# GDB工具的基本调试命令用法

GDB工具是Unix/Linux 系统上标配的调试工具，主要有如下几个功能：

1、启动程序，可以按照程序员自定义的要求随心所欲的运行程序。

2、可让被调试的程序在所指定的[断点](https://baike.baidu.com/item/%E6%96%AD%E7%82%B9)处停住。（断点可以是[条件表达式](https://baike.baidu.com/item/%E6%9D%A1%E4%BB%B6%E8%A1%A8%E8%BE%BE%E5%BC%8F)）

3、当程序被停住时，可以通过dump文件来检查此时程序当前所发生的事。Dump文件是进程的内存镜像，可以把程序的执行状态（包括内存状态、CPU 寄存器状态等）通过调试器保存到dump文件中，然后使用调试器加以查看和分析。

4、可以修改程序，将一个BUG修正从而测试其他BUG。

## 一、GDB工具的安装

参考gdb安装教程.pdf

## 二、GDB工具的常用命令

在使用gdb工具来调试可执行程序之前，需要在gcc编译时，带上-g选项才能产生具有调试符号的程序，才能供gdb进行有源码的调试。典型如：gcc –g xxx.c xxx.S –o xxx。在产生带调试符号的可执行程序之后，即可使用gdb工具中的各项命令来对其分析和调试。

GDB工具的命令很多，这里仅给出常用的一些命令。注意：（）里的小写字母为命令的简写形式。更多的命令可以自行查阅相关资料。

1. list(l)命令：列出多行源代码。如果代码较长，可以连续多个list命令来查看。
2. break(b)命令：设置断点，break命令之后可接源文件行、函数名称、地址等，用法有：

* break [line-num] ：按代码行来设置断点，例如：break 3
* break [function-name]：按函数名来设置断点，例如：break main
* break \*地址：按地址来设置断点，地址可以通过disassemble来查看，地址形如“0X400”;例如：break \*0x400
* break if <condition>: E.g. break if i=10： 条件成立时程序停住
* info break：查看断点

1. start：开始执行程序，在第一条可执行语句处停下来。
2. run(r)命令： 运行程序，直到第一个断点或程序结束；

* 如果程序运行不需要输入参数，则直接输入：run
* 如果程序运行需要输入参数，则形如：run argv[1] argv[2]

1. 单步执行命令：

* step(s)命令：执行下一条语句，如果该语句为函数调用，则执行所调用函数执行其第一行语句；
* next(n)命令：执行下一条语句，如果该下一条语句为函数调用，不会进入函数内部执行(即不会一步步地调试函数内部语句)。
* continue(c)命令：继续程序的运行，直到下一个断点。如果没有下一个端点，则程序执行结束。

1. info(i)命令：

* info registers 查看CPU各寄存器的值；
* info args 查看当前函数参数的值
* info locals 查看当前函数中局部变量的值
* info frame 查看当前栈帧的详细信息

函数调用栈由连续的栈帧组成。每个栈帧记录一个函数调用的信息，这些信息包括函数参数，函数变量，函数运行地址。当程序启动后，栈中只有一个帧，这个帧就是main函数的帧。我们把这个帧叫做初始化帧或者叫做最外层帧。每当一个函数被调用，一个新帧将被建立，每当一个函数返回时，函数帧将被剔除。如果函数是个递归函数，栈中将有很多帧是记录同一个函数的。但前执行的函数的帧被称作最深帧，这个帧是现存栈中最近被创建的帧。在程序内部，函数栈帧用函数的地址来标记。gdb 为所有存活的栈帧分配一个数字编号，最深帧的编号是0，被它调用的帧的编号就是1。

