使用KGDB调试技巧

目录

简介

课程目标

使大家能用GDB调试Linux下的应用程序及内核驱动程序，熟悉Linux编程环境下的调试技巧，熟悉Linux下的命令。

课程介绍

本课程是讲解GDB调试工作怎样去调试应用程序，以及利用KGDB去调试Linux内核去驱动的方法。

GDB是什么

GDB是GNU开源组织发布的一个强大的UNIX下的程序调试工具。GDB作为一个专用调试器，允许你去观察一个程序的运行过程，或者当一个程序发生崩溃时，程序内部发生了什么。

GDB主要帮忙你完成下面四个方面的功能：

1、启动你的程序，可以按照你的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你所指定的调置的断点处停住。（断点可以是条件表达式）

3、当程序被停住时，可以检查此时你的程序中所发生的事。

4、动态的改变你程序的执行环境。

目录

第一章 应用程序的调试

第1节. GDB常用命令介绍

1.1 通用命令

➢启动GDB

你可以直接敲入“gdb”开始gdb程序，也可以带各种参数启动，形式如下：

gdb program

gdb program core

gdb program 1234

➢退出GDB

quit [expression]

q

退出GDB用"quit"命令，或者是缩写"q"，或者按下ctrl-d。如果后面跟着表达式时，将退出gdb返回表达式计算的值作为返回值

第一章 应用程序的调试

- Shell 命令

shell command string

启动shell去执行 command string

make 是不需要用shell而直接可以运行。

- log文件设置命令

我们可以吧GDB命令的输出到文件中，下面是log文件的使用方法。

set logging on/off

set logging file file：改变当前的log文件名，默认是gdb.txt。

set logging overwrite[on/off] ：默认是追加的。

set logging redirect[on/off]：默认是输出到终端和日志文件，用这个命令可以只输出到logfile。

show logging

- 帮助命令

help [command/class]

敲入"help"或缩写"h"就可以启动gdb帮助信息。

apropos args

complete args

第一章 应用程序的调试

1.2 GDB运行控制类命令

- run
  r

  在gdb下运行程序，执行这个命令之前，我们必须通过GDB 带参数的形式制定了程序名，参数就是所制定的程序名。

- start

  在gdb下运行程序，执行这个命令等于设了个临时断点在main函数，然后再run。

- attach process-id

  跟踪一个已经运行的进程，process-id是进程ID。

第一章 应用程序的调试

- break
  b

  断点用于停止运行程序，是调试手段中一个非常重要的手段，命令形式主要有以下几种。

  break
  break function
  break filename:function
  break filename:linenum
  break address
  break...if cond

  设置条件断点，只有到cond条件满足，也就是con表达式的值为真的时候程序才会断下来。

  tbreak args

  设置一个临时断点，只有效一次。

第一章 应用程序的调试

➢watch

我们可以设置一个观察点，当所观察的值改变的时候，程序就会停下来，主要有以下几种形式。

watch expr
当表达式变为值的时候停止

rwatch expr
当值被读的时候停止

awatch expr
当值被读或者被写的时候停止

第一章 应用程序的调试

- continue[ignore-count]
  c[ignore-count]

  恢复程序执行，程序从当前断点继续往下执行，直到遇到下一个断点。
  [ignore-count]参数，告诉GDB往下执行忽略的程序运行中断次数。

- step [count]

  step 执行一行代码行然后停下来，如果这段代码行是函数的话，那么将进入函数内部单步执行。step count 连续执行count次step。

- next [count]

  next 执行一行代码行然后停下来，如果这段代码行是函数的话，那么将不进入函数内部而直接执行完整个函数 再停下来。

- finish

  继续执行当前函数，直到运行到当前函数返回后才停止，并且显示返回值。

第一章 应用程序的调试

1.3 GDB信号

信号是一种软中断，是一种处理异步事件的方法。一般来说，操作系统都支持许多信号。尤其是UNIX，比较重要应用程序一般都会处理信号。UNIX定义了许多信号，比如SIGINT表示中断字符信号，也就是Ctrl+C的信号。SIGSEGV 是当一个程序访问超出了访问区域时就会出现SIGSEGV。

- info handle/signals

  打印所有信号，并且告诉我们GDB怎么去处理每个信号。

- handle signal keywords...

