的唤醒

8. 线程互斥

原型：int pthread\_mutex\_init (pthread\_mutex\_t \*\_\_mutex,

const pthread\_mutexattr\_t \*\_\_mutexattr);

功能：初始化一个互斥锁

原型：int pthread\_mutex\_destroy (pthread\_mutex\_t \*\_\_mutex)

功能：释放一个互斥锁

原型：int pthread\_mutex\_lock (pthread\_mutex\_t \*\_\_mutex)

功能：加锁

功能：int pthread\_mutex\_unlock (pthread\_mutex\_t \*\_\_mutex)

功能：解锁

9. 补充说明：

线程同步使用的是条件变量，条件变量用来阻塞一个线程，等待条件符合后被唤醒后继续执行，通常和互斥锁配合使用，对某一个临界资源进行互斥的访问。

条件变量和互斥锁的使用过程是：等待线程中，先使用互斥锁加锁（pthread\_mutex\_lock），然后根据条件判断使用条件变量阻塞函数（pthread\_cond\_wait），然后是临界资源的访问，最后解锁（pthread\_mutex\_unlock）；激活线程中，先加锁（pthread\_mutex\_lock），临界区资源处理，然后发送条件变量（pthread\_cond\_signal）。

### 7.1.3 线程的简单实例

#### 1. 代码



#### 2. 补充说明

1）Linux下的线程编译需要加上 -lpthread

2）被线程调用的函数返回值和形参都是固定的 void\* 类型，传入和传入后可以强转，注意类型安全；

## 7.2 多线程程序调试实践

### 7.2.1 多线程基础程序调试

在多线程中，如果要起一个线程去执行某个函数代码，如果主线程需要该线程的返回值做处理的话，那么这个线程要么设置为 pthread\_join(),主线程等待该线程的返回值，如果不需要线程的返回值，则需要将该线程设置为 pthread\_detach(),无需关注该线程结束后资源的回收问题。另外如果将一些数据传递给线程和线程如何返回一些必要数据到主线程的情况，也将会在下方的实例中给出介绍。

#### 1. 代码



#### 2. 调试步骤



#### 3. 补充说明

1）和多线程有关的命令有 info thread，查看当前正在运行的线程；thread [id]，切换到指定的线程中；thread apply [id] [command] 可以在不切换线程的前提下，给其他线程发送调试命令；b [location] thread [id] if [expression]给指定线程打条件断点，条件可以为空；set scheduler-locking mode:mode 三种值，off：不锁定线程，任何线程都可以随时运行；on：锁定线程，只有当前线程或者指定线程运行；step：当单独执行某一线程时，其他线程不会运行，同时保证在调试过程中当先线程不会发生变化。但在该模式下执行 continue、until、finish命令，则其他线程也会执行，如果某一线程在执行过程中遇到断点，则GDB调试器会将该线程作为当前线程。

2）调试线程，切换线程有两种方式，一是直接在要调试的函数中打断点，GDB会在断点触发的时刻，切换到对应的线程；二是 thread [thread\_id] 直接切换到对应的线程，但是这种方法有可能在切换的中途会已经执行完对应的线程，当你切过去的时候调试不到自己的代码。

### 7.2.2 多线程线程锁程序调试

多线程的代码中，资源的竞争是绕不开的一个点。所以调试线程锁就至关重要了。本次代码实例中主要有线程互斥锁和线程的同步函数进行调试。

#### 1. 代码



#### 2. 调试步骤



#### 3. 补充说明

1)多线程的调试本质和普通的单进程单线程调试过程和设置命令一样。主要是找准出问题的某个函数，打断点然后跟踪调试。区别在于多线程要注意线程之间的运行推进顺序不当会导致的资源竞争出现的问题。

2）如果是使用 gdb attach [process\_id] 的方式进行调试，那么首先需要编译时加上调试的编译参数，然后使用 ps aux | grep [grogram\_name] 获取到运行的进程的进程号，然后使用 gdb attach 调试。如果权限不够，在 gdb 前加上 su 命令。后面的调试步骤就是普通的程序调试了。

