

内核调试

1. 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。

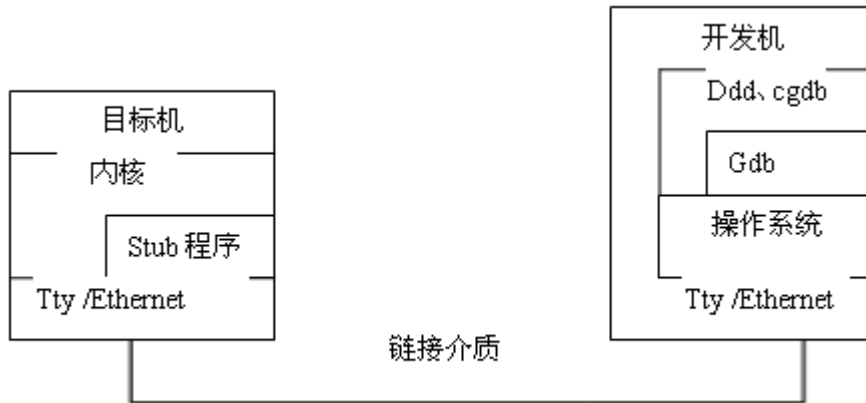
2. 实验目的

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。
并完成对某个变量（configured）的 watch，并显示出来（可选）。

3. 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。

安装 kgdb 调试环境需要为 Linux 内核应用 kgdb 补丁（目前内核都集成了这个，所以不用担心），补丁实现的 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 补丁的主要作用是在 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。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。



实验使用的 Linux 中已经拥有了 KGDB，只要在编译的时候选择即可，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。

4. Gdb 详解

4.1 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

4.2 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`：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：

4.3 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。

4.4 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
扩展 info locals：显示当前堆栈页的所有变量