  改变gdb对信号的处理方式，有以下几种处理方式

stop	nostop
print	noprint
pass	No pass

第一章 应用程序的调试

1.4 GDB检查

- print expr
- print variable=XX
- ’bt’（’backtrace’［回溯］的缩写）

  回溯命令让你知道程序现在在什么地方，每一行显示一帧。

- list
  list-
  list linenum
  list function

第一章 应用程序的调试

第2节.GDB使用前的准备

1. 安装GDB

2. 编译应用程序
   gcc（或 g++）下使用额外的 ‘-g’ 选项来编译程序。

   makefile 中如下定义 CFLAGS 变量：
   ```
   CFLAGS = -g
   ```

3. 运行 gdb
   - 带参数的运行方式。
   - file命令转入方式。

4. gdb帮助
   ```
   help
   ```

第一章 应用程序的调试

第3节.GDB实战

本章示例：

```c
#include <stdio.h>
int wib(int no1, int no2)
{
    int result, diff;
    diff = no1 - no2;
    result = no1 / diff;
    return result;
}
int main(int argc, char *argv[])
{
    int value, div, result, i, total;
    value = 10;
    div = 6;
    total = 0;
    for(i = 0; i < 10; i++)
    {
        result = wib(value, div);
        total += result;
        div++;
        value--;
    }
    printf("%d wibed by %d equals %d\n", value, div, total);
    return 0;
}
```

运行结果：

```
Floating point exception
```

第一章 应用程序的调试

1. 运行 gdb
   - 带参数的运行方式。
   - file命令转入方式。

2. 运行
   run （执行装载进的程序）。

3. 查看代码

4. 要查看变量的值：

5. ‘continue’ 命令和quit命令
- ’continue’ 命令告诉 gdb 继续执行。
- ‘quit’ 命令告诉 gdb 退出到shell模式下。

6. 动态修改变量的值：

```
print Variable=XX
```

第一章 应用程序的调试

- 在特定行上设定断点。
- 查询当前断点。
- 使能和禁用断点。
- 删除断点。

第一章 应用程序的调试

- 显示所有局部变量的 info locals变量

```
(gdb) info locals
value = 10
div = 6
result = 2637812
i = 0
total = 0
```

第一次循环

```
(gdb) info locals
value = 9
div = 7
result = 2
i = 1
total = 2
(gdb)
```

第二次循环

```
(gdb) info locals
value = 8
div = 8
result = 4
i = 2
total = 6
(gdb)
```

第三次循环

- 单步执行next命令
- 单步执行step命令
- finish

第一章 应用程序的调试

- 条件断点使用

"break <line number> if <conditional expression>"。

- 监视点的应用

```
(gdb) watch div==value
Hardware watchpoint 2: div==value
```

- 观察/使能/禁用 监视点

awatch ：当表达式（变量）的值被读或被写时，停住程序。

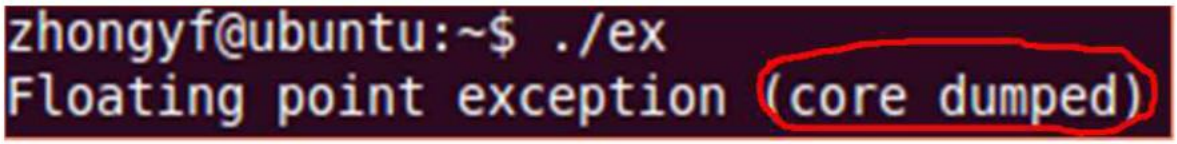
第一章 应用程序的调试

core 文件的应用

在 gdb 下运行程序可以使俘获错误变得更容易，但在调试器外运行的程序通常会中止而只留下一个 core 文件。gdb 可以装入 core 文件，并让您检查程序中止之前的状态。

1. linux下开启coredump
2. 重新运行示例程序

在 gdb 外运行示例程序 eg1 将会导致核心信息转储：



第一章 应用程序的调试

3.装入Core文件运行GDB

‘gdb eg1 core’ 或 ‘gdb eg1 -c core’看到的消息：

```
Core was generated by `./ex'.
Program terminated with signal 8, Arithmetic exception.
#0  0x08048401 in wib (no1=8, no2=8) at ex.c:6
6           result = no1 / diff;
(gdb)
```

- 堆栈命令 ’bt’（’backtrace’[回溯] 的缩写）：

  为了更容易地查明在调用 wib() 的函数中发生了什么情况，可以使用 gdb 的堆栈命令。

练习

下列程序显示一个简单的问候”hello here is TFSP”，再反序将它列出但是程序并不能按我们想法执行，请通过GDB查出问题所在。

```
#include<stdio.h>
#include<string.h>
void my_print (char *string)
{
    printf ("The string is %s\n", string);
}
void my_print2 (char *string)
{
    char *string2;
    int size, i;
    size = strlen (string);
    string2 = (char *) malloc (size + 1);
    for (i = 0; i < size; i++)
    string2[size - i] = string[i];
    string2[size+1] = '\0';
    printf ("The string printed backward is %s\n", string2);
}
main ()
{
    char my_string[] = "hello here is TFSP";
    my_print (my_string);
    my_print2 (my_string);
}
```

