

SIXUENET

To Learn Without Thinking Is To Be Useless, To Think Without Learning Is To Lose
(<http://www.mysixue.com/>)

DEBUGGING TECHNOLOGY

📅 April 8, 2019 ([Http://Www.Mysixue.Com/?P=94](http://Www.Mysixue.Com/?P=94)) 👤 JL
([Http://Www.Mysixue.Com/?Author=1](http://Www.Mysixue.Com/?Author=1)) 🔔

1 Introduction to the document structure

This article will introduce the first printing type debugging technology , which is similar printf to print information the way , and then view the information , in the process of commissioning the drive in , we often use this way .

Then we will introduce the query debugging method , which is generally to query the kernel information in the user space .

Finally, we will introduce several ways to tune a system failure : gdb how to use , oops how wrong and debugging .

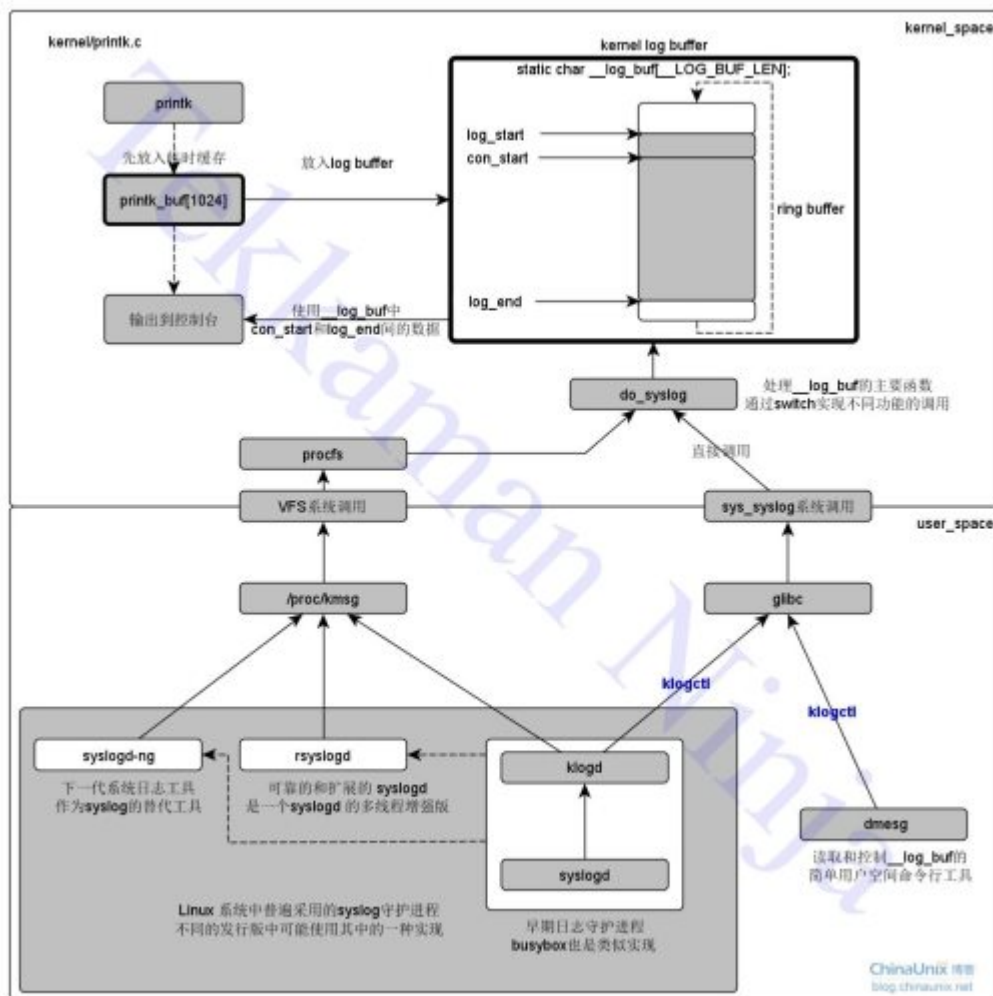
The last chapter also has a drive design guideline that summarizes some precautions .