3）关于线程之间的调试命令如下所示：

**info threads**：显示当前可调试的所有线程，每个线程会有一个GDB为其分配的ID，后面操作线程的时候会用到这个ID。 前面有\*的是当前调试的线程

**thread ID(1,2,3…)**： 切换当前调试的线程为指定ID的线程

**break [file\_name]:[line\_number]thread all**：（例：在相应函数的位置设置断点break pthread\_run1） 在所有线程中相应的行上设置断点

**thread apply[thread\_id1][thread\_id2] [command]**：让一个或者多个线程执行GDB命令command

**thread apply all [command]:**让所有被调试线程执行GDB命令command

**set scheduler-locking** **[command]** :设置线程是以什么方式来执行命令

其中 off 不锁定任何线程，也就是所有线程都执行，这是默认值

on 只有当前被调试程序会执行

**set scheduler-locking on step**： 在单步的时候，除了next过一个函数的情况(熟悉情况的人可能知道，这其实是一个设置断点然后continue的行为)以外，只有当前线程会执行

# 程序异常有关的调试实践

BUG分为两种，一种是程序正常执行到结束，或者一直在正常执行，但是执行的某些结果不符合预期，需要调试程序的执行过程，看数据的流转过程，来找出执行过程中不符合预期的地方，并修复。

另外一种是程序未能正常运行，比如程序崩溃，无响应，内存泄露等等，都是属于程序有问题的情况。本章就是用来介绍程序未能正常运行时调试情况。

## 8.1 与程序死锁有关的调试实践

在多线程中，会因为多个线程对于一些资源互斥的访问而使用锁，通常会使用互斥锁和条件变量对一种临时资源的访问进行控制。但是如果由于一些原因，比如使用的锁的个数是复数的时候，就会出现死锁的情况。

死锁的四个必要条件，分别是互斥条件，请求与保持条件，不可剥夺条件，循环等待条件。

### 8.1.1 死锁进程的调试

如果能直接调试发生死锁的进程，那么情况是比较简单的，如以下代码所示的实例。

#### 1.代码



#### 2.调试过程

**1）启动程序**。命令行启动调试程序，然后输入运行命令 r，因为是死锁，所以程序不会结束。为了查看程序处于的状态，可以按下 Ctrl+C 组合键，中断程序。



**2）查看函数调用栈**。查看当前程序暂停在哪里，使用 backtrace（bt）追踪函数的调用栈



**3）切换函数**。使用 frame 命令将当前函数的运行点切换到用户层代码。即 deadlock.cpp 的44行。



**4）查看线程**。44 行代码意思是在等待线程t1的返回。卡在等待线程返回，然后查看线程 t1 执行的代码中有加锁的代码，就应该意识到可能会存在死锁的情况。使用 info threads 命令查看当前进程的线程。



**5）切换线程**。步骤4中看出当前进程有3个线程，线程1运行main函数，此时需要使用 thread 切换到线程2和线程3查看这两个线程执行的函数。



**6）切换栈帧并查看源码**。在线程2的函数栈中，可以看到在 #2 处，调用了 doWork\_1() 函数，源码位于 deadlock.cpp 文件的第 17 行，那么查看该行代码。



**7）查看其他线程**。通过查看程序函数栈会发现，当前线程阻塞在了代码的第 17行，即 sleep，其实应该是第 18 行的代码无法执行到，所以表现在阻塞第 17 行。18 行的代码未被执行，说明无法申请到 mutex 互斥锁的使用权，猜测应该是另外一个线程获取到 mutex2。那么就继续查看线程3.