使用GDB Server

1.在目标板上运行gdbserver

gdbserver 192.168.0.3:2345 hello

其中192.168.0.3为目标板的IP，可以写localhost，也可以不写。2345为gdbserver打开的端口，可以自己设置。

2. 在宿主机上运行

```
#arm-linux-gdb hello
(gdb)target remote 192.168.0.2:2345
```

第一章 应用程序的调试

第二章 内核级调试

第二章 内核级调试

通用的桌面操作系统与嵌入式操作系统在调试环境上存在明显的差别。远程调试，调试器运行于通用桌面操作系统的应用程序，被调试的程序则运行于基于特定硬件平台的嵌入式操作系统（目标操作系统）

第1节.几种调试技术

- printk

  是调试内核代码时最常用的一种技术。在内核代码中的特定位置加入printk() 调试调用.

- kdb：

  Linux 自带的内核调试器，可以在一台机器中进行调试.

- Kprobes：

  提供了一个强行进入任何内核例程，并从中断处理器无干扰地收集信息的接口。

第二章 内核级调试

- KGDB

以上介绍了进行Linux内核调试和跟踪时的常用技术和方法。当然，内核调试与跟踪的方法还不止以上提到的这些。这些调试技术的一个共同的特点在于，他们都不能提供源代码级的有效的内核调试手段，有些只能称之为错误跟踪技术，因此这些方法都只能提供有限的调试能力。

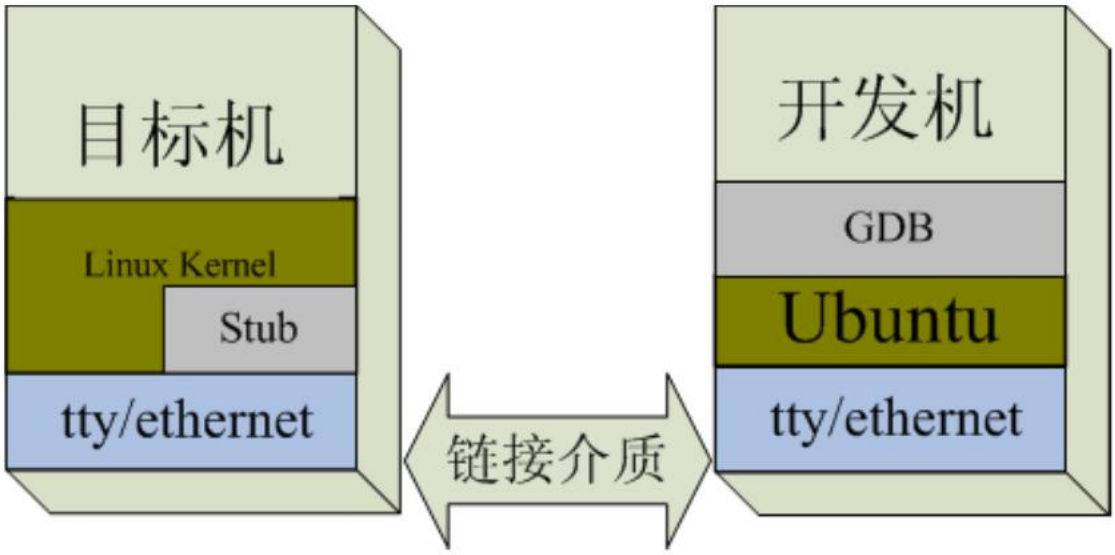
kgdb提供了一种使用 gdb调试 Linux 内核的机制。使用KGDB可以象调试普通的应用程序那样，在内核中进行设置断点、检查变量值、单步跟踪程序运行等操作。

第二章 内核级调试

第2节 Kgdb在Vmware的配置

- 实验环境：
  1. Fedora 11。
  2. gcc version 4.4.3 for i486-linux-gnu。
  3. Linux Source : 2.6.32。
  4. Vmware Workstation 7.1.4
  5. Inter Celeron CPU 2.4GHz。
  6. memeory 3.25GB
- kgdb的调试原理
  请看下图。

第二章 内核级调试


目标机
Linux Kernel
Stub
tty/ethernet
链接介质
开发机
GDB
Ubuntu
tty/ethernet

第二章 内核级调试

- Kgdb的安装与设置

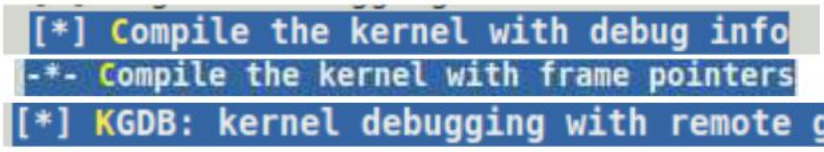
1. 编译内核前要先把clean一下
   sudo make mrproper