2 Printed debugging

2.1 printk

In the actual debugging process , the main problem is some information printed by printk , we can't see it (such as log level problems), the main purpose of this section is to let us know how to view this information .

Next, it will be introduced according to the data flow of printk . Let's first look at a picture :



Data flow : the printk -> recording string to log_buf -> Get console port semaphore -> call console port underlying write function to output content to the actual console port -> wakeup klogd process .

The klogd process will notify the syslogd process, and the syslogd process will log the information to /var/log/messages .

You can also get information from log_buf through the dmesg command .

This article does not analyze the specific implementation of printk , but explains printk from the perspective of use . If you want to understand the implementation details of printk , you can refer to this [article](http://blog.chinaunix.net/uid-20543672-id-3217096.html) (http://blog.chinaunix.net/uid-20543672-id-3217096.html) or this [article on the](http://www.cnblogs.com/zhouhaibing/archive/2013/01/21/2870506.html) (http://www.cnblogs.com/zhouhaibing/archive/2013/01/21/2870506.html) Internet . (http://blog.chinaunix.net/uid-20543672-id-3217096.html) (http://www.cnblogs.com/zhouhaibing/archive/2013/01/21/2870506.html)

Data generation : printk

Header file : include/linux/printk.h

Implementation file : kernel/printk/printk.c

In the kernel code , we often see the following print statements :

```
printk(KERN_DEBUG " debug message\n");
```

This is printk syntax format , it is the C language printf very similar , the only difference is that there is a KERN_DEBUG , this label is loglevel.

loglevel & CONFIG_MESSAGE_LOGLEVEL_DEFAULT

loglevel: print level , its function will be introduced later , here we mainly talk about the principle of loglevel .
loglevel is defined in the kernel header file include/linux/ kern_levels.h , printk.h will automatically include this header file . Kern_levels.h defines [0-7] these 8 printing levels:

```
#define KERN_SOH "\001"      /* ASCII Start Of Header */
#define KERN_SOH_ASCII '\001'

#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 */

#define KERN_DEFAULT KERN_SOH "d"  /* the default kernel loglevel */
```

Each print a character string using the macro level is defined , the first string is '\ 001' of this special character , back to keep a represents an integer of priority , **the smaller the value , the higher the priority** .

During the compilation process , the above macros will be expanded , and the compiler will automatically merge these strings together , similar to the following :

```
printk( "\001 7 debug message\n " );
```

This is why the form printk (KERN_DEBUG "debug message \ n") This statement , **KERN_DEBUG behind the direct space and " Debug the Message \ the n-" reason for the split** .

Of course , you can also write printk directly like this :

```
printk("debug message\n");
```

In this case , the article printk default print level is DEFAULT_MESSAGE_LOGLEVEL (if you printk.c search in the macro , you can understand the principle , here not elaborate).

Let's see how this macro is defined :

```
/* printk's without a loglevel use this.. */
#define MESSAGE_LOGLEVEL_DEFAULT CONFIG_MESSAGE_LOGLEVEL_DEFAULT

int console_printk [ 4 ] = {
    CONSOLE_LOGLEVEL_DEFAULT , /* console_loglevel */
    MESSAGE_LOGLEVEL_DEFAULT , /* default_message_loglevel */
    CONSOLE_LOGLEVEL_MIN , /* minimum_console_loglevel */
    CONSOLE_LOGLEVEL_DEFAULT , /* default_console_loglevel */
};

#define default_message_loglevel (console_printk[1])
```

From the above code , what do you know ? Yes , the default printing level is defined in menuconfig : **CONFIG_MESSAGE_LOGLEVEL_DEFAULT**

Enable/Disable TIME

Sometimes , we want to print information on each front plus a time information , to help us determine two messages how long interval , this time , we can menuconfig open in CONFIG_PRINTK_TIME switch . In this way , you'll get information like this :

```
[0.000198] Console: colour dummy device 80×30
```

```
[0.000230] Calibrating delay loop... 996.14 BogoMIPS (lpj=4980736)
```

If you want to know the details of this mechanism , in printk.c search for CONFIG_PRINTK_TIME , you know the principle .

