

# 内核调试

# 实验介绍

1. 实验背景
2. 实验目的
3. 实验原理
4. GDB详解
5. 实验流程
6. 附加实验

# 实验背景

- 大家在编写程序的时候当遇到问题的时候都会使用调试工具来进行调试，一般都是使用IDE集成的调试工具进行调试。
- 本次实验将会在Ubuntu上基于qemu来进行Linux内核的调试，需要两个操作系统完成Linux内核的调试。

# 实验目的

搭建Linux内核调试的一整套环境，实现使用gdb对linux内核的调试。完成对configured变量的watch，并显示出来。

*注：configured是gdb向内核注册模块中的函数中的一个变量*

# 实验原理

kgdb提供了一种使用 gdb调试 Linux 内核的机制。使用 KGDB可以像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。使用KGDB调试时需要两台机器，一台作为开发机（Development Machine），另一台作为目标机（Target Machine），两台机器之间通过串口或者以太网口相连。调试过程中，被调试的内核运行在目标机上，gdb调试器运行在开发机上。

本次实验的开发机是运行在Ubuntu，在Ubuntu中安装 qemu用于运行目标机，其中也是用到了busybox工具。

# 实验原理

目前内核都集成了kgdb调试环境，gdb远程调试所需要的功能包括命令处理、陷阱处理及串口通讯3个主要的部分。

kgdb的主要作用是在Linux内核中添加了一个调试Stub。调试Stub是Linux内核中的一小段代码，提供了运行gdb的开发机和所调试内核之间的一个媒介。

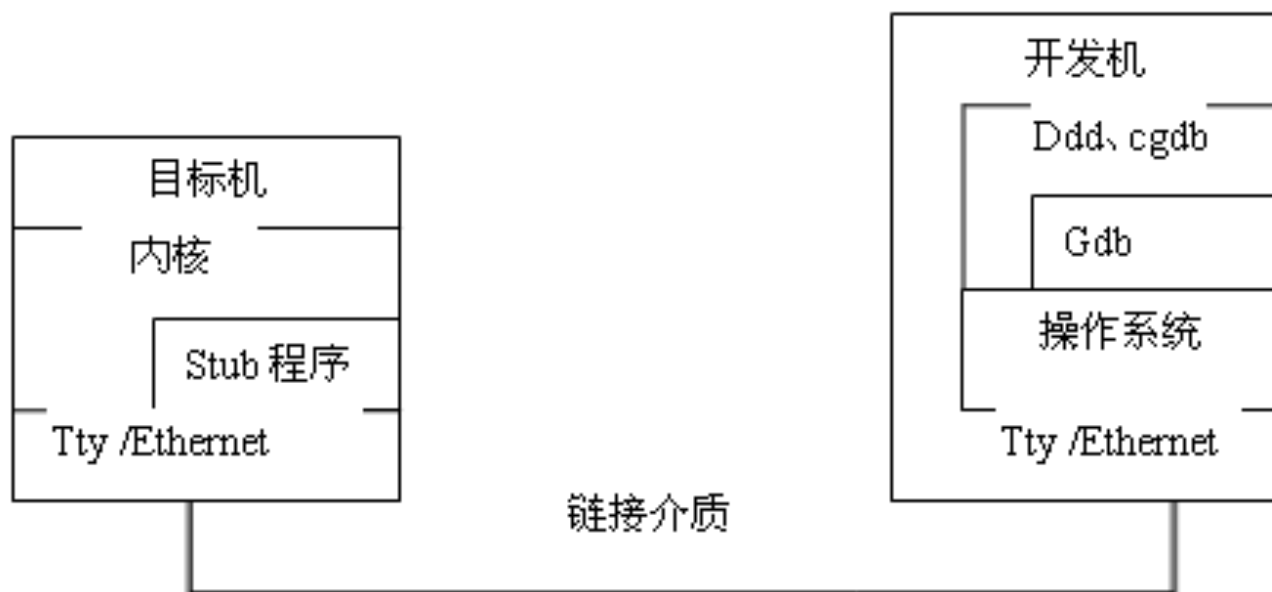
gdb和调试stub之间通过gdb串行协议进行通讯。gdb串行协议是一种基于消息的ASCII码协议，包含了各种调试命令。

