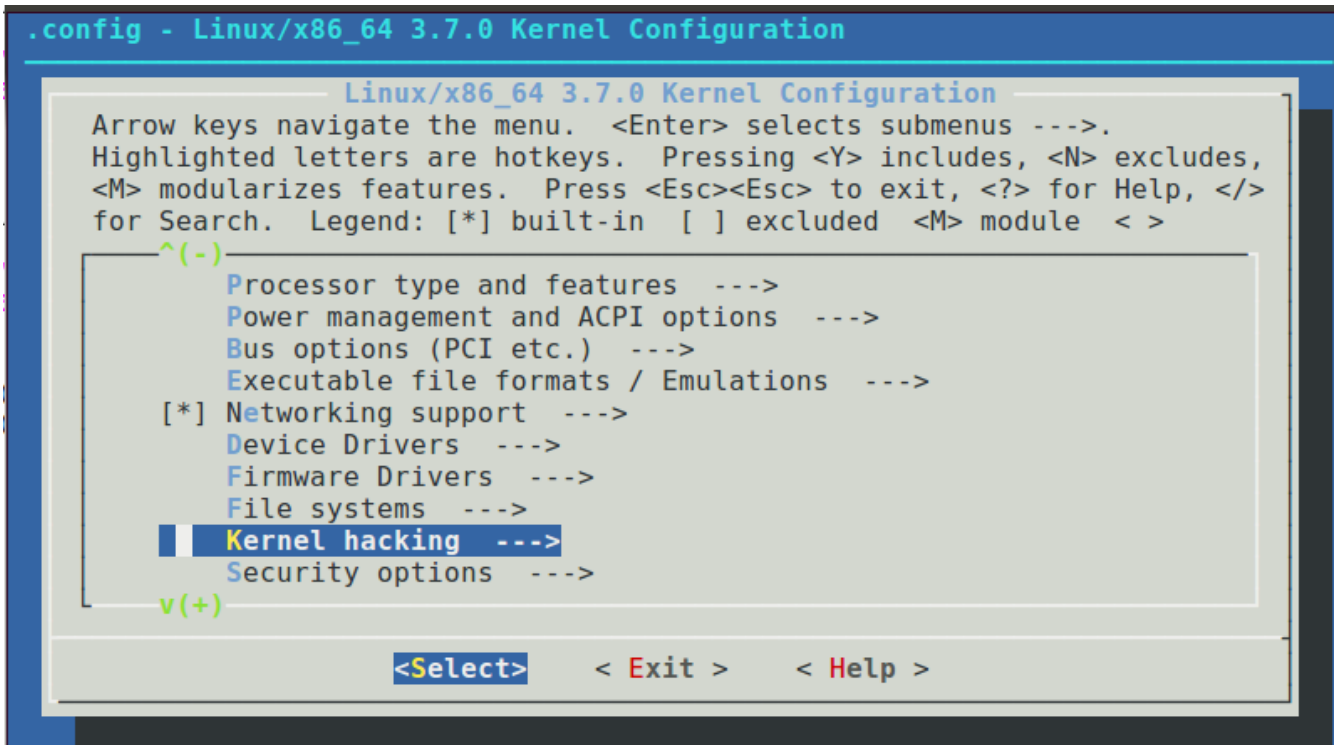
linux内核调试方法

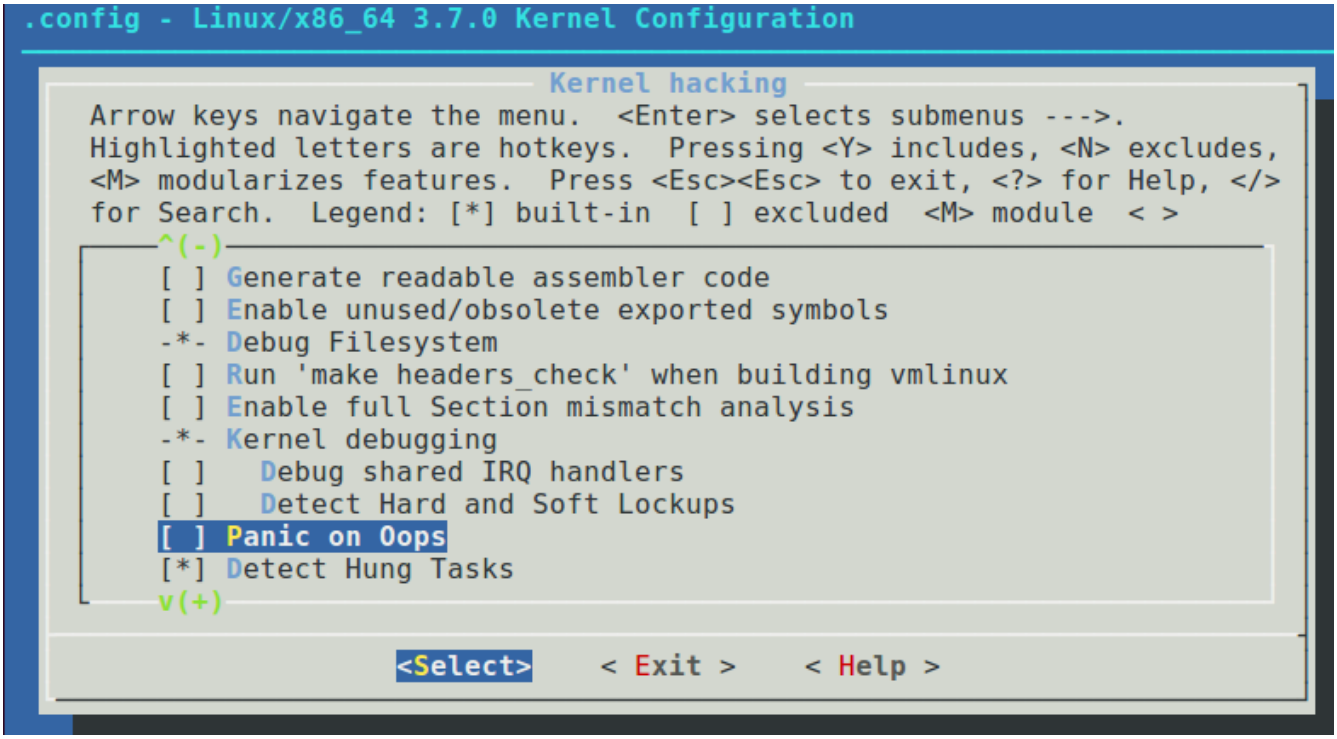
一、内核调试方法概述

内核调试方法有很多，主要有以下四类。

- 内核打印函数。
- 通过特殊文件系统获取内核信息
- 处理出错信息
- 利用KGDB调试源码
- sysrq系统请求键
- frace追踪(一种静态追踪工具)
- 动态追踪

  这些内核调试功能也通常需要在配置内核时选择，如下图所示，在Kernel hacking菜单下进行各种调试功能配置。





二、内核打印函数

1、printk工作原理概述

内核打印函数printk是调试内核、驱动最简单、最广泛的方法。printk函数不是直接向控制台设备发送打印信息，而是把打印信息先写到一个缓冲区log_buf中，然后内核会根据信息的级别按合适的顺序将满足打印级别的信息打印到控制台设备。 在控制台初始化前所有符合和不符合打印级别的信息都只能自在log_buf中，直到控制台初始化后才将符合打印级别的信息发送到控制台输出，如果在控制台初始化之前系统崩溃的话，将不会在控制台上看到本该输出的信息。 没有满足打印级别的信息不会自动输出到控制台，但还保留在log_buf中，此时**可以通过dmesg查看log_buf中完整的信息**，包括已经输出到控制台的信息，只要还没有被覆盖。还可以**通过cat /proc/kmsg实时查看log_buf中的信息（无论是否达到输出控制台的级别）**。

2、printk的打印级别

printk发送到log_buf中的信息有如下8种级别：

```
#define KERN_EMERG  KERN_SOH "0"    /* system is unusable */
#define KERN_ALERT  KERN_SOH "1"    /* action must be taken immediately */
#define KERN_CRIT   KERN_SOH "2"    /* critical conditions */
#define KERN_ERR    KERN_SOH "3"    /* error conditions */
#define KERN_WARNING    KERN_SOH "4"    /* warning conditions */
#define KERN_NOTICE KERN_SOH "5"    /* normal but significant condition */
#define KERN_INFO   KERN_SOH "6"    /* informational */
#define KERN_DEBUG  KERN_SOH "7"    /* debug-level messages */