By the way , in addition to enabling CONFIG_PRINTK_TIME through menuconfig , you can also modify it with echo/sys/module/printk/parameters/time . It is a module_param defined in printk.c

Control Speed

Sometimes , we may need to call printk in a loop . If the loop cannot be jumped out of the loop under certain circumstances , and it runs continuously there , thousands of printk may be generated at this time .

Such a large number of printk very bad , because printk messages printed , may eventually sent to the UART, but UART speed is slow , a lot of printk can cause the system stuck .

The kernel provides an API to solve this problem : int printk_ratelimit(void), its usage is generally as follows :

```
if (printk_ratelimit())
```

```
    printk(KERN_NOTICE " The is an error\n " );
```

printk_ratelimit can track the number of messages sent to the console . If the output speed exceeds a threshold , printk_ratelimit will return zero .

We can pass /proc/sys/kernel/printk_ratelimit (the number of seconds you should wait before reopening the message) and /proc/sys/kernel/printk_ratelimit_burst (the number of messages that can be received before rate limiting) to customize the behavior of printk_ratelimit .

NOTE: The two parameters in the above /proc are created in kernel/sysctl.c .

Data storage : log_buf

When you call printk when printing a message , the message will first be copied to a local variable textbuf [LOG_LINE_MAX] in (printk.c as -> vprintk_emit defined) , LOG_LINE_MAX size is (1024--32) bytes .

```
#define PREFIX_MAX 32
#define LOG_LINE_MAX (1024 - PREFIX_MAX)
```

Then , will textbuf data copied to log_buf in (printk.c as defined in a ring buffer).

```
#define __LOG_BUF_LEN (1 << CONFIG_LOG_BUF_SHIFT)
static char __log_buf [ __LOG_BUF_LEN ] __aligned ( LOG_ALIGN );
static char * log_buf = __log_buf ;
static u32 log_buf_len = __LOG_BUF_LEN ;
```

From the above code , log_buf size may menuconfig by the CONFIG_LOG_BUF_SHIFT configured , default is 64KB. When log_buf data is greater than 64KB when , will overwrite the previous data .

OK, the only purpose of this section is to let you know **how to configure the size of log_buf** .

Note that **no matter what the loglevel of printk is , it will be written to log_buf** .

Data display 1: console

When the message is stored in `log_buf`, type will be displayed to the console on .

What is a console ?

To understand the console, you first need to know a concept : the terminal .

Also known as terminal `tty`, this section does not intend to discuss the details of the terminal , there will be a special article to tell `tty` subsystem .

Variety of types of terminals , what is depending on the vehicle : for example, as a serial port support transmitting and receiving data , is serial terminal ; used as a carrier console send and receive data , that is, the console terminal

The amount , the console ? What is it ? Be careful not to put the console with the console confused , although the console translated also called console , but you remember Talia are two concepts it . Console can generally be understood as Linux 's display subsystem + input subsystem . console terminal means is displayed on the display subsystem (LCD, HDMI) display data , and from the input subsystem (keyboard , mouse) acquiring the terminal data .

OK, end over roughly interpretation , then the console it ? Terminal There are many types , any type of terminal can be registered as a console. Console registered API is `register_console` (`printk.c` defined), all registered console will Mount it under the global linked list of `console_drivers` .

It `printk` data which will be displayed in the console on it ? Look at the following code :

```
for_each_console {  
    ...  
    if (!( con -> flags & CON_ENABLED ))  
        continue ;  
    if (! con -> write )  
        continue ;  
    con -> write ( con , text , len );  
    ...  
}
```

The meaning is clear , for `console_drivers` each under the console, check `CON_ENABLED` if set , and check to achieve a write function . If the conditions are met , will be called `con-> write`. Assume that this `con` is a serial terminal , the write function A string of data will be written to the serial port . At this point , you can see the message printed by `printk` on the serial port .

The write function is generally implemented , the key question is Whether the `CON_ENABLED` bit is set . You may be thinking , can this be turned on when registering the console ? But in fact , when the console is registered , the `CON_ENABLED` bit is in the DISABLE state by default . Then who will enable these consoles? What ? Please see the next section .