7，x命令：查看内存的地址以及所存储的内容；

语法如下： x/<n/f/u> <addr>

n、f、u是可选的参数。

n是一个正整数，表示需要显示的内存单元的个数，也就是说从当前地址向后显示几个内存单元的内容，一个内存单元的大小由后面的u定义。

u 表示从当前地址往后请求的字节数，如果不指定的话，GDB默认是4个bytes。u参数可以用下面的字符来代替，b表示单字节，h表示双字节，w表示四字 节，g表示八字节。当我们指定了字节长度后，GDB会从指内存定的内存地址开始，读写指定字节，并把其当作一个值取出来。

注意： n表示单元个数，u表示每个单元的大小。

<addr>表示一个内存地址。

n/f/u三个参数可以一起使用。例如：

例如：x/3uh 0x54320 表示，从内存地址0x54320读取内容，h表示以双字节为一个单位，3表示输出三个单位，u表示按十六进制显示。

例如：x/20xw &a 显示变量a处开始的20个内存单元，16进制，4字节每单元

f 表示显示的格式，参见下面。如果地址所指的是字符串，那么格式可以是s，如果地址是指令地址，那么格式可以是i。输出格式

一般来说，GDB会根据变量的类型输出变量的值。但你也可以自定义GDB的输出的格式。例如，你想输出一个整数的十六进制，或是二进制来查看这个整型变量的中的位的情况。要做到这样，你可以使用GDB的数据显示格式：

x 按十六进制格式显示变量。

d 按十进制格式显示变量。

u 按十六进制格式显示无符号整型。

o 按八进制格式显示变量。

t 按二进制格式显示变量。

a 按十六进制格式显示变量。

c 按字符格式显示变量。

f 按浮点数格式显示变量。

8，disassemble(disass)命令：是反汇编命令，可以对具体的函数进行反汇编。

* disassemble function\_name：查看函数function\_name对应的汇编代码；
* 添加/r选项，即（gdb） disassemble /r function\_name：在前述基础上增加显示机器码；
* 添加/m选项，即（gdb） disassemble /m function\_name：在前述基础上增加显示源码（源码是C语句则显示C代码，源码是汇编语句则显示汇编代码）；
* 也可以同时添加/rm选项；

9，display命令：跟踪查看某个变量，每次单步停下来都显示该变量当前的值。 undisplay命令则是取消跟踪观察变量。

10，watch命令： 监视变量值的变化。当被设置的观察点变量发生变化时，打印显示！

与display命令一直显示某个变量的值的方式相比，用watch命令只显示变量值的变化情况，如果变量值不变则不显示，调试效率要高很多。

11，file命令：在当前程序运行完毕退出之后，可以使用file命令重新载入待调试程序，重新进行调试。典型格式是：file executablefilenam。

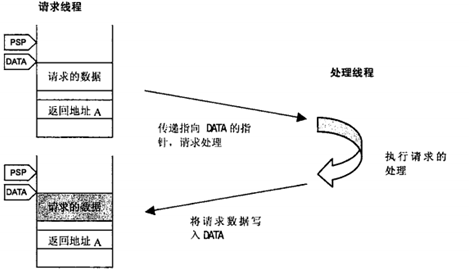
12，kill命令：终止当前的调试过程；

13，backtrace(bt)命令：查看当前函数及其被调用的信息

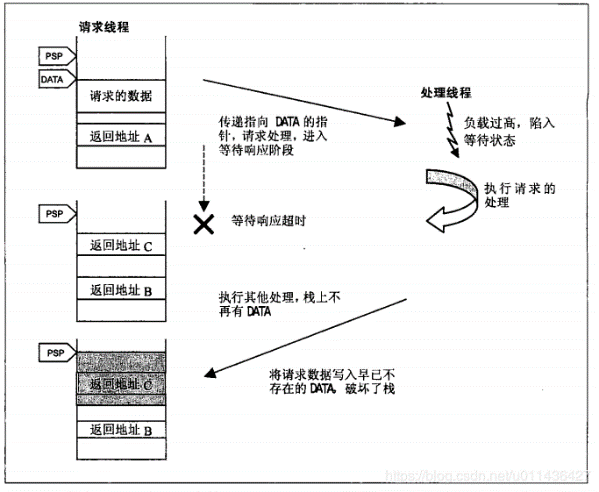
[backtrace](http://hi.chinaunix.net/?uid-1825075-action-viewspace-itemid-40672)命令是依据进程栈里保存的函数返回地址来显示函数间调用关系的，可以用于回溯函数调用栈。如果bt命令下所显示的某些函数不正确或者出现段错误，则可以认为是栈被破坏了，还可以看到是在哪个函数的调用处产生的段错误。