# 实验原理

当设置断点时，kgdb负责在设置断点的指令前增加一条**trap指令**，当执行到断点时控制权就转移到调试stub中去。

此时，调试stub的任务就是使用远程串行通信协议将当前环境传送给gdb，然后从gdb处接受命令。

gdb命令告诉stub下一步该做什么，当stub收到继续执行的命令时，将恢复程序的运行环境，把对CPU的控制权重新交还给内核。





# GDB详解

## 运行

- **run**：简记为 **r**，其作用是运行程序，当遇到断点后，程序会在断点处停止运行，等待用户输入下一步的命令。
- **continue**（简写 **c**）：继续执行，到下一个断点处（或运行结束）
- **next**：（简写 **n**），单步跟踪程序，当遇到函数调用时，也不进入此函数体；此命令同 **step** 的主要区别是，**step** 遇到用户自定义的函数，将步进到函数中去运行，而 **next** 则直接调用函数，不会进入到函数体内。
- **step**（简写 **s**）：单步调试如果有函数调用，则进入函数；与命令 **n** 不同，**n** 是不进入调用的函数的
- **until**：当你厌倦了在一个循环体内单步跟踪时，这个命令可以运行程序直到退出循环体。
- **until+行号**：运行至某行，不仅仅用来跳出循环
- **finish**：运行程序，直到当前函数完成返回，并打印函数返回时的堆栈地址和返回值及参数值等信息。
- **call 函数(参数)**：调用程序中可见的函数，并传递“参数”，如：`call gdb_test(55)`
- **quit**：简记为 **q**，退出 **gdb**

# GDB详解

## 设置断点

- **break n** ( 简写 **b n** ) : 在第n行处设置断点
- ( 可以带上代码路径和代码名称 : **b OAGUPDATE.cpp:578** )
- **b fn1 if a > b** : 条件断点设置
- **break func** ( **break**缩写为**b** ) : 在函数func()的入口处设置断点, 如 : **break cb\_button**
- **delete 断点号n** ( **delete**缩写为**d** ) : 删除第n个断点
- **disable 断点号n** : 暂停第n个断点
- **enable 断点号n** : 开启第n个断点
- **clear 行号n** : 清除第n行的断点
- **info b** ( **info breakpoints** ) : 显示当前程序的断点设置情况
- **delete breakpoints** : 清除所有断点

# GDB详解

## 查看源代码

- **list** : 简记为 **l** , 其作用就是列出程序的源代码, 默认每次显示10行。
- **list** 行号 : 将显示当前文件以 “行号” 为中心的前后10行代码, 如: `list 12`
- **list** 函数名 : 将显示 “函数名” 所在函数的源代码, 如: `list main`
- **list** : 不带参数, 将接着上一次 `list` 命令的, 输出下边的内容。

# GDB详解

## 打印表达式

- **print 表达式**：简记为 `p`，其中“表达式”可以是任何当前正在被测试程序的有效表达式，比如当前正在调试C语言的程序，那么“表达式”可以是任何C语言的有效表达式，包括数字，变量甚至是函数调用。
- `print a`：将显示整数 `a` 的值
- `print ++a`：将把 `a` 中的值加1,并显示出来
- `print name`：将显示字符串 `name` 的值
- `print gdb_test(22)`：将以整数22作为参数调用 `gdb_test()` 函数
- `print gdb_test(a)`：将以变量 `a` 作为参数调用 `gdb_test()` 函数
- **display 表达式**：在单步运行时将非常有用，使用`display`命令设置一个表达式后，它将在每次单步进行指令后，紧接着输出被设置的表达式及值。如：`display a`
- **watch 表达式**：设置一个监视点，一旦被监视的“表达式”的值改变，`gdb`将强行终止正在被调试的程序。如：`watch a`
- `whatis`：查询变量或函数
- `info function`：查询函数
- `info locals`：显示当前堆栈页的所有变量

# GDB详解

## 查询运行信息

- where/bt ：当前运行的堆栈列表；
- bt **backtrace** 显示当前调用堆栈
- up/down 改变堆栈显示的深度
- set args 参数:指定运行时的参数
- show args ：查看设置好的参数
- info program ：来查看程序的是否在运行，进程号，被暂停的原因。