Data is displayed in which console on

Remember how the bootargs parameters passed by uboot are written , generally as follows :

`b ootargs = "console=ttyS0,115200 console=tty1 ignore_loglevel earlyprintk";`

Kernel in the startup phase , will resolve bootargs, for " Console = " string , parse it functions in printk.c defined console_setup .

```
__setup ( "console=" , console_setup );
```

The console_setup function will parse the name, index, and options of the console ; and use a structure struct console_cmdline to represent this console.

```
struct console_cmdline
{
    char      name [ 16 ]; /* Name of the driver */
    int index ; /* Minor dev. to use */
    char * options ; /* Options for the driver */
#ifdef CONFIG_Ally_BRAILLE_CONSOLE
    char * brl_options ; /* Options for braille driver */
#endif
};
```

bootargs each " Console = " string will result console_setup is called a secondary , each call will generate a console_cmdline structure .

By convention the kernel , there must be a global something to store these console_cmdline ' S. Yes , printk.c defines a global array of structures :

```
#define MAX_CMDLINECONSOLES 8

static struct console_cmdline console_cmdline [ MAX_CMDLINECONSOLES ];
```

This array of structures to store up to 8 elements , which means , bootargs up to only 8 Ge " Console = " .

When bootargs is parsed ,All these consoles specified by " console=" are stored in **console_cmdline** .

The parsing time of bootargs is very early , almost when the kernel has just started, and then the driver code for registering the console will be executed .

当这些代码调用API register_console注册一个console时, register_console函数会检查这个console->name & console->index在**console_cmdline**中是否存在, 如果存在, 则证明这个新注册的console被选中用于显示printk的打印消息了, 接而会执行newcon->flags |= CON_ENABLED

另外一个问题: 如果bootargs里面没有指定"**console=**", 那么printk的信息会显示到哪里呢?

The answer is also very simple , the flags of the **first registered console** will be set to CON_ENABLED .

OK, now you should understand printk print information which will be displayed in the console on the right .

printk all printed information will be displayed out of it ? Obviously not , or else to log_level doing , then we talk about what data is displayed in the console on .

What loglevel will appear in the console on

In the log_buf display data to the console before , to check loglevel. Code is as follows :

```
static void call_console_drivers ( int level , const char * text , size_t len )
{
    struct console * con ;
    ...
}
```

```

    if ( level >= console_loglevel && ! ignore_loglevel )
        return;
    if (! console_drivers )
        return ;

    for_each_console ( con ) {
        ...
        if (!( con -> flags & CON_ENABLED ))
            continue ;
        if (! con -> write )
            continue ;
        ...
        con -> write ( con , text , len );
    }
}

```

Note the red one piece if statement , if the conditions are not met , then the direct return, will not write to the console on .

There are two judgments involved in this if , one is to compare loglevel, and the other is to judge the bool variable ignore_loglevel . Let's look at the latter first .

ignore_loglevel

You can guess the meaning from the name : if ignore_loglevel = to true, then no matter loglevel what is , will unconditionally write to the console on .

ignore_loglevel is false by default , there are two ways to modify it :

One is through /sys/module/printk/parameters/ignore_loglevel

The other is through bootargs. If bootargs contains the string " ignore_loglevel " , then ignore_loglevel is true.

bootargs = "console=ttyS0,115200 console=tty1 ignore_loglevel earlyprintk";

" ignore_loglevel " is parsed by the ignore_loglevel_setup function in printk.c :

```

static int __init ignore_loglevel_setup ( char * str )
{
    ignore_loglevel = true ;
    pr_info ( "debug: ignoring loglevel setting.\n" );

    return 0 ;
}

early_param ( "ignore_loglevel" , ignore_loglevel_setup );
module_param ( ignore_loglevel , bool , S_IRUGO | S_IWUSR );
MODULE_PARM_DESC ( ignore_loglevel , "Setting the ignore the LogLevel, to"
    "Print all kernel messages to the console." );

```

Compare loglevel

If ignore_loglevel = to false, comparisons should LogLevel, when Level < console_loglevel time , will be write to the console on .

level that you use printk time , printing level specified in [0-7].

So console_loglevel is how much ?