```

其中数字越小代表级别越高，在include/linux/printk.h中定义了如下宏:

```
#define console_loglevel (console_printk[0])
#define default_message_loglevel (console_printk[1])
#define minimum_console_loglevel (console_printk[2])
#define default_console_loglevel (console_printk[3])
```

第一个宏表示printk中的级别数字比console_loglevel小才会输出到控制台，第二个宏表示如果printk中没有加级别则该消息级别就是default_message_loglevel。第3和第4个宏不常用，暂不讨论。当系统启动挂载了proc文件系统后，这4个值可以从**/proc/sys/kernel/printk**读取或者更改。系统启动后通过cat指令看到的值如下，说明所有级别的信息都会输出到控制台，因为都比8小，printk缺省的级别为4。

```
# cat /proc/sys/kernel/printk
8   4   1   7
```

```
#include <linux/init.h>
#include <linux/module.h>
MODULE_LICENSE("GPL");

static int __init yeshen_init(void)
{
    printk("yeshen's first driver coming\n");

    printk(KERN_DEBUG "debug-level messages\n");
    printk(KERN_INFO "informational\n");
    printk(KERN_NOTICE "normal but significant condition\n");
    printk(KERN_ERR "error conditions\n");
    printk(KERN_CRIT "critical conditions\n");
    printk(KERN_ALERT "action must be taken immediately\n");
    printk(KERN_EMERG "system is unusable\n");

    return 0;

}

static int __exit yeshen_exit(void)
{
    printk("yeshen's first driver leaving\n");
    return 0;
}

module_init(yeshen_init);
module_exit(yeshen_exit);
```

