## 简介

GDB(GNU Debugger)是一个在Unix及类Unix系统命令行环境下的调试器。具有非常强大的功能。常用有断点，查看和修改内存数据，查看调用栈，修改程序运行逻辑，信号处理等功能。熟练运用它可以有效提高定位程序bug的效率。

GDB只能用于调试ELF可执行文件或动态库，并且只能调试运行在用户态的程序。shell脚本，ko或进入内核部分的代码是无法调试的。

## 使用准备

为了使用gdb，我们需要编译出可调试版本的程序或动态库。在gcc编译c或c++源文件时需要加一个编译参数-g。gdb支持4种不同的调试级别，分别是-g0-3。等级越高，可执行文件和调试占用内存会越大。一般我们直接按默认-g就行，如果需要调试宏定义需要使用-g3。

如果完全编译调试版本，版本程序会比非调试版本程序大非常多，有时候能达到十几倍。后续会提供几种方式来裁剪文件大小。

文本

描述已自动生成

## 常用命令

1. r/run [arg] 重新运行，arg参数。
2. bt/backtrace 显示调用栈信息。
3. f/frame n 切换到第n帧。其中n会在bt命令中显示，从0开始。0表示栈顶。
4. up/down n 往栈顶/栈底移动n帧。当不输入n时，默认为1。
5. p/print[/x] exp打印信息，exp是表达式，可以是数组，对象，算式等。/x 16进制显示
6. display[/x] exp 和print一样，不过每次程序停止都会自动打印
7. undisplay 删除display
8. call fun 调用函数，会破坏原有程序执行逻辑
9. l/list 查看源码，嵌入式host模式下基本无法使用。
10. i/info thread 查看线程信息，调试前最好在代码中添加线程名
11. t thread\_id 切换到某个线程，id是pthread库定义的id
12. b/break fun 在fun函数上设置断点
13. b/break xx.c:line 在xx.c文件的line行设置断点
14. save breakpoints gdb.cfg 保存断点信息到文件
15. source gdb.cfg 加载断点信息
16. watch/rwatch/awatch exp 监控表达式，当表达式被写/读/读写时中断
17. c/continue 继续运行
18. until xx.c:line 程序运行到哪一行后停止，若不带参数，为执行完循环体后结束
19. finish 跳出函数，执行完本程序
20. return 跳出函数，同c语言return，会破坏原有程序执行逻辑
21. s/step 单步调试，进入函数。
22. n/next 单步调试，不进入函数。
23. set var x=10 改变当前变量x的值。也可以这样用：set {int}0x83040 = 10把内存地址0x83040的值强制转换为int并赋值为10
24. i/info locals 打印当前栈帧的本地变量
25. i/info reg 查看cpu内部寄存器
26. dump memory file addr addr+size dump内存到文件
27. jump使当前执行的程序跳转到某一行，会破坏原有程序逻辑，而且可能导致奔溃
28. dprintf xx.c:line,”%s”.str 在某个位置添加格式化打印
29. set solib-search-path 设置动态库搜索路径，一般gdbserver会用到

## gdb调试常见问题

##### 段错误

段错误是由非法内存访问引起的。Linux中，发生段错误后操作系统会向引发段错误的进程发送SIGSEGV信号。gdb捕获到信号后会暂停，之后就可以用bt，p等命令查看出错部分的代码了。

文本

中度可信度描述已自动生成

文本

描述已自动生成

##### Watch定位踩内存问题

watch/rwatch/awatch可以监测表达式值，一旦被读或写就会中断程序运行。用这个特性可以很容易的定位踩内存问题

文本

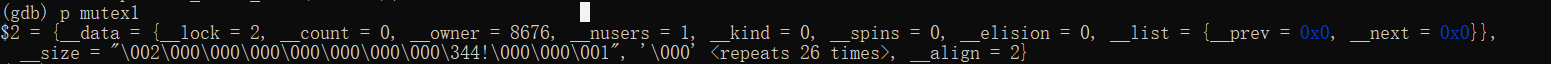
描述已自动生成

文本

描述已自动生成

##### 死锁问题

##### 这是一个死锁的例子。我们可以通过glibc的pthread\_mutex\_t结构体定位死锁发生的位置。\_\_owner为持有锁的轻量级进程tid，通过gdb可以很容易的定位处哪个线程没有释放锁。



## 嵌入式环境下使用gdb

由于嵌入式环境一般资源都比较有限，所以有必要对程序进行一些裁剪。

1. 最简单的方法就是控制编译过程，只在编译要调试的c文件时加-g。由于我们编译框架使用shell加cmake方式，可以设置一下cmake文件属性控制哪些文件加-g。

set\_source\_files\_properties(file1.c

file2.c

…

PROPERTIES COMPILE\_FLAGS -g

)