线程间的冲突是导致进程栈被破坏的原因之一。举一个例子来展示栈被破坏的过程：如下图所示，在线程间的数据处理上传递了栈的指针，导致了其他线程向该地址写入了数据。而这个其他线程向栈内写入数据的操作被推迟了，推迟后的写操作写入了一个其它线程的栈空间，从而导致了栈破坏

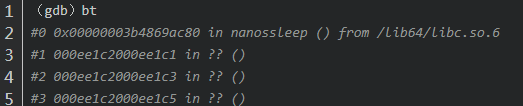
正常时，应用程序行为如下：



问题时，应用程序的行为如下：



当因为上述原因出现段错误的时候，执行backtrace，如下图所示，可以看到是哪里的调用，产生的这个段错误。？？( )就表明调用当前函数的上一级函数是有误的，其原因可能是进程栈被破坏所导致的。



14，help(h)命令：查看命令帮助，具体命令查询为输入help+命令；

15，quit(q)命令或ctrl+c命令：退出gdb环境

## 三、示例代码

**1，示例代码1：**

C语言-ArmV8汇编混合编程，实现两个数的相加：

------------example.c---C代码主函数main--------

#include <stdio.h>

#include <stdlib.h>

typedef unsigned int u32;

extern int plus(u32, u32);

int main()

{

u32 a = 2;

u32 b = 3;

printf("%d\n", plus(a, b));

return 0;

}

-------------plus.S-----汇编代码函数plus--------------------

#include "plus.h"

ENTRY(plus)

add w0, w1, w0

ret

ENDPROC(plus)

-----------------plus.h----------汇编代码的头文件------------------

#ifndef ENTRY

#define ENTRY(name) \

.globl name ; \

.align 4 ; \

name:

#endif

/\* If symbol 'name' is treated as a subroutine (gets called, and returns)

\* \* then please use ENDPROC to mark 'name' as STT\_FUNC for the benefit of

\* \* static analysis tools such as stack depth analyzer.

\*

\* \*/

#ifndef ENDPROC

#define ENDPROC(name) \

.type name, @function ; \

.size name, .-name

#endif



**注意：用gcc 编译汇编的文件，汇编文件扩展名必须是S，不能是s，s只做编译未做预处理。**

2，对例1代码进行带调试信息的编译：

gcc –g example.c plus.S –o example

生成带调试信息的可执行文件example



**2，示例代码2：**

纯汇编程序hello.s（或者hello.S，在这里不区分s的大小写）

.text @ 这个定义可选；

.global tart1

tart1:

mov x0,#0

ldr x1,=msg

mov x2,len

mov x8,64

svc #0

mov x0,123

mov x8,93

svc #0

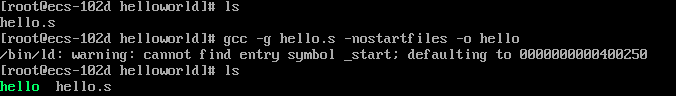
.data

msg:

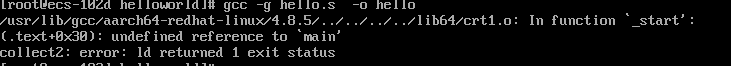
.ascii "Hello World!\n"

len=.-msg

对该hello.s采用gcc进行编译，使用了-g 和 –nostartfiles选项，编译生成了带调试符号表的可执行文件hello。



如果不采用-nostartfiles选项，则会出现如下错误：



即找不到主函数main或者入口函数的错误，这是采用gcc编译器时会出现的问题。如果gcc编译时采用了 –nostartfiles选项，则可以避免该错误。

**3，示例代码3采用gcc来编译**

如果将示例代码2中的“tart1”修改为“\_start”，即如下的示例代码3：

# h2.s

.global \_start

\_start:

mov x0,#0

ldr x1,=msg

mov x2,len

mov x8,64

svc #0

mov x0,123

mov x8,93

svc #0

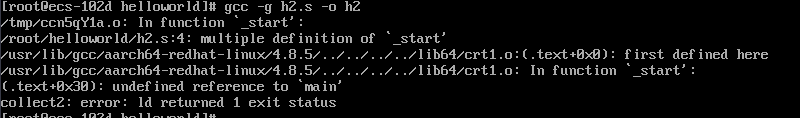
.data

msg:

.ascii "Hello World!\n"

len=.-msg

在采用gcc对其进行编译时就会出现对“\_start”多个定义导致定义冲突的问题：



此时可以添加“-nostartfiles选项”来避免该问题。需要说明的是，由于采用了-nostartfiles选项，那么在带有调试信息的可执行文件hello中就不再含有程序从哪里开始start的信息了，此时gdb中就不能使用start命令来执行程序的第一行代码了。但是可以采用 break 1和run组合的方式来代替start命令寻找程序的第一行并开始执行。

**4，示例代码3采用as和ld命令来编译**

as和ld是linux下的汇编命令和连接命令，可以用其来编译和连接汇编源代码，生成可执行程序。由于gcc在执行时会默认使用自身库中的crt1.o中的\_start作为程序的入口点，就会与示例代码中定义的“\_start”入口点冲突，导致示例3中的问题，对具体过程感兴趣的同学可以参考“<http://www.doc88.com/p-9847334917507.html>”。使用as和ld这两个命令可以避免gcc编译时产生的“\_start”多定义冲突问题。

为了生成带调试符号表的可执行程序来在gdb环境下进行调试，需要在使用as命令时使用-gstabs选项。这两个命令使用示例如下：



这样就可以生成带调试符号表的可执行文件h2。

综上所述，如果要对含有”\_start”标号的汇编代码进行带调试符号表的编译，为避免\_start多定义冲突问题，有两种方式：一是使用gcc –g –nostartfiles xx.s –o xx命令来编译；二是使用as –gstabs xx.s –o xx.o命令和ld xx.o –o xx命令来编译和链接。在gdb环节下对这两种方式生成的可执行文件进行调试时，采用break 1和run命令来取代start命令来开始调试可执行文件的第一行代码。

4、示例代码4 移位运算

在命令行输入 vim lab1.S

复制粘贴以下指令

.data@数据段

xx:

.byte 6

.text@代码段

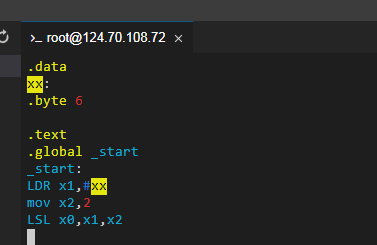
.global \_start @程序入口

\_start:

mov x1,#xx@将变量xx赋值给x1寄存器

mov x2,2@将立即数赋值给x2寄存器

LSL x0,x1,x2@将x1寄存器的数值左移x2数值，然后赋值给x0.