# 实验流程

- 内核编译（准备）
- 安装ddd（可选）
- busybox制作根文件系统
- gdb + qemu调试内核

# 内核编译(准备)

这一步在第一章的课后作业中有，这里就不在重复了，要注意的是在设置config的时候要保证KGDB是开启的。

```
.config - Linux/x86 5.3.0 Kernel Configuration
> Kernel hacking > KGDB: kernel debugger

KGDB: kernel debugger
Arrow keys navigate the menu. <Enter> selects submenus ---> (or empty
submenus ----). Highlighted letters are hotkeys. Pressing <Y>
includes, <N> excludes, <M> modularizes features. Press <Esc><Esc> to
exit, <?> for Help, </> for Search. Legend: [*] built-in [ ]

-- KGDB: kernel debugger
<*> KGDB: use kgdb over the serial console
[ ] KGDB: internal test suite
[*] KGDB: Allow debugging with traps in notifiers
[*] KGDB_KDB: include kdb frontend for kgdb
(0x1) KDB: Select kdb command functions to be enabled by default
[*] KGDB_KDB: keyboard as input device
(0) KDB: continue after catastrophic errors

<Select> < Exit > < Help > < Save > < Load >
```

注：如果之前有编译过的，这里勾选过的，就不需要重新编译了，如果没有选择的需要重新编译

# 内核编译(准备)

运行如下命令复制bzImage、vmlinux、initrd.img-version\_number到kDebug

```
>> cd ~
```

```
>> mkdir kDebug
```

```
>> cd kDebug
```

```
>> cp ../linux-5.3/vmlinux .
```

```
>> cp ../linux-5.3/arch/x86/boot/bzImage .
```

```
>> cp /boot/initrd.img-5.3.0 .
```



# 安装DDD(可选)

GDB本身是一种命令行调试工具，但是通过DDD(Data Display Debugger)可以图形化界面

下载地址：

<https://www.gnu.org/software/ddd/>



# busybox制作根文件系统

## **Busybox**

BusyBox是一个遵循GPL协议、以自由软件形式发行的应用程序。Busybox在单一的可执行文件中提供了精简的Unix工具集，可运行于多款POSIX环境的操作系统，例如Linux（包括Android）、Hurd、FreeBSD等等。

由于BusyBox可执行文件尺寸小、并通常使用Linux内核，这使得它非常适合使用于嵌入式系统。

此外，由于BusyBox功能强大，有些人将BusyBox称为“嵌入式Linux的瑞士军刀”。

# 安装busybox

## 下载

<https://www.busybox.net/downloads/>

## 解压

>> `tar -jxvf busybox-1.31.1.tar.bz2`(根据版本号解压)