First look at the definitions in printk.c and printk.h by default :

```
#define CONSOLE_LOGLEVEL_DEFAULT 7 /* anything MORE serious than KERN_DEBUG */

int console_printk [ 4 ] = {
    CONSOLE_LOGLEVEL_DEFAULT , /* console_loglevel */
    MESSAGE_LOGLEVEL_DEFAULT , /* default_message_loglevel */
    CONSOLE_LOGLEVEL_MIN , /* minimum_console_loglevel */
    CONSOLE_LOGLEVEL_DEFAULT , /* default_console_loglevel */
};

#define console_loglevel (console_printk[0])
```

The code indicates that console_loglevel default is 7 , which means printing level [0--6] can be displayed on the console on .

The above is the default , so how to modify the console_loglevel ? It can be modified through /proc, through bootargs, or through the dmesg command in user space .

Let me talk about how to modify it through /proc :

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

The first number is the corresponding console_loglevel, by echo. 5> / proc / SYS / Kernel / the printk modified console_loglevel .

NOTE: the above / proc in the printk parameters in kernel / sysctl.c created .

Let's talk about how to modify through bootargs :

b ootargs = "console=ttyS0,115200 console=tty1 ignore_loglevel quiet /debugloglevel =xx";

If bootargs defined in this quiet or debug; loglevel these early_param , then , may be modified console_loglevel.

These early_param of analytic functions in init / main.c in :

```
static int __init debug_kernel ( char * str )
{
    console_loglevel = CONSOLE_LOGLEVEL_DEBUG ; //CONSOLE_LOGLEVEL_DEBUG = 10
    return 0 ;
}

static int __init quiet_kernel ( char * str )
{
    console_loglevel = CONSOLE_LOGLEVEL_QUIET ; //CONSOLE_LOGLEVEL_QUIET = 4
    return 0 ;
}

early_param ( " debug " , debug_kernel );
early_param ( " quiet " , quiet_kernel );

static int __init loglevel ( char * str )
```



```

{
    int newlevel ;

    /*
     * Only update loglevel value when a correct setting was passed,
     * to prevent blind crashes (when loglevel being set to 0) that
     * are quite hard to debug
     */
    if ( get_option (& str , & newlevel )) {
        console_loglevel = newlevel ;
        return 0 ;
    }

    return - EINVAL ;
}

early_param ( " loglevel " , loglevel );

```

The code is already very clear , so I won't say more .

Talking about how to modify through the dmesg command :

`dmesg -n xx`

You can modify the value of `console_loglevel` .

Data display 2: User space

Looking back at the picture at the beginning of this chapter , in user space :

syslogd process can read `log_buf` data , and these data are stored in `/var/log/messages` in . We know that no matter what level of messages printed , will be stored in `log_buf` in , that syslogd when reading the data , will do loglevel filter it ? should also , without careful study . syslogd much used in the commissioning of the drive , it does not elaborate .

In addition to the syslogd, can also be used dmesg from `log_buf` read data , and the information is displayed in stdout on .

In use dmesg time , it will also focus on one issue , whether it would `log_buf` data do loglevel filtering ?

Experimented , if you use the dmesg command directly , it will print all the information in `log_buf` , and then if you use `dmesg -l warn`, only messages above the warn level will be printed .

The implementation of dmesg on each machine is different , use `dmesg -h` to see which functions your dmesg supports .

DEMO

This is also DEMO, yes , get hold DEMO it .

DEMO main purpose is to let you know with `printk` print out the news , where you can view . Because when we debugging kernel driver , most want to see is the driver inside `printk` those messages printed out , so that we can debug .

To design such a simple DEMO, use a sentence of `printk (DEBUG "XXX\n ")` in the loading and unloading functions ;

In the load module before , thinking about a problem ? When the load module then , you can in the console to see the information printed on it ? Use `dmesg` to see print information it ? If not , how to see these print information .

https://gitlab.com/study-kernel/debugging_techniques/printk
kernel/debugging_techniques/printk)

(https://gitlab.com/study-kernel/debugging_techniques/printk)

2.2 Variations of printk

pr_xxx

If you think that every time `printk` time , have a knock `loglevel` very much trouble , kernel code also provides some macros , to help you save .