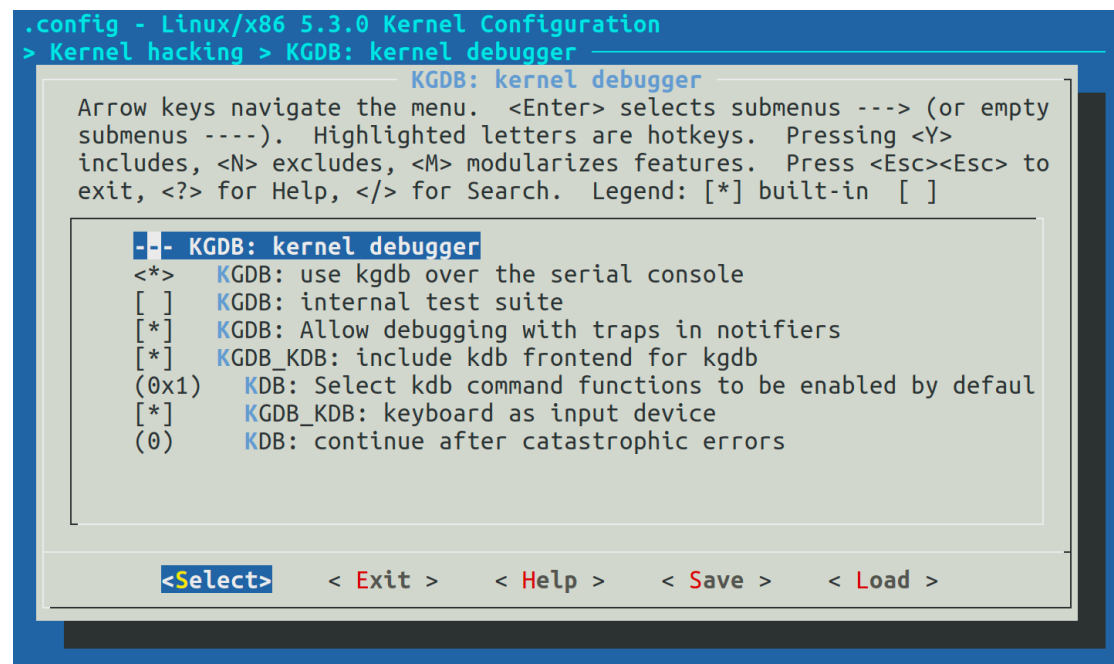
4.5 查询运行信息

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。

5. 实验流程

5.1. Linux 内核源码编译

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，如下图：



（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）

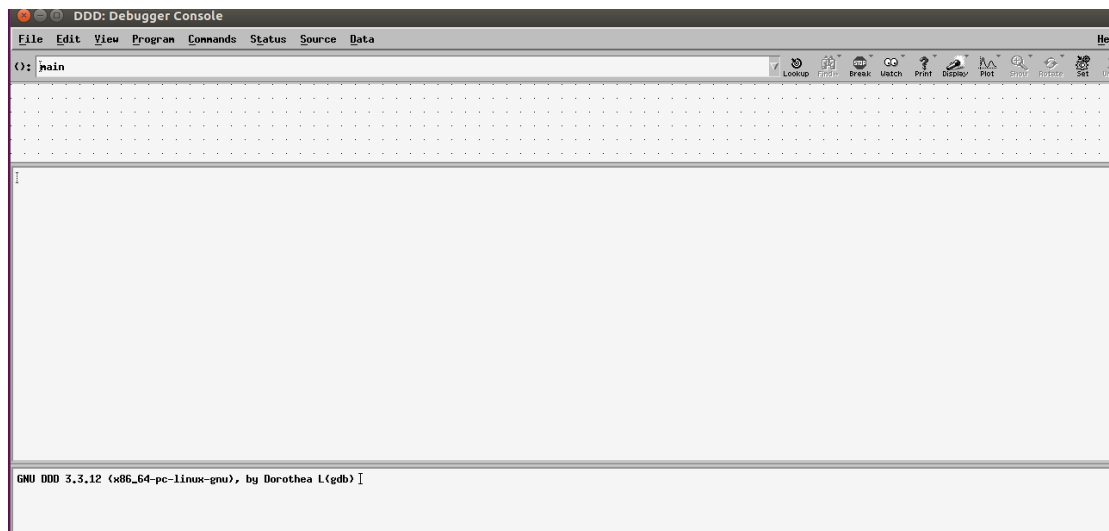
运行如下命令复制 bzImage、vmlinuz、initrd.img-version_number 进行备份

```
cd ~
mkdir kDebug
cd kDebug

cp ../linux-5.3/vmlinuz .
cp ../linux-5.3/arch/x86/boot/bzImage .
cp /boot/initrd.img-5.3.0 .
```

5.2. 安装 ddd

GDB 本身是一种命令行调试工具，但是通过 DDD(data Display Debugger, 地址是 <https://www.gnu.org/software/ddd/>)可以图形化界面(在文件有 deb 包，可以双击安装)