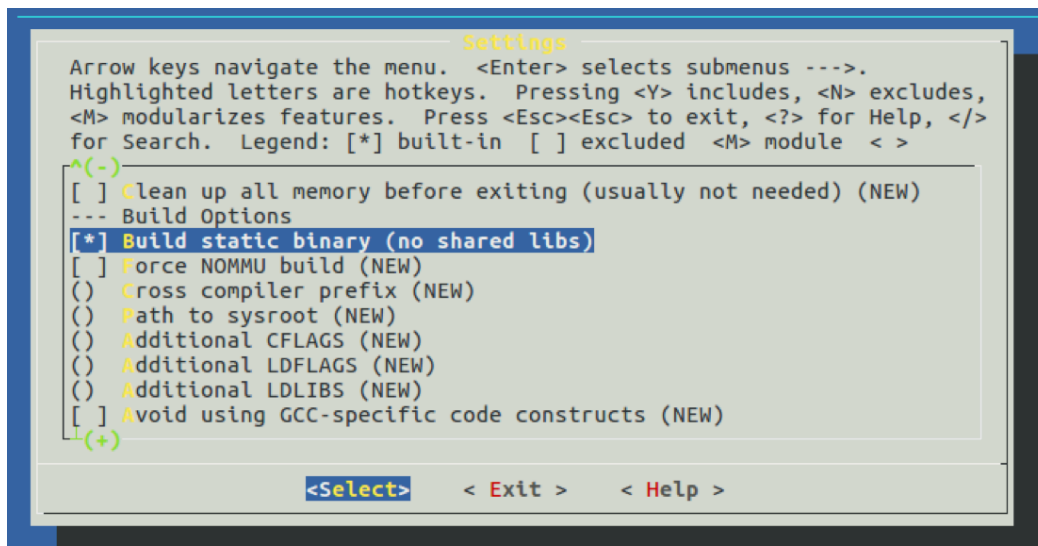
## 配置、编译，安装

>> `make menuconfig`

要保证图中选项打开：

>> `make`

>> `sudo make install`



# busybox制作根文件系统

该部分可参照实验手册中的详细步骤

亦可在网上查阅相关教程

完成后会得到一个busybox.img文件系统镜像

# gdb + qemu调试内核

## 安装qemu

QEMU是一套由Fabrice Bellard所编写的模拟处理器的自由软件。它与Bochs, PearPC近似, 但其具有某些后两者所不具备的特性, 如高速度及跨平台的特性。经由KVM (早期为kqemu加速器, 现在kqemu已被KVM替换) 这个开源的加速器, QEMU能模拟至接近真实电脑的速度。

运行如下命令进行安装：

```
>> sudo apt-get install qemu
```

# gdb + qemu调试内核

## qemu启动目标机

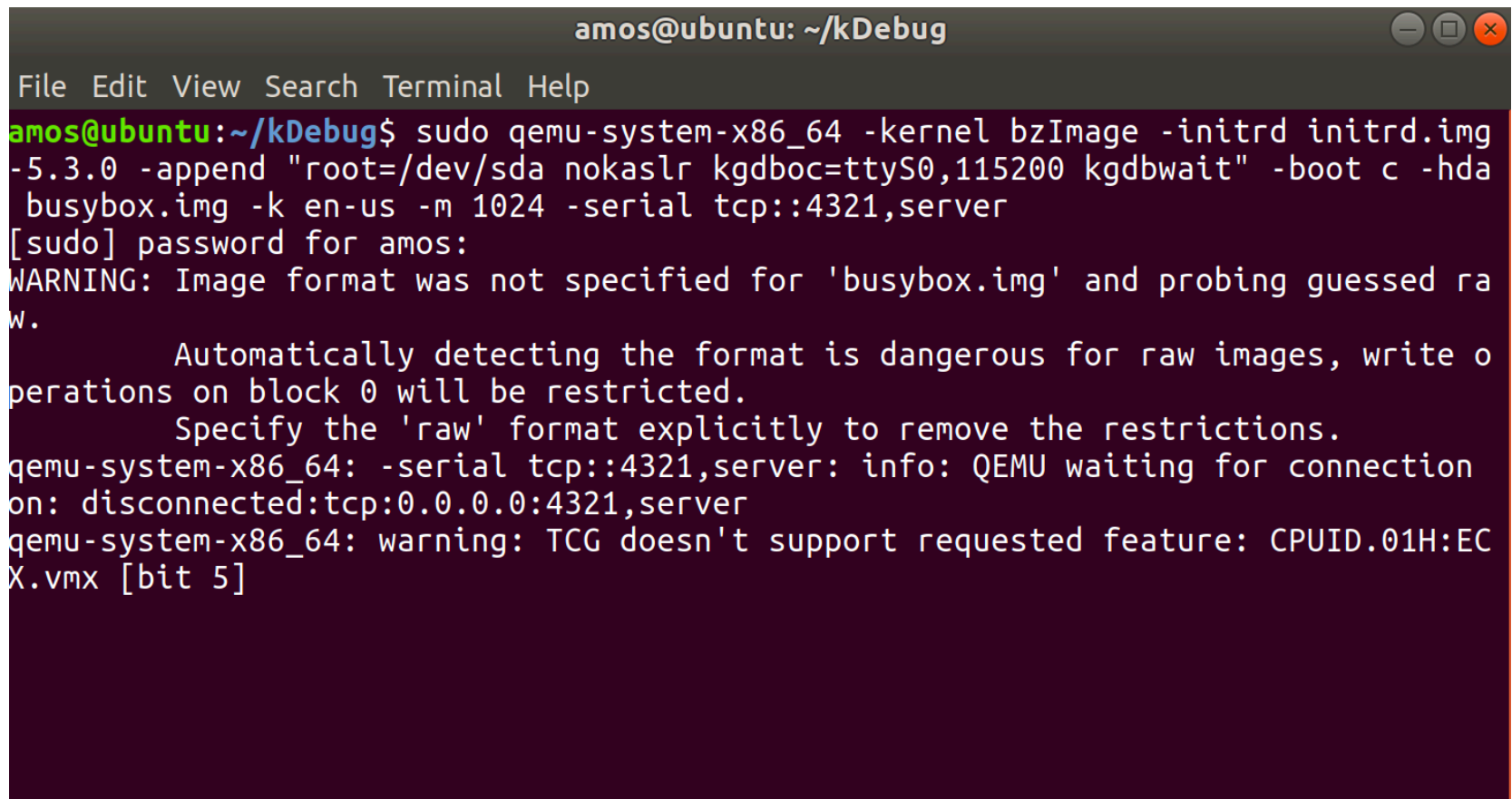
kgdb可以在内核启动时增加参数(也可以在内核启动后echo kgdboc模块的参数来达到目的), (kgdboc=ttyS0,115200 kgdbwait)的方式, 运行如下命令使qemu运行自己编译的内核:

```
sudo qemu-system-x86_64 -kernel bzImage -initrd initrd.img-5.3.0 -append  
"root=/dev/sda nokaslr kgdboc=ttyS0,115200 kgdbwait" -boot c -hda  
busybox.img -k en-us -m 1024 -serial tcp::4321,server
```