我通过以上一个模块示例来展示，通过insmod加载模块后发现模块加载函数中所有printk信息都打印了出来，如下图。

然后将触发打印级别8改为4：

```
# echo "4 4 1 7" > /proc/sys/kernel/printk
```

再重新加载模块，通过如下打印信息可以发现级别大于和等于4的信息都没有显示出来，接着再将触发打印级别改为5再重新加载模块发现级别等于4的信息又显示出来了。

```
# echo "5 4 1 7" > /proc/sys/kernel/printk
```

三、通过特殊文件系统获取内核信息

linux内核提供了一些与用户空间通信的机制，可供应用程序和驱动程序利用。主要使用的两种机制是proc和sysfs文件系统。

1、proc文件系统

proc文件系统是一种伪文件系统，系统运行时在内存创建，实时提供系统的状态信息，关机或者掉电后消失。linux中的许多工具的工作就是读取和回报proc文件系统中的信息，比如ps和top。 proc文件系统通常挂载在/proc目录，可手工在shell挂载或者在/etc/fstab中设置使得开机自动挂载。挂载命令如下：

```
# mount -t proc /proc /proc
```

mount命令的一般形式为：

```
mount [-t fstype] something somewhere
```

可以将第一个/proc替换成none，如：

```
# mount -t proc none /proc
```

因为proc是个伪文件系统，不是一个真实的设备。 proc文件系统中各种文件对应不同的内核信息，通过读取这些文件获取相应的内核信息，如甚至可以通过更改某些文件更改内核的状态和行为，比如前文通过cat和echo指令对/proc/sys/kernel/printk的读取和修改：

```
# cat /proc/sys/kernel/printk
# echo "4 4 1 7" > /proc/sys/kernel/printk
```

注意不要使用文本编辑器访问，因为这些文件是实时变化的，用文本编辑器访问会数据不同步。 /proc/n目录是进程号为n的进程对应的条目，里面包含了许多该进程的信息。如进程1信息:

```
# ls -l /proc/1
total 0
dr-xr-xr-x 2 root root 0 Jul 10 01:36 attr
-rw-r--r-- 1 root root 0 Jul 10 18:43 autogroup
-r-------- 1 root root 0 Jul 10 18:43 auxv
-r--r--r-- 1 root root 0 Jul 10 01:36 cgroup
--w------- 1 root root 0 Jul 10 18:43 clear_refs
-r--r--r-- 1 root root 0 Jul 10 01:31 cmdline
-rw-r--r-- 1 root root 0 Jul 10 01:36 comm
-rw-r--r-- 1 root root 0 Jul 10 18:43 coredump_filter
-r--r--r-- 1 root root 0 Jul 10 18:43 cpuset
```

```
lrwxrwxrwx 1 root root 0 Jul 10 18:43 cwd -> /
-r-------- 1 root root 0 Jul 10 07:23 environ
lrwxrwxrwx 1 root root 0 Jul 10 01:36 exe -> /lib/systemd/systemd
dr-x------ 2 root root 0 Jul 10 02:02 fd
dr-x------ 2 root root 0 Jul 10 18:43 fdinfo
-rw-r--r-- 1 root root 0 Jul 10 18:43 gid_map
-r-------- 1 root root 0 Jul 10 18:43 io
-r--r--r-- 1 root root 0 Jul 10 02:17 limits
-rw-r--r-- 1 root root 0 Jul 10 01:36 loginuid
dr-x------ 2 root root 0 Jul 10 18:43 map_files
-r--r--r-- 1 root root 0 Jul 10 18:43 maps
-rw------- 1 root root 0 Jul 10 18:43 mem
-r--r--r-- 1 root root 0 Jul  9 18:00 mountinfo
.....
```

里面包含了进程1的各种信息，比如环境变量environ，启动该进程的命令行cmdline，运行状态信息status。对调试来说一个比较有用的条目是maps。

```
# cat /proc/1/maps
55dbccdf6000-55dbccf44000 r-xp 00000000 08:01 6296083
 /lib/systemd/systemd
55dbcd143000-55dbcd17b000 r--p 0014d000 08:01 6296083
 /lib/systemd/systemd
55dbcd17b000-55dbcd17c000 rw-p 00185000 08:01 6296083
 /lib/systemd/systemd
55dbcde23000-55dbcdfb9000 rw-p 00000000 00:00 0                          [heap]
7f2cf8000000-7f2cf8021000 rw-p 00000000 00:00 0
7f2cf8021000-7f2cfc000000 ---p 00000000 00:00 0
7f2cfc124000-7f2cfc125000 ---p 00000000 00:00 0
7f2cfc125000-7f2cfc925000 rw-p 00000000 00:00 0
7f2cfc925000-7f2cfc926000 ---p 00000000 00:00 0
7f2cfc926000-7f2cfd126000 rw-p 00000000 00:00 0
7f2cfd126000-7f2cfd2c3000 r-xp 00000000 08:01 6296249                    /lib/x86_64-linux-
gnu/libm-2.27.so
7f2cfd2c3000-7f2cfd4c2000 ---p 0019d000 08:01 6296249                    /lib/x86_64-linux-
gnu/libm-2.27.so
7f2cfd4c2000-7f2cfd4c3000 r--p 0019c000 08:01 6296249                    /lib/x86_64-linux-
gnu/libm-2.27.so
7f2cfd4c3000-7f2cfd4c4000 rw-p 0019d000 08:01 6296249                    /lib/x86_64-linux-
gnu/libm-2.27.so
...
7f2d01bdc000-7f2d01bdd000 rw-p 00000000 00:00 0
7fffcf739000-7fffcf75a000 rw-p 00000000 00:00 0                          [stack]
7fffcf7f8000-7fffcf7fb000 r--p 00000000 00:00 0                          [vvar]
7fffcf7fb000-7fffcf7fd000 r-xp 00000000 00:00 0                          [vdso]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0                  [vsyscall]
```