Header file : `include/linux/printk.h`

```
/*
 * These can be used to print at the various log levels.
 * All of these will print unconditionally, although note that pr_debug()
 * and other debug macros are compiled out unless either DEBUG is defined
 * or CONFIG_DYNAMIC_DEBUG is set.
 */
#define pr_ emerg (fmt, ...) \
    printk(KERN_EMERG pr_fmt(fmt), ##__VA_ARGS__)
#define pr_ alert (fmt, ...) \
    printk(KERN_ALERT pr_fmt(fmt), ##__VA_ARGS__)
#define pr_ crit (fmt, ...) \
    printk(KERN_CRIT pr_fmt(fmt), ##__VA_ARGS__)
#define pr_ err (fmt, ...) \
    printk(KERN_ERR pr_fmt(fmt), ##__VA_ARGS__)
#define pr_ warning (fmt, ...) \
    printk(KERN_WARNING pr_fmt(fmt), ##__VA_ARGS__)
#define pr_ warn pr_warning
#define pr_ notice (fmt, ...) \
    printk(KERN_NOTICE pr_fmt(fmt), ##__VA_ARGS__)
#define pr_ info (fmt, ...) \
    printk(KERN_INFO pr_fmt(fmt), ##__VA_ARGS__)
#define pr_ cont (fmt, ...) \
    printk(KERN_CONT fmt, ##__VA_ARGS__)

/* pr_devel() should produce zero code unless DEBUG is defined */
#ifndef DEBUG
#define pr_ devel (fmt, ...) \
    printk(KERN_DEBUG pr_fmt(fmt), ##__VA_ARGS__)
#else
#define pr_devel(fmt, ...) \
```

```

    no_printk(KERN_DEBUG pr_fmt(fmt), ##__VA_ARGS__)
#endif

#include <linux/dynamic_debug.h>

/* If you are writing a driver, please use dev_dbg instead */
#if defined(CONFIG_DYNAMIC_DEBUG)
/* dynamic_pr_debug() uses pr_fmt() internally so we don't need it here */
#define pr_ debug (fmt, ...) \
    dynamic_pr_debug(fmt, ##__VA_ARGS__)
#elif defined(DEBUG)
#define pr_debug(fmt, ...) \
    printk(KERN_DEBUG pr_fmt(fmt), ##__VA_ARGS__)
#else
#define pr_debug(fmt, ...) \
    no_printk(KERN_DEBUG pr_fmt(fmt), ##__VA_ARGS__)
#endif

```

Note : If you need to **print out the** message with `pr_devel` or `pr_debug` , you need to define `DEBUG` in your C file .
`#define DEBUG`

dev_xxx

In the driver code , we should use `dev_xxx` to print messages , their advantage is that they will automatically display device->name and driver->name for you , which is convenient for you to debug .

Header file : `include/linux/device.h`

Implementation file : `drivers/base/core.c`

Although `dev_xxx` is not a direct call to `printk`, you can simply understand that they are an encapsulation of `printk` .

```

define_dev_printk_level ( dev_ emerg , KERN_EMERG );
define_dev_printk_level ( dev_ alert , KERN_ALERT );
define_dev_printk_level ( dev_ crit , KERN_CRIT );
define_dev_printk_level ( dev_ err , KERN_ERR );
define_dev_printk_level ( dev_ warn , KERN_WARNING );
define_dev_printk_level ( dev_ notice , KERN_NOTICE );
define_dev_printk_level(_dev_info, KERN_INFO);
#define dev_ info (dev, fmt, arg...) _dev_info(dev, fmt, ##arg)

```

There are also the following macros , but when using these macros , you should define `DEBUG` in your C file .

Or , you can also turn on the `CONFIG_DYNAMIC_DEBUG` switch in `menuconfig` , so that not only a certain C file , but all the places where the following macros are used can print out messages .

```

#if defined(CONFIG_DYNAMIC_DEBUG)
#define dev_dbg (dev, format, ...) \
do {\
    dynamic_dev_dbg(dev, format, ##__VA_ARGS__); \
} while (0)
#elif defined(DEBUG)

```

```

#define dev_dbg(dev, format, arg...) \
    dev_printk(KERN_DEBUG, dev, format, ##arg)
#else
#define dev_dbg(dev, format, arg...) \
({ \
    if (0) \
        dev_printk(KERN_DEBUG, dev, format, ##arg); \
    0; \
})
#endif

```

DEMO

This section is not necessary to do DEMO a , printing messages described in this section of the macro , one way only data are generated , similar to the printk, they regard the message on log_buf the ring buffer .