其中-initrd指定引导镜像, -append指定启动参数(其中nokaslr 是关闭内核地址随机化, 不指定这个参数可能造成gdb无法找到断点位置, 导致无法在设置的断点处暂停, 影响调试), -boot c指定磁盘引导, -hda (hard disk a)指定镜像文件, -k指定语言, -m指定内存(Mb为单位), -serial指定当前运行qemu的终端将提示等待远程连接到本地端口4321:  
QEMU waiting for connection on: tcp:0.0.0.0:4321,server

# gdb + qemu调试内核

## qemu启动目标机



```
amos@ubuntu: ~/kDebug
File Edit View Search Terminal Help
amos@ubuntu:~/kDebug$ sudo qemu-system-x86_64 -kernel bzImage -initrd initrd.img
-5.3.0 -append "root=/dev/sda nokaslr kgdboc=ttyS0,115200 kgdbwait" -boot c -hda
busybox.img -k en-us -m 1024 -serial tcp::4321,server
[sudo] password for amos:
WARNING: Image format was not specified for 'busybox.img' and probing guessed ra
w.
    Automatically detecting the format is dangerous for raw images, write o
perations on block 0 will be restricted.
    Specify the 'raw' format explicitly to remove the restrictions.
qemu-system-x86_64: -serial tcp::4321,server: info: QEMU waiting for connection
on: disconnected:tcp:0.0.0.0:4321,server
qemu-system-x86_64: warning: TCG doesn't support requested feature: CPUID.01H:EC
X.vmx [bit 5]
```

# gdb + qemu调试内核

## 开发机中调试

在开发机中(在kDebug中再打开一个终端)使用如下命令：

```
>> gdb vmlinux
```

```
>> (gdb) target remote localhost:4321
```