maps中每个条目的格式如下：

```
vmstart-vmend attr pgoffset devname inode filename
```

vmstart和vmend代表虚拟内存的起始和结束地址；attr表示这块内存区域是否可读、可写、可执行和可共享。pgoffset是这块区域的页面偏移；devname显示为xx:xx，是与该内存区域关联的设备ID，如果没有与一个文件关联，那么也不会与设备关联，就会显示为00:00；inode和filename表示关联的文件inode号和文件名，如果没有关联文件则inode为0，filename为空。 除了/proc/n中相关的进程条目外，常用的/proc条目有/proc/cpuinfo、/proc/meminfo和/proc/version。/proc/cpuinfo包含了处理器的相关信息，/proc/meminfo提供内存统计信息，/proc/version包含内核版本号，编译内核使用的编译器版本和平台。另外/proc还有许多其他有用的条目，如/proc/mounts包含所有的文件系统的挂载信息，/proc/modules包含当前已加载的模块信息。

2、sysfs文件系统

sysfs文件系统也是一个伪文件系统，sysfs对具体的内核对象(比如设备和驱动)进行建模，并关联起来。sysfs的主要目的是展示设备驱动模型的各组件的层次关系，udev或者mdev就是根据sysfs中的信息创建设备节点来让用户访问对应的设备。 和proc文件系统一样，sysfs文件系统也通常挂载在一个指定目录/sys，可手工在shell挂载或者在/etc/fstab中设置使得开机自动挂载，挂载方法和proc一样：

```
# mount -t sysfs none /sys
```

/sys目录下各主要子目录的名称和功能如下：

（1）bus：内核中注册的每条总线在该目录下对应一个目录，包括虚拟总线platform。

（2）bus/某一总线/devices：该总线下所有设备都在该目录下对应一个目录。

（3）bus/某一总线/drivers：该总线下所有驱动都在该目录下对应一个目录。

（4）class：将设备按类分类。

（5）devices：包含所有设备的信息，并根据设备挂接总线类型组织成层次结构。

需要注意的是真正的设备信息只放在/sys/devices目录下，其他目录下都是连接到该目录下对应子目录的符号链接。

四、处理出错信息

1、oops信息描述

当系统或程序的一些bug出错时会打印出oops信息，当发生oops后，系统不稳定，可能已经崩溃，也可能继续运行。以下是一段出错后的oops信息。

```
Unable to handle kernel paging request at virtual address 55550000
pgd = bfb74000
[55550000] *pgd=00000000
Internal error: Oops: 5 [#1] PREEMPT SMP ARM
Modules linked in: hello(O)
CPU: 0    Tainted: G           O  (3.7.0 #1)
PC is at hello_read+0x10/0x20 [hello]
LR is at vfs_read+0x94/0xcc
pc : [<7f00002c>]    lr : [<800d74a8>]    psr: 20000013
sp : bfb1bf70  ip : 7f00001c  fp : 00000000
r10: 00000001  r9 : bfb1a000  r8 : 00000000
r7 : 000105c4  r6 : bfb1bf80  r5 : 000105c4  r4 : bf95fa80
r3 : 55550000  r2 : 00000001  r1 : 000105c4  r0 : 00000000
Flags: nzCv  IRQs on  FIQs on  Mode SVC_32  ISA ARM  Segment user
```

```
Control: 10c5387d  Table: 3fb7404a  DAC: 00000015
Process hello_test (pid: 592, stack limit = 0xbfb1a240)
Stack: (0xbfb1bf70 to 0xbfb1c000)
bf60:                                     00000000 00000000 bf95fa80 800d751c
bf80: 00000000 00000000 00000026 00000000 7e9c3ce8 00008325 00000003 8000e428
bfa0: 00000000 8000e280 00000000 7e9c3ce8 00000003 000105c4 00000001 000083fd
bfc0: 00000000 7e9c3ce8 00008325 00000003 00000000 00000000 76fd8000 00000000
bfe0: 00000000 7e9c3cc4 00008423 76f5c1cc 40000010 00000003 05158105 12585154
[<7f00002c>] (hello_read+0x10/0x20 [hello]) from [<800d74a8>] (vfs_read+0x94/0xcc)
[<800d74a8>] (vfs_read+0x94/0xcc) from [<800d751c>] (sys_read+0x3c/0x70)
[<800d751c>] (sys_read+0x3c/0x70) from [<8000e280>] (ret_fast_syscall+0x0/0x30)
Code: e3003328 e3a00000 e3473f00 e5933000 (e5932000)
---[ end trace c34dd3ea53423d33 ]---
Segmentation fault
```