5.3. 制作自己的文件系统

5.3.1. Busybox

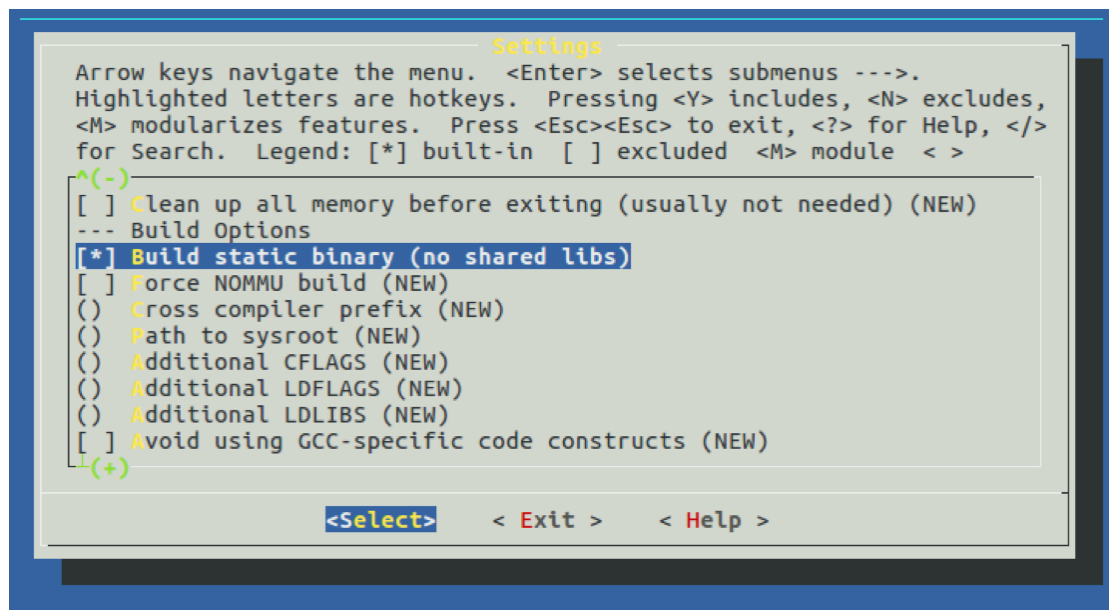
BusyBox 是一个遵循 GPL 协议、以自由软件形式发行的应用程序。Busybox 在单一的可执行文件中提供了精简的 Unix 工具集，可运行于多款 POSIX 环境的操作系统，例如 Linux（包括 Android）、Hurd、FreeBSD 等等。由于 BusyBox 可执行文件尺寸小、并通常使用 Linux 内核，这使得它非常适合使用于嵌入式系统。此外，由于 BusyBox 功能强大，因此有些人将 BusyBox 称为“嵌入式 Linux 的瑞士军刀”。

5.3.1.1. 下载、解压

到 <https://www.busybox.net/downloads/> 中下载最新的 busybox，并运行如下命令解压：
`tar -jxvf busybox-1.31.1.tar.bz2`(根据版本号解压)

5.3.1.2. 编译、安装

1. 运行 `make menuconfig` 进行配置(并保证图中的选项是选择了的)



2. 运行 make 进行编译
3. 运行 sudo make install 进行安装
4. 当前目录下的 busybox 就是运行文件

5.3.2. 制作文件系统

使用如下的命令制作一个文件系统

1. 进入 kDebug, 在当前目录下创建一个名为 busybox.img, 大小为 100M 的文件, 并将其格式化为 ext3 的文件系统

```
dd if=/dev/zero of=./busybox.img bs=1M count=100
mkfs.ext3 busybox.img
```

2. 将这个虚拟磁盘文件挂载到本地系统中, 这样我们可以像访问本地文件一样访问它, 并将生成好的 busybox 的文件拷贝到这个文件里。

```
sudo mkdir /mnt/disk
sudo mount -o loop ./busybox.img /mnt/disk
sudo cp -rf ~/busybox-1.31.1/_install/* /mnt/disk
```

3. 创建必须的文件系统目录

```
cd /mnt/disk/
sudo mkdir dev sys proc etc lib mnt
```

4. 创建 rcS 文件

```
cd etc/
sudo mkdir init.d
sudo vim /mnt/disk/etc/init.d/rcS
```

5. 将下面内容拷贝到 rcS 里，并加执行权限:

```
/sbin/mdev -s  
sudo chmod +x rcS
```

6. 做完上面对工作后，我们就可以卸载虚拟磁盘文件了

```
cd ~/kDebug/  
sudo umount /mnt/disk
```

5.4. 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
```

5.5. gdb + qemu 调试内核

1. qemu 启动目标机

kgdb 可以在内核启动时增加使能参数，也可以在内核启动后 echo kgdboc 模块的参数来达到目的，这里我们采取在内核启动时增加启动参数(kgdboc=ttyS0,115200 kgdbwait)的方式，运行如下命令使用 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  
nokalr kgdboc=ttyS0,115200 kgdbwait" -boot c -hda busybox.img -k en-us -m 1024 -serial  
tcp::4321,server
```