1. 编译完整的调试版本，然后使用strip或objcopy编辑可执行文件，来删除或者保留哪些信息，动态加载符号表调试信息。
2. 使用gdbserver

#### 动态加载符号表和调试信息

调试版本的程序，当我们使用bt命令是可以看到很多信息。

文本

描述已自动生成

调试版本程序也可以用print命令打印变量的值，查看任意合法内存值等操作。

屏幕上有字

描述已自动生成

对于非调试版本的程序，当我们用gdb调试时获取的信息非常有限，比如未加-g使用文本

描述已自动生成

可以看到，虽然bt可以看到调用栈但是无法看到行号，也无法查看变量的值。如果用strip去除符号表，这时候连调用栈都无法看到

文本

描述已自动生成

为了解决编译文件过大，偶现问题，调试版本不适合作为发布等问题。这里提供一种动态加载符号表的方式。在gdb交互界面使用命令symbol-file test.debug加载调试信息即可。如果是调试动态库，需要先i sharedlibrary查看动态库的加载地址，之后使用add-symbol-file test.debug addr加载动态库的调试信息。

文本

描述已自动生成

#### gdbcore

对于内存不足flash充足的设备，可以采用core文件的方式调试。Core文件是当进程运行出错退出时会产生的文件。Core文件由于存储了进程的内存镜像，往往比较大。所以只有在flash资源充足时才能使用。

Core文件使用时需要修改一下系统配置。

1. 首先使用ulimit -c unlimit命令放开core文件大小限制，否则限制为0或太小是无法生成core文件的
2. 然后可以使用echo /devinfo/core.%e.%p> /proc/sys/kernel/core\_pattern格式的命令修改core生成路径和文件名规则，详细可以man core查看。默认是在进程运行目录下生成
3. 之后如果进程错误退出就会生成core文件

文本

描述已自动生成

4． 使用gdb bubble core调试，core文件只保存内存进行和进程信息。符号信息和调试信息是依赖于前面的可执行程序的。

值得注意的是，并不是所有错误都会生成core文件。linux只有SIGABRT,SIGBUS,SIGFPE,SIGILL,SIGIOT,SIGQUIT,SIGSEGV,SIGSYS,SIGTRAP,SIGUNUSED,SIGXCPU,SIGXFSZ这几个信号的默认处理方式会生成core文件。详细man 7 signal查看。另外，core方式调试是不能运行命令的。只能查看退出时的进程状态。

#### gdbserver远程调试

有时候设备资源实在是无法满足运行gdb的条件。如果有网络能与主机联通，可以采用gdbserver。gdbserver也支持动态加载符号表，attach等操作，但是不支持run命令。

gdb host方式，gdb进程和ezapp进程都运行在arm设备端，内存，flash，cpu都使用设备的资源。符号表，调试信息等都是加载到设备内存中。

图示

描述已自动生成

gdb server运行模式，设备端只运行ezapp和一个gdbserver进程。gdbserver只负责和主机通讯，基本不占用资源。调试信息等内存都是直接加载到主机端。

图示

描述已自动生成

gdbserver的使用步骤如下：

1. 在设备端运行gdbserver 127.0.0.1:port test
2. 在主机端运行gdb
3. 在gdb交互界面使用target remote xxx.xxx.xxx.xxx:port连接设备

gdbserver也支持通过串口方式远程调试，不过需要主机端能通过。使用步骤如下:

1. 在设备端运行gdbserver /dev/ttyS0 test #设备文件为本地串口文件
2. 在主机端运行gdb
3. 在gdb交互界面使用target remote /dev/ttyS0连接设备 #设备文件为连接远程设备的串口