这些信息主要分为以下部分：

1.1 一段文本描述信息

```
Unable to handle kernel paging request at virtual address 55550000
```

它说明了发生了哪类错误，上面指出访问虚拟地址55550000时发生错误。

1.2 出错所在的内核中加载的模块名称

```
Modules linked in: hello(O)
```

上面信息显示出错在hello.ko模块中，但注意如果不是出错在模块，这里会为空字符串。

1.3 发生错误的CPU的序号和内核版本号

```
CPU: 0    Tainted: G           O  (3.7.0 #1)
```

1.4 发生错误时的出错CPU的各个寄存器值

```
PC is at hello_read+0x10/0x20 [hello]
LR is at vfs_read+0x94/0xcc
pc : [<7f00002c>]    lr : [<800d74a8>]    psr: 20000013
sp : bfb1bf70  ip : 7f00001c  fp : 00000000
r10: 00000001  r9 : bfb1a000  r8 : 00000000
r7 : 000105c4  r6 : bfb1bf80  r5 : 000105c4  r4 : bf95fa80
r3 : 55550000  r2 : 00000001  r1 : 000105c4  r0 : 00000000
Flags: nzCv  IRQs on  FIQs on  Mode SVC_32  ISA ARM  Segment user
```

1.5 当前进程的名字和pid

```
Process hello_test (pid: 592, stack limit = 0xbfb1a240)
```

以上信息说明出错时的进程是hello_test，但不一定是该进程的用户空间代码出错，也可能是代该进程进行服务的内核代码出错，如驱动或系统调用或中断处理函数(但这一般是用户空间传入错误的参数)，pid为592。

1.6 栈信息

```
Stack: (0xbfb1bf70 to 0xbfb1c000)
bf60:                                     00000000 00000000 bf95fa80 800d751c
bf80: 00000000 00000000 00000026 00000000 7e9c3ce8 00008325 00000003 8000e428
bfa0: 00000000 8000e280 00000000 7e9c3ce8 00000003 000105c4 00000001 000083fd
bfc0: 00000000 7e9c3ce8 00008325 00000003 00000000 00000000 76fd8000 00000000
bfe0: 00000000 7e9c3cc4 00008423 76f5c1cc 40000010 00000003 05158105 12585154
```

发生错误时，将栈的内存从当前sp指向的栈底往上打印出来(栈往下增长)。

1.7 栈回溯

```
[<7f00002c>] (hello_read+0x10/0x20 [hello]) from [<800d74a8>] (vfs_read+0x94/0xcc)
[<800d74a8>] (vfs_read+0x94/0xcc) from [<800d751c>] (sys_read+0x3c/0x70)
[<800d751c>] (sys_read+0x3c/0x70) from [<8000e280>] (ret_fast_syscall+0x0/0x30)
```

栈回溯清晰地展示了出错的函数调用过程，从下往上，比如上面的信息调用关系为：ret_fast_syscall -> sys_read -> vfs_read -> hello_read。

1.8 出错指令

```
Code: e3003328 e3a00000 e3473f00 e5933000 (e5932000)
```

2、手工分析栈回溯

上面的oops信息很容易能找到出错的地方，比如通过1.2的Modules linked in: hello(O)和1.4的PC is at hello_read+0x10/0x20 [hello]就能马上知道在hello.ko模块的hello_read中出错，或者通过栈回溯信息也可以。但这些信息并不是什么时候都有的，通常需要内核配置，有的内核默认配置了有的需要自己配置，具体请上网查看。因此在没有这些函数调用信息时我们只能通过寄存器的值和栈信息手工推导回溯信息。

2.1 通过pc寄存器找到出错函数，确定它的栈大小

从上文的oops信息中可知pc值为7f00002c，从System.map中发现内核中的符号都是8开头，如下：

```
00000020 A cpu_v7_suspend_size
80004000 A swapper_pg_dir
80008000 T _text
80008000 T stext
80008088 t __create_page_tables
80008128 t __turn_mmu_on_loc
80008134 t __fixup_pv_table
80008174 t __vet_atags
800081c0 T __exception_text_start
800081c0 T _stext
800081c0 T asm_do_IRQ
800081c4 T do_undefinstr
8000838c T do_IPI
80008390 T do_DataAbort
8000842c T do_PrefetchAbort
800084c8 T gic_handle_irq
80008520 t __do_fixup_smp_on_up
```

```
80008520 T __exception_text_end
80008534 T fixup_smp
8000854c t __fixup_a_pv_table
80008570 T fixup_pv_table
80008594 t run_init_process
800085b8 T do_one_initcall
...
```