Vim指令 参见在 Linux 中使用 vim常用命令.docx

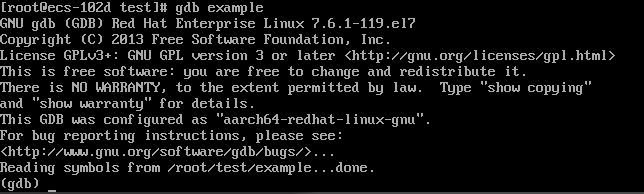
使用gcc编译lab1.S

gcc –g –lab1.S –nostartfiles –o lab1

## 四、采用GDB工具来调试含调试信息的可执行文件example

1，采用gdb来调试example文件

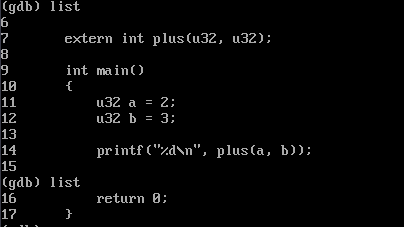
在命令行输入： gdb example，回车后出现：



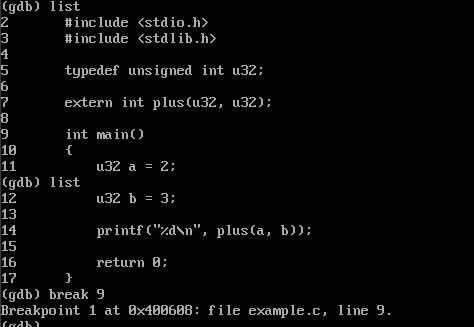
如果gcc编译时不使用-g选项，则出现“no debugging symbols found”字样，无法对其进行gdb调试。