这时，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

```
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 raw.
        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]
```

2. 开发机中调试

在开发机中使用如下命令：

```
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，然后打印出她的值（第一次远程连接可能会失败，可以再运行一次 target remote localhost:4321）

运行完成后出现如下界面，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 locadone.
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) █
```



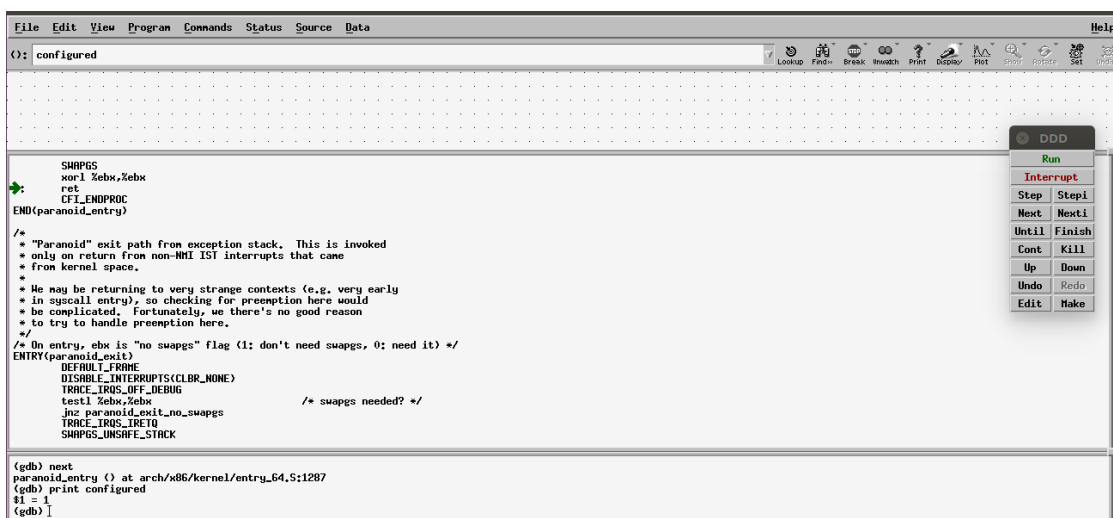
```
QEMU
2.397760] Initialise system trusted keyrings
2.399258] Key type blacklist registered
2.400272] workingset: timestamp_bits=36 max_order=18 bucket_order=0
2.416336] zbud: loaded
2.424936] squashfs: version 4.0 (2009/01/31) Phillip Lougher
2.429902] fuse: init (API version 7.31)
2.433265] Platform Keyring initialized
2.545930] Key type asymmetric registered
2.546072] Asymmetric key parser 'x509' registered
2.546370] Block layer SCSI generic (bsg) driver version 0.4 loaded (major 2
4)
2.546959] io scheduler mq-deadline registered
2.549148] shpchp: Standard Hot Plug PCI Controller Driver version: 0.4
2.559949] input: Power Button as /devices/LNXSYSTM:00/LNXPWRBN:00/input/inp
t0
2.561950] ACPI: Power Button [PWRF]
2.566939] Serial: 8250/16550 driver, 32 ports, IRQ sharing enabled
2.598826] 00:05: ttyS0 at I/O 0x3f8 (irq = 4, base_baud = 115200) is a 1655
0A
2.626821] KGDB: Registered I/O driver kgdboc
2.630393] 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
```

3. 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)
```

也可通过 *ddd* 来完成, 更加直观方便



6. 指定 helloworld 作为系统 init 入口，并观察一个变量进行追踪

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。

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？

(1) 编写一个 helloworld.c 程序