因此这不是内核编译时确定的函数中的地址，因此是外部加载的函数的地址，如加载的模块。所以先需要通过/proc/kallsyms查看接近该pc值的符号地址，先cat /proc/kallsyms > symbols.txt将所有的符号地址导出到symblos.txt，然后通过grep "7f00" symbols.txt寻找pc值7f00002c附近的符号，如下：

```
root@socfpga_cyclone5:/# grep "7f00" symbols.txt
80247f00 T sdev_evt_send_simple
803a7f00 r __func__.26733
80477f00 r __ksymtab_enable_hlt
8047f000 r __ksymtab_task_handoff_unregister
8047f008 r __ksymtab_tasklet_hrtimer_init
7f000000 t $a   [hello]
7f000000 t hello_open   [hello]
7f00001c t hello_read   [hello]
7f00003c t hello_write  [hello]
7f000044 t $a   [hello]
7f000044 t hello_exit   [hello]
7f00005c t $d   [hello]
7f000044 t cleanup_module       [hello]
root@socfpga_cyclone5:/#
```

通过以上信息基本确定出错地址在**模块hello.ko的函数hello_read**中。再通过反汇编hello.ko模块得到：

```
hello.ko:     file format elf32-littlearm


Disassembly of section .text:

00000000 <hello_open>:
   0:   e3003000    movw    r3, #0
   4:   e3a02000    mov r2, #0
   8:   e3403000    movt    r3, #0
   c:   e3452555    movt    r2, #21845  ; 0x5555
  10:   e5832000    str r2, [r3]
  14:   e3a00000    mov r0, #0
  18:   e12fff1e    bx  lr

0000001c <hello_read>:
  1c:   e3003000    movw r3, #0
  20:   e3a00000    mov r0, #0
  24:   e3403000    movt r3, #0
  28:   e5933000    ldr r3, [r3]
  2c:   e5932000    ldr r2, [r3]
  30:   e3822001    orr r2, r2, #1
```

```
  34:   e5832000    str r2, [r3]
  38:   e12fff1e    bx  lr

0000003c <hello_write>:
  3c:   e3a00000    mov r0, #0
  40:   e12fff1e    bx  lr
...
```

通过反汇编信息知道出处地址为hello_read中的2c: e5932000 ldr r2, [r3]，该函数没有使用栈，因此栈大小为0。

2.2 通过lr值找到调用函数，确定栈大小

lr的值为800d74a8，这个值明显属于内核函数中，因此可以通过反汇编vmlinux查看哪个符号地址接近800d74a8。首先在System.map中看到：

```
。。。
800d7348 T vfs_write
800d7414 T vfs_read
800d74e0 T sys_read
800d7550 T sys_write
。。。
```

因此hello_read的调用函数为vfs_read，将vmlinux反汇编搜索vfs_read得到如下代码：

```
800d7414 <vfs_read>:
800d7414:       e92d4070        push    {r4, r5, r6, lr}
...
800d74a4:       e12fff3c        blx     ip
800d74a8:       e1a02000        mov     r2, r0
...
```

说明800d74a4: e12fff3c blx ip这一行就是在vfs_read中调用hello_read的指令，从该处到vfs_read开始对栈的操作只有第一条指令，因此栈信息中vfs_read的部分为00000000 00000000 bf95fa80 800d751c，按寄存器的序号从低往高排放，因此800d751c就是vfs_read保存的lr，即调用vfs_read的函数中的调用vfs_read处的下一条指令地址。

```
800d74e0 <sys_read>:
800d74e0:       e92d45f0        push    {r4, r5, r6, r7, r8, sl, lr}
800d74e4:       e24dd00c        sub     sp, sp, #12
...
800d7518:       ebffffbd        bl      800d7414 <vfs_read>
800d751c:       e1cd20d0        ldrd    r2, [sp]
800d7520:       e3580000        cmp     r8, #0
...
```

因此栈信息中vfs_read的部分为00000000 00000000 00000026 00000000 7e9c3ce8 00008325 00000003 8000e428 00000000 8000e280，因此8000e280就是sys_read保存的lr，即调用sys_read的函数中的调用sys_read处的下一条指令地址。

```
8000e280 <ret_fast_syscall>:
8000e280:       f10c0080        cpsid   i
8000e284:       e5991000        ldr     r1, [r9]
```

```
8000e288:       e3110007        tst     r1, #7
8000e28c:       1a000007        bne     8000e2b0 <fast_work_pending>
8000e290:       e59d1048        ldr     r1, [sp, #72]   ; 0x48
8000e294:       e5bde044        ldr     lr, [sp, #68]!  ; 0x44
8000e298:       e16ff001        msr     SPSR_fsxc, r1
8000e29c:       f57ff01f        clrex
8000e2a0:       e95d7ffe        ldmdb   sp, {r1, r2, r3, r4, r5, r6, r7, r8, r9, sl, fp,
ip, sp, lr}^
8000e2a4:       e1a00000        nop                     ; (mov r0, r0)
8000e2a8:       e28dd00c        add     sp, sp, #12
8000e2ac:       e1b0f00e        movs    pc, lr
```

所以调用路径为ret_fast_syscall -> sys_read -> vfs_read -> hello_read，和自动打印出的回溯信息一致。

五、利用KGDB调试源码

KGDB调试需要一台主机和一个目标开发板，主机和开发板之间通过一条串口线链接。内核源码在主机上交叉编译并且通过交叉GDB调试，内核镜像下载到开发板运行，两者通过串口通信。调试步骤如下：