此步骤也可以在ddd中完成，可以watch一个变量，然后打印出它的值（远程连接可能会失败，可以再次尝试即可）



# gdb + qemu调试内核

## 开发机中调试

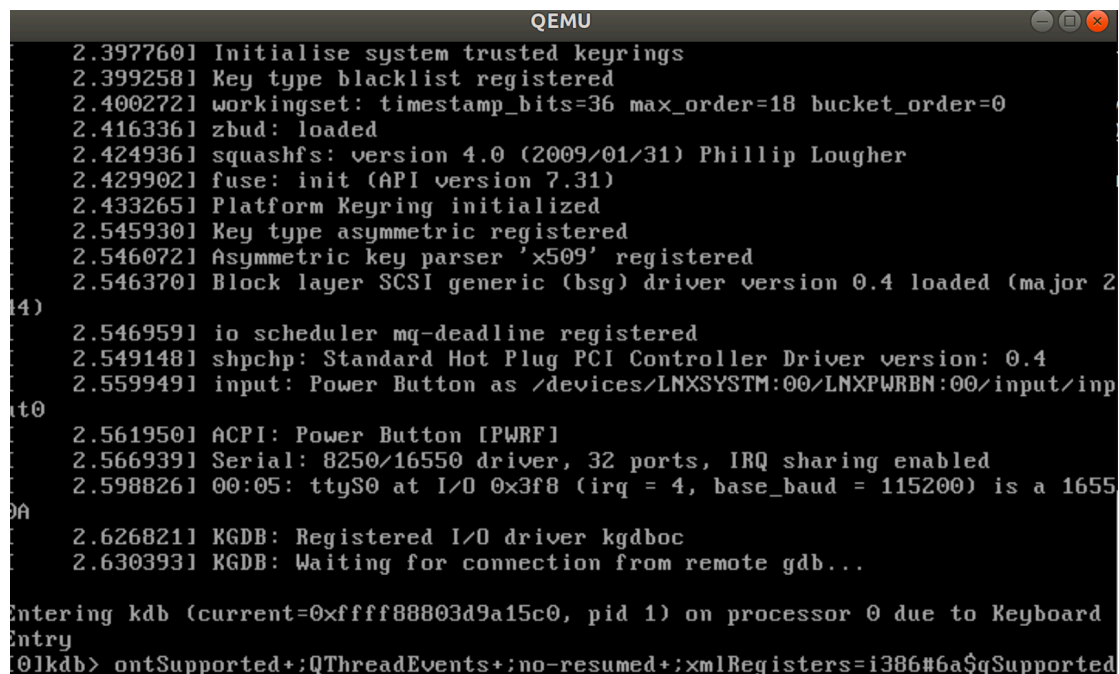
运行完成后同时出现如下两个界面，就说明环境搭建完成，可以输入gdb的命令进行调试了

```
GNU gdb (Ubuntu 8.1-0ubuntu3.2) 8.1.0.20180409-git
Copyright (C) 2018 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from vmlinux...target localdone.
l(gdb) target localhost:4321
Undefined target command: "localhost:4321".  Try "help target".
(gdb) target remote localhost:4321
Remote debugging using localhost:4321
kgdb_breakpoint () at kernel/debug/debug_core.c:1136
1136          wmb(); /* Sync point after breakpoint */
(gdb) █
```

# gdb + qemu调试内核

## 开发机中调试

运行完成后同时出现如下两个界面，就说明环境搭建完成，可以输入gdb的命令进行调试了

A screenshot of a QEMU terminal window. The window title is "QEMU". The terminal displays a series of kernel boot logs with timestamps. The logs include messages for initializing trusted keyrings, registering key types, setting up the workingset, loading zbud, initializing squashfs, fuse, Platform Keyring, and the Block layer SCSI generic driver. It also shows the registration of the io scheduler mq-deadline, the shpchp driver, and the input Power Button. The logs end with "KGDB: Registered I/O driver kgdboc" and "KGDB: Waiting for connection from remote gdb...". Below the logs, a message indicates that the user is entering kdb on processor 0 due to a keyboard entry. The prompt is "0lkdb>".

```
QEMU
2.3977601 Initialise system trusted keyrings
2.3992581 Key type blacklist registered
2.4002721 workingset: timestamp_bits=36 max_order=18 bucket_order=0
2.4163361 zbud: loaded
2.4249361 squashfs: version 4.0 (2009/01/31) Phillip Lougher
2.4299021 fuse: init (API version 7.31)
2.4332651 Platform Keyring initialized
2.5459301 Key type asymmetric registered
2.5460721 Asymmetric key parser 'x509' registered
2.5463701 Block layer SCSI generic (bsg) driver version 0.4 loaded (major 2
14)
2.5469591 io scheduler mq-deadline registered
2.5491481 shpchp: Standard Hot Plug PCI Controller Driver version: 0.4
2.5599491 input: Power Button as /devices/LNXSYSTM:00/LNXPWRBN:00/input/inp
t0
2.5619501 ACPI: Power Button [PWRB]
2.5669391 Serial: 8250/16550 driver, 32 ports, IRQ sharing enabled
2.5988261 00:05: ttyS0 at I/O 0x3f8 (irq = 4, base_baud = 115200) is a 1655
0A
2.6268211 KGDB: Registered I/O driver kgdboc
2.6303931 KGDB: Waiting for connection from remote gdb...

Entering kdb (current=0xffff88803d9a15c0, pid 1) on processor 0 due to Keyboard
Entry
0lkdb> ontSupported+;QThreadEvents+;no-resumed+;xmlRegisters=i386#6a$gSupported
```