**9）总结**。如步骤 8 所示，线程 3 用于互斥锁 mutex2，正在申请 mutex1，而线程 2 拥有互斥锁 mutex1，正在申请 mutex2。发生死锁。

#### 3.补充说明

1）当程序在运行，一直没有得到自己想要的结果，但是能接受终止或者中断的信号，并且进程中有线程操作，就可能有死锁的可能。

2）调试死锁时，查看各个线程的是否阻塞在对应加锁的代码附近，或者阻塞附近有加锁的代码。

### 8.1.2 转储文件的死锁调试

如果可以直接调试源码的进程那是最好不过的，但是如果出问题的进程无法被直接调试，比如程序在客户那边运行出现问题，那么就可以使用将进程运行的数据转储出来，然后在本地加载调试。

#### 1. 代码

代码和8.1.1章节保持一致。

#### 2. 调试过程

1. 在客户环境中，先查看运行的进程的进程号



1. GDB 附加到进程。gdb attach [process\_id]，注意权限问题，如果附加进程的时候，有提示 Could not attach to process. 则需要使用 sudo 执行 gdb 的 attach 命令。



1. 生成 core 文件。使用命令 gcore [file\_name].core 生成 core 文件，然后分离进程，并退出。



1. gdb 加载 core 文件。上述步骤是模拟客户环境，让客户帮忙生成有问题的程序的core文件。接下来是通过 core 文件来分析程序的问题。首先使用 gdb 加载 core 文件。gdb 加载 core 文件的方法是：gdb [program\_name] [core\_file\_name].core



5）接下来的分析步骤就同 8.1.1 类同。此处不在赘述。

#### 补充说明

1. 使用 gcore 命令生成的 core 文件的目录默认是当前文件夹，如果有需要，则指定 core 文件的保存位置；
2. 客户环境运行中用来生成的 core 文件的运行的程序，和本地调试 core 文件的程序最好都是 debug 版本的，至少本地调试的程序得是 debug 版本，需要有调试用到的符号表。如果客户的程序是 release 版本的，则至少得保证两者的源码尽可能少不一样，否则调试会有潜在的错误存在。

## 8.2 与内存检查有关的调试实践

C/C++中由于会直接操作内存，所以对于了解内存中错误的类型是很有必要的。内存错误可能有内存泄露，内存溢出，内存访问错误这三种。以下是几种内存错误的详细分类。

关于内存检查的调试，在编译时可以使用 -fsanitize=address 参数可以在让内存有问题的程序在运行的时候提示错误的信息。如果只是单纯的编译，则存在内存错误的程序在运行时会直接崩溃，出现段错误的提示。单单提示出错的信息是不够的。

### 8.2.1 内存泄露

内存泄露广义上指的是申请的内存，没有及时的释放掉，所以在程序的运行中，有一些内存被占用后一直未释放的情况。

#### 1. 代码



#### 2. 调试过程

编译运行后的提示如下：



两次内存泄露，并且在下面打印出两处的函数调用过程。直接定位到内存泄露的调用函数。前者是 new 函数后的未 delete，后者是 malloc 之后的内存未 free