```
/hello.c/
#include <stdio.h>
void main()
{
    printf( "Hello World\n" );
    printf( "This is an entry\n" );
    printf( "Author:fenghao\n" );
    fflush(stdout);
    while(1);
}
```

(2) gcc 编译

```
gcc -static -o helloworld helloworld.c
```

(3) 将 helloworld 制作成 cpio

```
echo helloworld | cpio -o --format=newc > hellofs
```

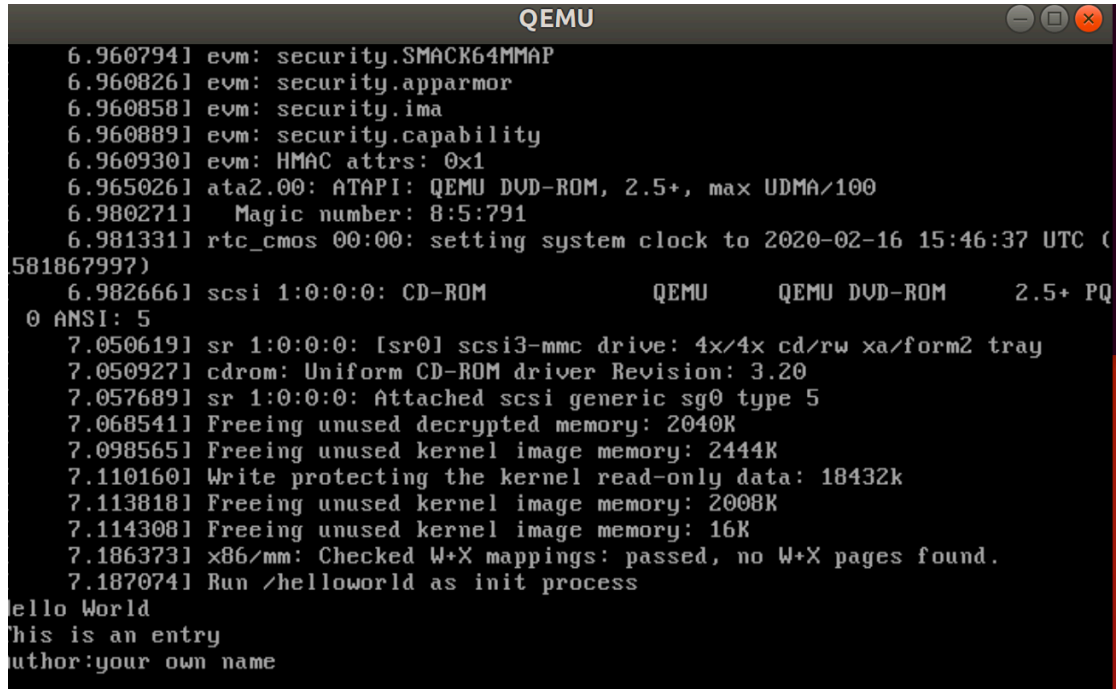
(4) 查看 hellofs

ls -lah rootfs

(5) qemu 启动目标机

*sudo qemu-system-x86_64 -kernel bzImage -initrd hellofs -append "root=/dev/sda
rdinit=/helloworld"*

(6) 观察 qemu 输出结果



```
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
ello World
his is an entry
uthor:your own name
```

(7) 可以看到，系统进入了我们指定的 helloworld 中，并且通过循环停在该处。通过以上这些步骤，我们就可以指定内核加载之后执行我们所指定的程序。

(8) 用调试模式启动，为该过程加入断点，调试：系统是如何通过指定的参数一步步进入我们指定的 helloworld 中的？系统的当前状态 system_state 变量是如何变化的？（该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）

*sudo qemu-system-x86_64 -kernel bzImage -initrd hellofs -append "root=/dev/sda
rdinit=/helloworld nokaslr kgdboc=ttyS0,115200 kgdbwait" -k en-us -m 1024 -serial
tcp::4321,server*