1、使用KGDB的内核配置

```
Kernel hacking --->
    ....
    [*] Compile the kernel with debug info
    ...
    [*] KGDB: kernel debugger --->
        <*> KGDB: use kgdb over the serial console
    ...
```

2、在开发板上启动内核

在启动开发板前，先在uboot中设置bootargs环境变量，添加如下启动参数:

```
kgdboc=ttyS0,57600 kgdbwait
```

kgdboc表示使用串口连接，kgdbwait表示内核的串口驱动加载成功后，将会等待主机的gdb连接。

未调通，后续再做

六、sysrq系统请求键

1、配置与使能sysrq

要使用sysrq系统请求键，首先在编译内核时候配置**CONFIG_MAGIC_SYSRQ=y**，然后它就被编译进了内核。 通过**/proc/sys/kernel/sysrq**里面的值控制使能或者关闭sysrq功能，0表示关闭，1表示打开，通过echo往该文件写0或者1来关闭或者打开。 在开发板上使用sysrq有限制，可以通过**echo x > /proc/sysrq-trigger**来使用，其中x表示其中一个功能。

2、常用sysrq功能

- b 重启。
- o 关闭机器。
- e 给init之外的所有进程发送SIGTERM信号。
- i 给init之外的所有进程发送SIGKILL信号。
- l 给包括init之内的所有进程发送SIGKILL信号。
- m 在控制台上显示内存信息。
- p 在控制台上显示寄存器。
- t 在控制台上dump出一系列任务信息。

七、ftrace追踪

ftrace主要使用的是静态追踪技术，但也有动态追踪功能。

1、ftrace使用前的设置

在使用ftrace之前先确定是否挂载了**debugfs文件系统**和配置了相应的**内核配置选项**。

1.1 挂载debugfs文件系统

通过mount命令可以看到是否挂载了debugfs文件系统，在socfpga平台如下所示：

```
root@socfpga_cyclone5:~# mount
rootfs on / type rootfs (rw)
/dev/root on / type ext3 (rw,relatime,errors=continue,user_xattr,barrier=1,data=ordered)
devtmpfs on /dev type devtmpfs (rw,relatime,size=500652k,nr_inodes=125163,mode=755)
proc on /proc type proc (rw,relatime)
sysfs on /sys type sysfs (rw,relatime)
debugfs on /sys/kernel/debug type debugfs (rw,relatime)
tmpfs on /var/volatile type tmpfs (rw,relatime)
tmpfs on /media/ram type tmpfs (rw,relatime)
devpts on /dev/pts type devpts (rw,relatime,gid=5,mode=620)
```

可以看到debugfs文件系统被挂载到了/sys/kernel/debug。如果没有挂载则可以修改**/etc/fstab**自动挂载或者输入如下命令来手工挂载。

```
mount -t debugfs none /sys/kernel/debug
```

1.2 内核配置选项

> Kernel hacking ---> Tracers

```
/* 这些是相对通用的设置，socfpga平台中已默认选中 */
CONFIG_FTRACE=y
CONFIG_HAVE_FUNCTION_TRACER=y
CONFIG_HAVE_FUNCTION_GRAPH_TRACER=y
CONFIG_HAVE_DYNAMIC_FTRACE=y
CONFIG_ENABLE_DEFAULT_TRACERS=y

/* 这是是各种追踪器或功能的设置，socfpga平台没有默认选中，因此要menuconfig配置 */
CONFIG_FUNCTION_TRACER=y
CONFIG_IRQSOFF_TRACER=y
CONFIG_SCHED_TRACER=y
CONFIG_PREEMPT_TRACER=y
CONFIG_FTRACE_SYSCALLS=y
```

1.3 操作展示

在挂载debugfs文件系统和配置编译了ftrace相应功能后会在debugfs下创建一个 tracing目录 **/sys/kernel/debug/tracing**，该目录下包含了 ftrace 的控制和输出文件，目录下文件和目录比较多，有些是各种跟踪器共享使用的，有些是特定于某个跟踪器使用的。在操作这些数据文件时，通常使用 echo 和cat命令来修改和查看其值，也可以在程序中通过文件读写相关的函数来操作这些文件的值。下面只对部分文件进行描述，读者可以参考内核源码包中 Documentation/trace 目录下的文档以及 kernel/trace 下的源文件以了解其余文件的用途。 更详细的解析参考内核调试工具集合**一文。下面直接展示一个ftrace中的一个子功能**函数追踪器**的简单操作。

```
#!/bin/bash

debugfs=/sys/kernel/debug
echo nop > $debugfs/tracing/current_tracer      // 清空追踪器类型,同时会清空之前trace的残余信息
echo 0 > $debugfs/tracing/tracing_on            // 配置完成前先关闭tracer跟踪
echo $1 > $debugfs/tracing/set_ftrace_pid       // 让函数跟踪器仅跟踪单个线程，运行脚本时输入PID
echo function_graph > $debugfs/tracing/current_tracer   // 设置当前current_tracer追踪器为
function_graph追踪器
echo 5 > $debugfs/tracing/max_graph_depth               // 设置追踪深度
echo functionname > $debugfs/tracing/           // 可以设置指定的函数名
echo 1 > $debugfs/tracing/tracing_on            // 开启跟踪
....    //运行一段时间
echo 0 > $debugfs/tracing/tracing_on            // 关闭跟踪
cat $debugfs/tracing/trace                      //查看跟踪信息
```