2. 设置内核选项

   make menuconfig，启动内核设置界面，进入

去掉: 

进入

选择: 

去掉: 

第二章 内核级调试

3. 编译内核

```
$make all
$make modules_install
$make install
```

4. 安装完成后,关闭Linux,然后Clone一台同样的计算机.步骤如下:

·点击VM->Clone

·选中默认的From current state,点击Next

·选中Create a full clone, 点击Next

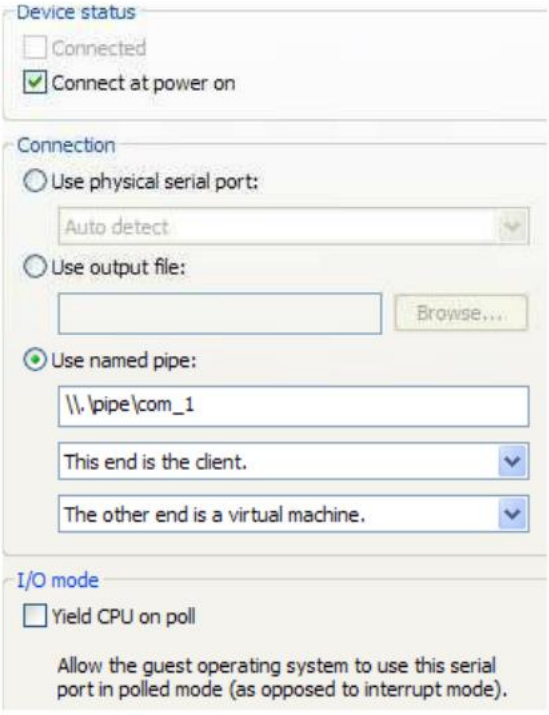
· Virtual Machine name 输入Target,将克隆的机器命令为目标机.

第二章 内核级调试

5.克隆完成后需要给两个系统增加串口，以"Output to named pipe"方式,其中:

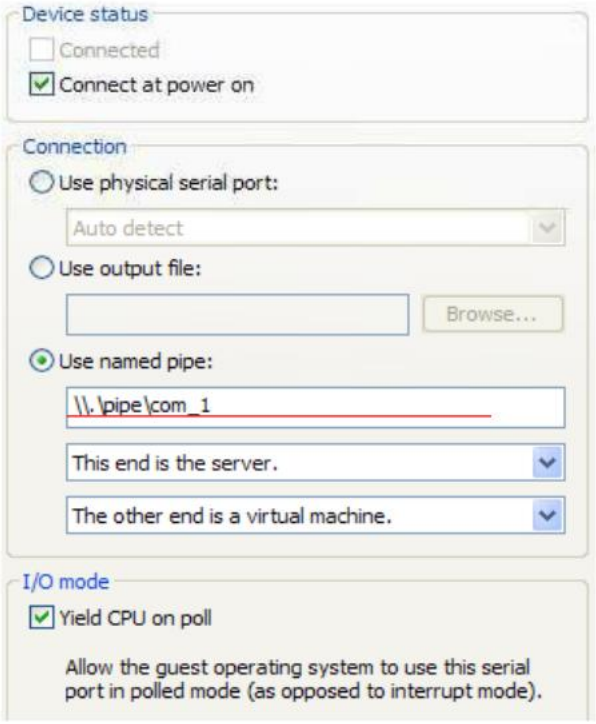
Host端选择“this end is the client”，“the other end is a virtual machine”，设置完成后如图所示

第二章 内核级调试


Device status
Connected
Connect at power on
Connection
Use physical serial port:
Auto detect
Use output file:
Browse...
Use named pipe:
\\.\pipe\com_1
This end is the client.
The other end is a virtual machine.
I/O mode
Yield CPU on poll
Allow the guest operating system to use this serial port in polled mode (as opposed to interrupt mode).