As for whether the information in log_buf can be seen in the console or dmesg , follow the rules described in the previous section .

If you want to experiment , you can test on the basis of the DEMO in the previous section .

2.3 early_printk

First of all , early_printk is not a simple encapsulation of printk , it exists to solve a certain type of problem .

In order to solve the problem ? If you can guess , it proves that you have a thorough understanding of the previous chapter .

In the " printk " chapter we know , a board , suppose you put it to the serial port of the PC on , the kernel boot process , want to see the message kernel serial printing .

In this case , you need to register the serial port as a console in the kernel code , and use a flag like " console=ttyS0 " in the bootargs .

When the **kernel code successfully registered serial console after** , you will be able to see the information printed on the serial port .

However , during the entire boot process of the kernel , the console registration is relatively late .

If the kernel hangs before the console is registered , you will not be able to see any printed information at this time .

To solve this problem , the kernel provides an early_printk mechanism .

To use this mechanism , you need to do the following things :

- Open CONFIG_EARLY_PRINTK in menuconfig
- In bootargs passing a parameter : bootargs = "... . **Earlyprintk** ";
- assembly language functions : addruart , waituart , senduart , busyuart these functions , using the UART print information

The whole mechanism is like this :

Kernel parameter analysis stage , Arch / ARM / Kernel / early_printk.c which implements a code , for parsing earlyprintk

```

extern void printch ( int );

static void early_write ( const char * s , unsigned n )
{
    while ( n - > 0 ) {
        if ( * s == '\n' )
            printch ( '\r' );
        printch ( * s );
        s ++;
    }
}

static void early_console_write ( struct console * con , const char * s ,
unsigned n )
{
    early_write ( s , n );
}

static struct console early_console_dev = {
    . name = "earlycon" ,
    . write =      early_console_write ,
    . flags =      CON_PRINTBUFFER | CON_BOOT ,
    . index = - 1 ,
};

static int __init setup_early_printk ( char * buf )
{
    early_console = & early_console_dev ;
    register_console ( & early_console_dev );
    return 0 ;
}

early_param ( " earlyprintk " , setup_early_printk );

```

The entire implementation is clear at a glance , so I wo n't repeat it . The above code will eventually call **printch** to print the message , so where is printch implemented ?

printch implementation

It is defined in arch/arm/kernel/debug.S .

The assembly file will eventually call addruart, waituart, senduart, busyuart .

So you need to have an assembly file to implement the above 4 functions . Generally speaking , semiconductor manufacturers will write these functions .

So how to see whether the original provides these functions do ? Debug.S is how to call these functions it (also need menuconfig open in CONFIG_DEBUG_LL) ? We will in the next section to explain (on the next section to explain More appropriate).

When the parameters are parsed , we can use `early_printk` to print the message .

`early_printk` in kernel / `printk` / `printk.c` implemented in , code is as follows :

```
#ifdef CONFIG_EARLY_PRINTK
struct console * early_console ;

void early_vprintk ( const char * fmt , va_list ap )
{
    if ( early_console ) {
        char buf [ 512 ];
        int n = vsnprintf ( buf , sizeof ( buf ) , fmt , ap );

        early_console -> write ( early_console , buf , n );
    }
}

asmlinkage __visible void early_printk ( const char * fmt , ...)
{
    va_list ap ;

    va_start ( ap , fmt );
    early_vprintk ( fmt , ap );
    va_end ( ap );
}
#endif
```

The code implementation is also very clear .

Note : From the above implementation , `early_printk` with `printk` not the same , it does not print this level say , everything calls `early_printk` print news , unconditionally displayed in `early_printk.c` registered in `earlycon` the console on .

2.4 early_print

Why are there more `early_prints`?

The reason is similar , the kernel boot process , if the bootargs before parameter analysis , kernel and hung up , this time you use `early_printk` not see any messages . At this point you need to use `early_print`.