八、动态追踪

1、一些概念

- **静态追踪和动态追踪**

  静态追踪利用的是固定的静态探测点tracepoints的代码和函数入口处的静态插桩代码实现跟踪。需要重新编译内核。 **动态追踪**利用的是在动态探测点（几乎任何位置）实现陷入异常，然后在异常中收集信息然后返回异常前继续运行。不需要重新编译内核，利用kprobe技术。

- **tracepoint**

参考奔跑吧linux内核6.2.6和6.2.7。

- **kprobe**

  参考Linux内核调试技术——kprobe使用与实现。 利用kprobes技术，用户可以定义自己的回调函数，然后在内核或者模块中几乎所有的函数中（有些函数是不可探测的，例如kprobes自身的相关实现函数，后文会有详细说明）动态的插入探测点，当内核执行流程执行到指定的探测函数时，会调用该回调函数，用户即可收集所需的信息了，同时内核最后还会回到原本的正常执行流程。如果用户已经收集足够的信息，不再需要继续探测，则同样可以动态的移除探测点。 kprobes技术包括的3种探测手段分别时kprobe、jprobe和kretprobe。首先kprobe是最基本的探测方式，是实现后两种的基础，它可以在任意的位置放置探测点（就连函数内部的某条指令处也可以），它提供了探测点的**调用前、调用后和内存访问**出错3种回调方式，分别是**pre_handler、post_handler和fault_handler**，其中pre_handler函数将在被探测指令被执行前回调，post_handler会在被探测指令执行完毕后回调（注意不是被探测函数），fault_handler会在内存访问出错时被调用；jprobe基于kprobe实现，它用于获取被探测函数的入参值；最后kretprobe从名字种就可以看出其用途了，它同样基于kprobe实现，用于获取被探测函数的返回值。

- **uprobe**

  类似kprobe，但是追踪的是用户空间的进程。

2、systemtap

systemtap利用kprobe的api来动态监测和跟踪内核。它使用了自定义的脚本语言，一个systemtap脚本描述了探测点和定义了相关联的处理函数。systemtap脚本解释器将脚本解释转换成c代码，最后编译链接成一个可加载的内核模块。模块加载时调用kprobe的注册api注册探测点，内核运行到探测点会回调用户自定义函数，然后输出。

未完待续

九、perf

perf没有systemtap的自定义编程能力，但是能观测的东西很多，可以支持静态追踪，动态追踪和profiling。

未完待续

十、qemu

1、下载内核源码和busybox源码

```
https://www.kernel.org/pub/linux/kernel/v4.x/linux-4.0.tar.gz
https://busybox.net/downloads/busybox-1.24.0.tar.bz2
```

2、安装工具

```
sudo apt-get install qemu libncurses5-dev gcc-arm-linux-gnueabi build-essential
```

3、编译制作根文件系统

3.1 initramfs方式

参考奔跑吧linux内核6.1.1。还有把交叉编译工具中的lib拷贝到根文件系统。

3.2 普通方式

先制作和格式化一个镜像：

```
dd if=/dev/zero of=rootfs.ext4 bs=1M count=64
mkfs.ext4 rootfs.ext4
```

然后拷贝内容进去：

```
/* 先挂载到主机的tmpfs目录 */
mkdir tmpfs
mount -t ext4 rootfs.ext4 tmpfs/ -o loop
/* 然后将制作好的根文件系统拷贝到挂载点  */
cp _install/* tmpfs/ -rf
/* 卸载 */
umount tmpfs
```

最后启动qemu时加参数 -sd rootfs.ext4，同时内核编译将initramfs方式去掉。

4、编译内核

```
export ARCH=arm
export CROSS_COMPILE=arm-linux-gnueabi-
make distclean
make vexpress_defconfig
make menuconfig     //修改initramfs相关选项，参考奔跑吧linux内核6.1.1
make zImage -j8
make modules -j8
make dtbs -j8
```

5、启动qemu模拟开发板

使用initramfs：

```
qemu-system-arm -M vexpress-a9 -smp 4 -m 512M -dtb arch/arm/boot/dts/vexpress-v2p-ca9.dtb -
kernel arch/arm/boot/zImage -nographic -append "rdinit=/linuxrc console=ttyAMA0"
```

不用initramfs：

```
qemu-system-arm -M vexpress-a9 -smp 4 -m 512M -dtb arch/arm/boot/dts/vexpress-v2p-ca9.dtb -
kernel arch/arm/boot/zImage -nographic -append "root=/dev/mmcblk0 rw console=ttyAMA0" -sd
rootfs.ext4
```

gdb调试：

```
qemu-system-arm -M vexpress-a9 -smp 4 -m 512M -dtb arch/arm/boot/dts/vexpress-v2p-ca9.dtb -
kernel arch/arm/boot/zImage -nographic -append "rdinit=/linuxrc console=ttyAMA0" -S -s
```

参考奔跑吧linux内核6.1.2。

6、退出

在另一个终端输入：

```
killall qemu-system-arm
```