2，用多个list命令查看多行源代码

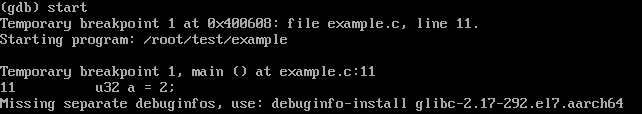


3，在代码的第9行插入第一个断点

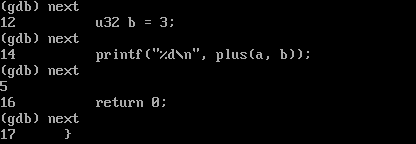


在此基础上，再使用run命令，则表示程序先执行到第9行处就暂停。

4，采用start命令来运行第一行代码

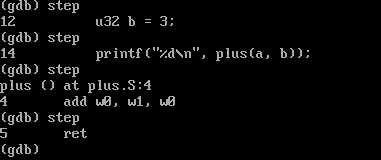


5，采用next命令来单步执行



注意观察：next命令不展开plus函数内部的代码来进行单步，直接跳过。

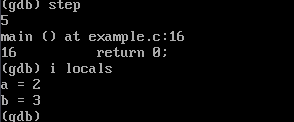
6，采用step命令来单步执行



注意观察：step命令展开plus函数内部的代码来进行单步。

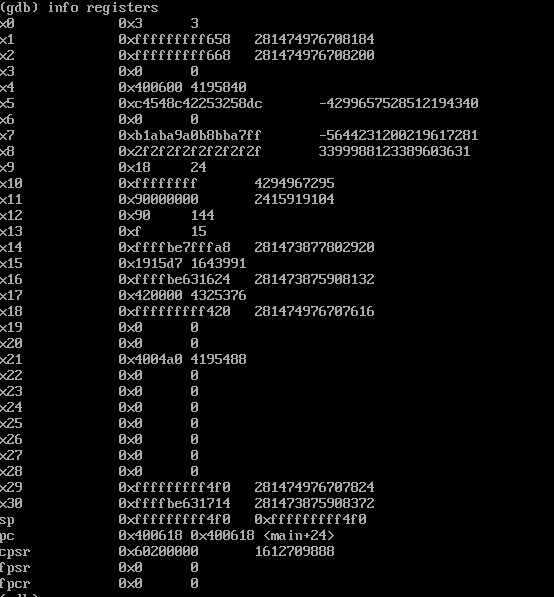
1. 在程序单步执行过程中，可以：

* 使用info locals命令来查看当前函数中局部变量的值

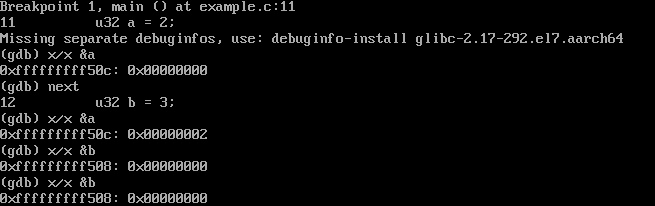


当前函数main中的局部变量为a和b。

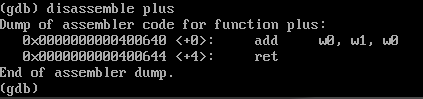
* 使用info registers命令来查看处理器中各寄存器的值。



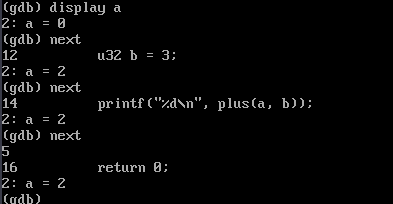
* 使用x命令来查看example程序中变量a和b所在的内存地址（用&a和&b表示）以及取值情况，在下图中采用了x/x &a以及x/x &b命令。



* 可以使用disassemble命令来查看函数如plus的汇编代码

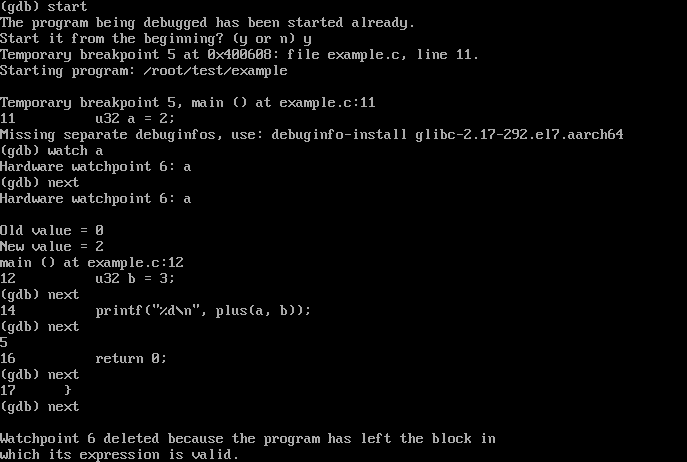


* 可以使用display命令来在每一步查看待观察的变量值，比如display a



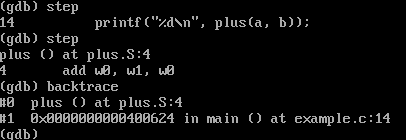
在每一次next之后，都会显示变量a的值。

* 使用watch命令来在变量a的值的变化情况



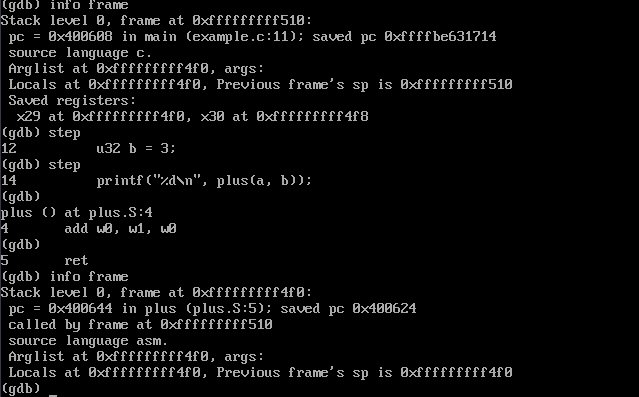
可以看出，在使用watch a命令之后，只有当变量a的值发生变化时才在屏幕上打印出变化前的a值和变化后的a值，其它时刻不在屏幕上打印a的值。在监控效率上，使用watch命令要比display命令的效率高的多。

* 使用backtrace命令来在观察当前函数被调用的情况



当前函数（用“#0”标志）plus被上一级函数（用“#1”）main所调用。

* 使用info frame命令来查看当前栈帧的信息



## 五、作业题：

参照第四节，用gdb的各项命令来查看示例2代码编译生成的hello可执行文件。