# gdb + qemu调试内核

## watch configured

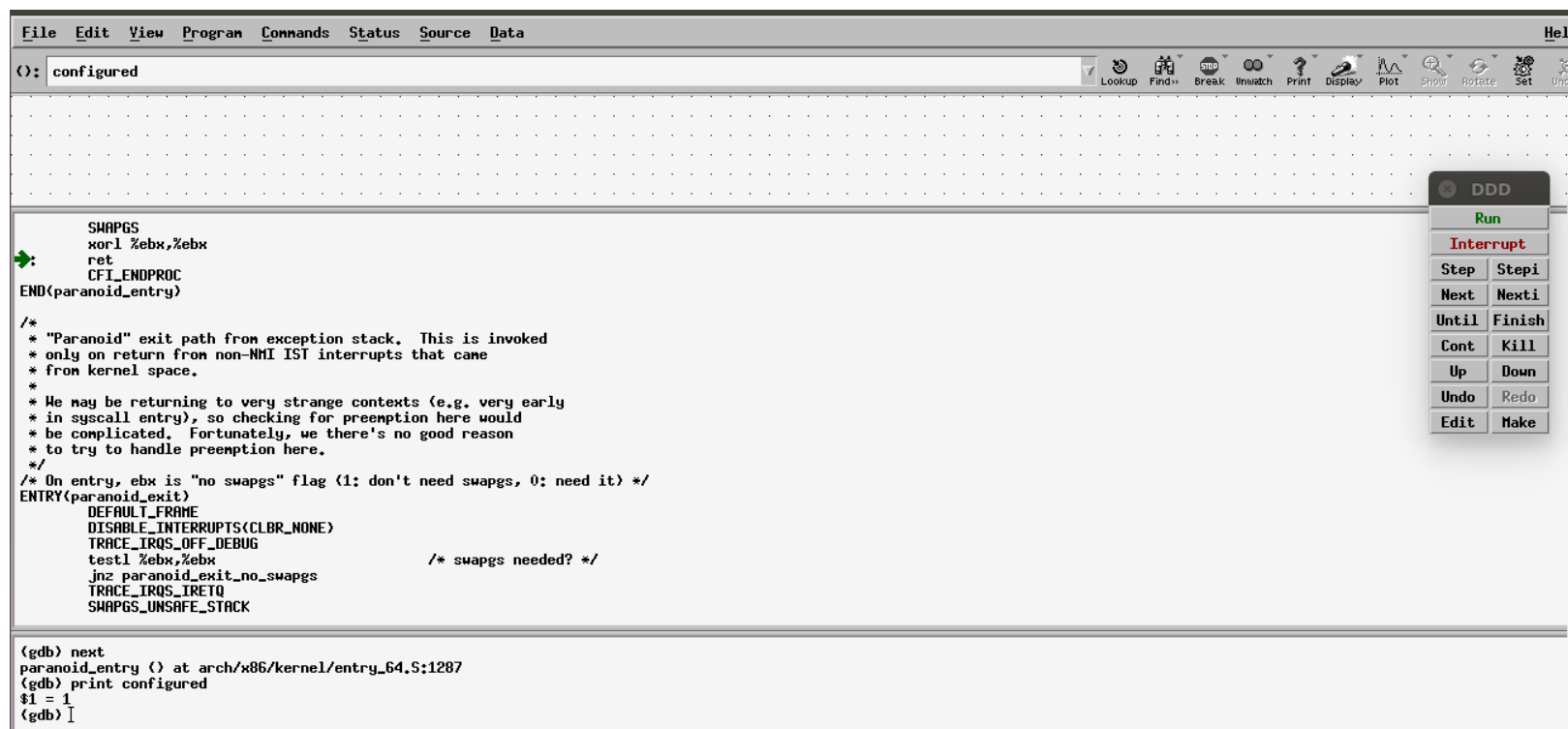
一直输入n执行下一步，当遇到一个变量的时候，例如 configured，输入 watch configured 对这个变量进行监视，然后输入 print configured 输出它的值

```
188         if (err)
(gdb)
187         err = kgdb_register_io_module(&kgdboc_io_ops);
(gdb)
188         if (err)
(gdb)
191         err = kgdb_register_nmi_console();
(gdb)
192         if (err)
(gdb)
191         err = kgdb_register_nmi_console();
(gdb)
192         if (err)
(gdb)
195         configured = 1;
(gdb) watch configured
Hardware watchpoint 1: configured
(gdb) n
debug () at arch/x86/entry/entry_64.S:1192
1192     idtentry debug                do_debug                has_error_code=0
paranoid=1 shift_ist=IST_INDEX_DB ist_offset=DB_STACK_OFFSET
(gdb) print configured
$1 = 1
(gdb)
```