第二章 内核级调试

Target端选择"this end is the server", "the other end is a virtual machine"



两个pipe的名称要相同,并且选中下面的Connect at power on,及Advanced里面的Yield CPU on poll 。

第二章 内核级调试

6. 测试两系统是否能建立联系:

Target端 输入: cat /dev/ttyS0

Development端 输入: echo "hello" >/dev/ttyS0

串口添加完成后会在 Target端输出 “hello”，这说明说明串口通讯正常。

7. 在Target机上设置 /boot/grub/menu.lst文件 :

timeout=10

增加到Linux 2.6.32下的启动选项下：

kernel /vmlinuz-2.6.32 ro root=/dev/mapper/vg_fjtlinux-lv_root rhgb kgdboc=ttyS0,115200 kgdbwait quiet

第二章 内核级调试

重新启动target机器
会显示： kgdb: Waiting for connection from remote gdb

8.启动GDB调试:
转到Host机器上进入Linux源文件所在目录，可以看到目录下生成了vmlinux文件，这个就是没有经过压缩的linux内核文件。

执行
Gdb ./vmlinux

9.设置调试参数 :
(gdb) set remotebaud 115200
(gdb) target remote /dev/ttyS0。

敲回车后gdb就会得到target的控制停下来，等待用户输入命令，剩下的用户就可以像调试应用程序一样调试内核了。

要中断正在运行的target系统的话，只要在target的终端上敲入echo g > /proc/sysrq-trigger，即可终端现行系统，Host端gdb又进入命令模式。

第二章 内核级调试

第3节.驱动的调试方法

1. 在这里，我们先写一个最简单的驱动程序，hello驱动。程序清单如下：

```
#define _KERNEL_
#define  MODULE
#define _LINUX_MODULE_H
#define MODULE_LICENSE(_license) MODULE_INFO(license, _license)
#include<linux/module.h>
#include<linux/kernel.h>
#include<linux/init.h>
#include<linux/sched.h>
#ifdef LICENSE_GPL
MODULE_LICENSE("GPL");/* declare the license of the module ,it is necessary */
#endif
static int hello_init(void)
{
    printk( "<0>Hello World enter!\n");
    return 0;

}
static void hello_exit(void)
{
    printk( "<0>Hello world exit!\n");
}
module_init(hello_init); /* load the module */
module_exit(hello_exit); /* unload the module */
```

第二章 内核级调试

2. 制作makefile：

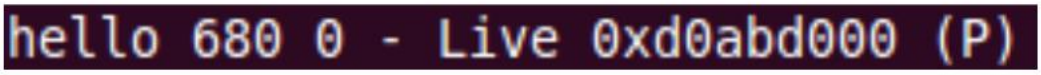
```
DEBUG = y

ifeq ($(DEBUG),y)
    DEBFLAGS = -O -g
else
    DEBFLAGS = -O2
endif
EXTRA_CFLAGS += $(DEBFLAGS)-I$(LDDINC)
ifneq ($(KERNELRELEASE),)
obj-m := hello.o
else
KERNELDIR ?= /lib/modules/$(shell uname -r)/build
PWD       := $(shell pwd)
modules:
        $(MAKE) -C $(KERNELDIR) M=$(PWD) LDDINC=$(PWD)/../include modules
endif
clean:
        rm -rf *.o *~ core .depend .*.cmd *.ko *.mod.c .tmp_versions
depend .depend dep:
        $(CC) $(CFLAGS) -M *.c > .depend
ifeq (.depend,$(wildcard .depend))
        include .depend
endif
```

3. 编译驱动。

4. 安装/卸载驱动模块：

执行
insmod hello.ko文件安装驱动，如果没有错误信息就代表安装成功。
执行
rmmod hello.ko文件卸载驱动。

5. 取得模块动态加载信息
执行 cat /proc/modules 可以得到所加载模块的动态信息。



0xd0abd000 就是hello的加载地址。

第二章 内核级调试

6. 加载符号文件。执行

```
(gdb) add-symbol-file /home/zhongyifeng/Downloads/hello/hello.ko 0xd0abd000
add symbol table from file "/home/zhongyifeng/Downloads/hello/hello.ko" at
        .text_addr = 0xd0abd000
(y or n) y
Reading symbols from /home/zhongyifeng/Downloads/hello/hello.ko...done.
```

7. 加载成功后，可以设断点，等动作。

```
(gdb) break /home/zhongyifeng/Downloads/hello/hello.c:34

Breakpoint 1 at 0xd0abd00b: file /home/zhongyifeng/Downloads/hello/hello.c, line
 34.
```

设置成功后继续运行程序。continue

第二章 内核级调试

在target端我们执行rmmod hello。

可以看到刚才设的断点起作用停止了hello卸载的过程，控制权又转交给了gdb

```
[New Thread 3021]
[Switching to Thread 3021]

Breakpoint 1, hello_exit () at /home/zhongyifeng/Downloads/hello/hello.c:34
34      printk( "<0>Hello world exit!\n");
(gdb)
```

这时我们就可以进行驱动调试了，这种方法存在一定的缺陷。