### 8.2.2 堆溢出

在 C/C++ 程序中，使用 new/malloc 函数分配的内存，存放的位置都是放在堆中，堆溢出指的是在使用申请函数得到的内存访问过程中发生了溢出。

#### 1. 代码



#### 2. 调试过程

编译时加上参数 -fsanitize=address 后，运行结果如下：



### 8.2.3 栈溢出

栈溢出，指的是在函数中直接分配的空间使用出现了溢出的问题

#### 1. 代码



#### 2. 调试过程



### 8.2.4 全局内存溢出

#### 1. 代码

#### 2. 调试过程



### 8.2.5 使用释放的内存

一个指针指向一块内存，然后释放这块内存，该指针没有置空，再使用这个指针的话，就出现使用释放的内存，导致程序崩溃。

#### 1. 代码

#### 2. 调试过程

## 8.3 release版本程序调试实践

Debug 版本，被称为调试版本，通过一系列编译选项的配合，生成的执行文件中包含很多调试信息，并且也没有优化，为开发人员的调试提供各种信息。

Release 版本，又称为发布版本，该版本是不包含编译信息，并且进行了各种优化，目的就是为了以期代码体积最小以及速度运行最快，为用户的使用提供便利。

但是如果 release 版本在运行时出现问题，那么我们该如何调试和定位问题呢。以下就是两种调试 release 版本的方法。

本质来说， release 版本无法调试原因是 release 版本中没有调试符号，但是出问题的又是 release 版本，所以解决这个问题的方法就是帮 release 版本的程序找到对应的调试符号，而无论是不带调试符号 release 版本和带调试符号的 debug 版本的来源都是源码，那么解决的方法就来了，从 Debug 版中提取调试符号或者将对应的调试版本作为符号源。

### 8.3.1 从调试版中提取调试符号

提取符号表的方法是：

1. objcopy --only-keep-debug [debug\_program][symbol\_name].symbol
2. gdb --symbol=[symbol\_name].symbol -exec=[release\_program]

#### 1. 代码

代码以 ProgramDebug\code\chapter\_6\function\function.cpp 代码为例

#### 2. 调试过程



#### 3. 补充说明

1）gdb 运行起来以后，使用 l main 查看 main 函数附近代码成功以后，就意味着加载符号表成功了，接下来就是正常的程序调试即可。

2）调试 release 版本，最好保证 debug 版本的代码是一样的，否则会出现符号表不匹配的警告，会有调试出错的风险。

### 8.3.2 直接使用调试版作为符号源

直接将调试版的代码作为符号源调试 release 版本程序的命令是：gdb --symbol=[debug\_program] -exec=[release\_program]

这种方式除了调试程序以外，还可以调试 release 版本的 core 文件，命令为：gdb [debug\_program] [core\_file].core

#### 1. 代码

代码以 ProgramDebug\code\chapter\_6\function\function.cpp 代码为例

#### 2. 调试过程

借用 debug 程序调试 release 程序的步骤同 8.3.1 类似，此处不在赘述，下面介绍一下 debug 程序调试 core 文件的步骤



#### 3. 补充说明

1）注意版本之前的源码不要有出入

2）有源码，有运行程序（release、debug）版本就可以调试

3）错误的程序记得设置生成的 core 文件配置。

# 参考文献

资料参考：

<https://wizardforcel.gitbooks.io/100-gdb-tips/content/show-copying-warranty.html>

<https://mp.weixin.qq.com/s/w_Z2ftOnb4VtB4jUaMF0fw>

<<the Art of Debugging with GDB,DDD, and Eclipse>>--软件调试的艺术

http://c.biancheng.net/stl/algorithms/ C++ STL容器

<http://c.biancheng.net/cplus/>

[如何排查大型C程序中的内存写越界导致的coredump？ - 知乎 (zhihu.com)](https://www.zhihu.com/question/51735480/answer/2663700699)

《C/C++调试的艺术》[C/C++代码调试的艺术-张海洋-微信读书 (qq.com)](https://weread.qq.com/web/reader/423320c07228f7b6423975ak09332a2023b093f65e0888c)

GDB调试-从入门实践到原理 https://mp.weixin.qq.com/s/w\_Z2ftOnb4VtB4jUaMF0fw

# 致谢

本文其实还是很感谢领导**gang.xie@aptiv.com**的支持和要求的。一是支持我们以论文的方式不断的学习和总结，并在这个过程中帮我们查找资料；二是要求我们无论如何也要有所输入和输出，没有领导的要求，估计也不会有如此的坚持和毅力完成这篇论文。

然后还有在此文撰写过程中帮我解决一些疑问的同事和鼓励我将这篇文章继续写下去的朋友。

最后还是要感谢自己的努力，本文历时一年，我一共翻阅了三本相关的书籍，和数十篇博客论文，还有GDB的官网资料，其中还有涉及到的Linux的一些知识，都有反复的查阅的修订，也是花费了不小的精力和时间来完成。

最终这个论文自己感觉还有一些未完善的部分，在以后的学习过程中，我会一点点的查缺补漏。

最后感谢阅读到此处的读者。