# gdb + qemu调试内核

## watch configured

也可通过ddd来完成，更加直观方便（可选）



The screenshot shows a GDB session with the 'configured' function selected in the top toolbar. The main window displays the assembly and C code for the 'paranoid\_entry' function. The assembly code includes instructions like 'xorl %ebx,%ebx', 'ret', and 'CFI\_ENDPROC'. The C code is a comment block describing the 'paranoid' exit path. The bottom status bar shows the current location in the kernel source file: 'arch/x86/kernel/entry\_64.S:1287'. A DDD (Data Display Debugger) window is open on the right side, showing a 'Run' button and a table of memory addresses and values.

```
File Edit View Program Commands Status Source Data Help
(): configured

SWAPGS
xorl %ebx,%ebx
ret
CFI_ENDPROC
END(paranoid_entry)

/*
 * "Paranoid" exit path from exception stack. This is invoked
 * only on return from non-NMI IST interrupts that came
 * from kernel space.
 *
 * We may be returning to very strange contexts (e.g. very early
 * in syscall entry), so checking for preemption here would
 * be complicated. Fortunately, we there's no good reason
 * to try to handle preemption here.
 */
/* On entry, ebx is "no swaps" flag (1: don't need swaps, 0: need it) */
ENTRY(paranoid_exit)
    DEFAULT_FRAME
    DISABLE_INTERRUPTS(CLR_NONE)
    TRACE_IRQS_OFF_DEBUG
    testl %ebx,%ebx          /* swaps needed? */
    jnz paranoid_exit_no_swaps
    TRACE_IRQS_IRETQ
    SWAPGS_UNSAFE_STACK

(gdb) next
paranoid_entry () at arch/x86/kernel/entry_64.S:1287
(gdb) print configured
$1 = 1
(gdb) I
```

# 附加实验

## 指定helloworld作为系统init入口，并对一个变量进行追踪

Linux内核获得控制权后，首先会加载initramfs到内存中，逐步由init获得系统控制权。那么，系统究竟是在哪里指定了init进行执行的呢？通过这次实验，你可以掌握这一点。

## 实验要求

通过课件和网上查阅资料，弄清楚并在实验报告中回答以下几个问题：

Q1：结合课件，描述一下Linux系统的启动过程是什么？

Q2：结合源代码，描述一下start\_kernel()函数是怎样一步步进入kernel\_init()中的。（写出函数调用结构层次）在kernel\_init()中做了哪些工作？（提示：在init/main.c中找答案）

Q3：在gdb调试中打印ramdisk\_execute\_command的变量值，并观察system\_state变量是怎么变化的？

# 实验流程

该部分可参照实验手册中的步骤

其中最后一步没有给出完整做法，为额外加分项，能描述清楚调用流程可以酌情给分

```
QEMU
6.960794] evm: security.SMACK64MMAP
6.960826] evm: security.apparmor
6.960858] evm: security.ima
6.960889] evm: security.capability
6.960930] evm: HMAC attrs: 0x1
6.965026] ata2.00: ATAPI: QEMU DVD-ROM, 2.5+, max UDMA/100
6.980271]   Magic number: 8:5:791
6.981331] rtc_cmos 00:00: setting system clock to 2020-02-16 15:46:37 UTC (
581867997)
6.982666] scsi 1:0:0:0: CD-ROM                QEMU      QEMU DVD-ROM    2.5+ PQ
0 ANSI: 5
7.050619] sr 1:0:0:0: [sr0] scsi3-mmc drive: 4x/4x cd/rw xa/form2 tray
7.050927] cdrom: Uniform CD-ROM driver Revision: 3.20
7.057689] sr 1:0:0:0: Attached scsi generic sg0 type 5
7.068541] Freeing unused decrypted memory: 2040K
7.098565] Freeing unused kernel image memory: 2444K
7.110160] Write protecting the kernel read-only data: 18432k
7.113818] Freeing unused kernel image memory: 2008K
7.114308] Freeing unused kernel image memory: 16K
7.186373] x86/mm: Checked W+X mappings: passed, no W+X pages found.
7.187074] Run /helloworld as init process
hello World
This is an entry
author:your own name
```