The usefulness of `early_print` is that we often encounter a situation :

uncompressing linux...ok, booting the kernel

Then , there is no more , the kernel is stuck , there are no messages , we don't know what happened , at this time we need to use `early_print` to debug .

If you want to use `early_print`, you need to do the following things :

- In `menuconfig` in , turn on `CONFIG_DEBUG_LL`
- assembly language functions : `addruart` , `waituart` , `senduart` , `busyuart` these functions , using the UART print information

early_print code is in arch / arm / kernel / setup.c achieved in :

```
void __init early_print ( const char * str , ...)  
{  
    extern void printascii ( const char *);  
    char buf [ 256 ];  
    va_list ap ;  
  
    va_start ( ap , str );  
    vsnprintf ( buf , sizeof ( buf ), str , ap );  
    va_end ( ap );  
  
#ifdef CONFIG_DEBUG_LL  
    printascii ( buf );  
#endif  
    printk ( "%s" , buf );  
}
```

From the above code we can know :

If CONFIG_DEBUG_LL is turned on , early_print will print the message through printascii .

At the same time , it will also call printk, which means that even if you do not turn on CONFIG_DEBUG_LL , the message printed by early_print will eventually be displayed through printk .

So where is printascii implemented ?

On one we talk , prntch is in arch / arm / kernel / debug.S implemented in , printascii is implemented in the file , the code is as follows :

```
#if !defined(CONFIG_DEBUG_SEMIHOSTING)  
#include CONFIG_DEBUG_LL_INCLUDE  
#endif  
  
#ifdef CONFIG_MMU  
    . macro addruart_current , rx , tmp1 , tmp2  
    addruart      \tmp1 , \tmp2 , \rx  
    mrc p15 , 0 , \rx , c1 , c0  
    tst \rx , # 1  
    moveq \rx , \tmp1  
    movne \rx , \tmp2  
    . endm  
  
#else /* !CONFIG_MMU */  
    . macro addruart_current , rx , tmp1 , tmp2  
    addruart \rx , \tmp1  
    . endm  
  
#endif /* CONFIG_MMU */  
...  
#ifndef CONFIG_DEBUG_SEMIHOSTING
```

```

ENTRY ( printascii )
    addruart_current r3 , r1 , r2
    b    2f
1 : waituart r2 , r3
    senduart r1 , r3
    busyuart r2 , r3
    teq r1 , # '\n'
    moveq r1 , # '\r'
    beq 1b
2 :      teq r0 , # 0
    ldrneb r1 , [ r0 ], # 1
    teqne r1 , # 0
    bne 1b
    ret lr
ENDPROC ( printascii )

ENTRY ( printch )
    addruart_current r3 , r1 , r2
    mov r1 , r0
    mov r0 , # 0
    b    1b
ENDPROC ( printch )

...
#else
...
#endif

```

Code logic is very clear , the only question is , are listed in blue 4 functions Where did achieve it ?

Generally, semiconductor manufacturers will implement these 4 functions .

So how to check whether the original factory is implemented? How does head.S call these 4 functions ?

Pay attention to the top statement of the above code :

```

#if !defined(CONFIG_DEBUG_SEMIHOSTING)
#include CONFIG_DEBUG_LL_INCLUDE
#endif

```

CONFIG_DEBUG_LL_INCLUDE actually refers to the path of an assembly code, and the 4 functions implemented by the original factory are placed in the assembly code . #include means to quote this assembly code .

So where is the path referred to by CONFIG_DEBUG_LL_INCLUDE ?

CONFIG_DEBUG_LL_INCLUDE is in arch / arm / Kconfig.debug assigned in , arch / arm / Kconfig will automatically referenced Kconfig.debug:

```

config DEBUG_LL_INCLUDE
string
default "debug/sa1100.S" if DEBUG_SA1100
.....
default " debug/omap2plus.S " if DEBUG_OMAP2PLUS_UART

```


debug/omap2plus.S is the path where the original assembly code is located, and its actual directory is arch/arm/include/debug/omap2plus.S

Check omap2plus.S, all 4 functions are in it .

At this point , you should understand how early_print works .

2.5 print stack

Kernel provides a print function call stack API, to better understand the stack of backtracking process , you might need a little stack frame of knowledge , following the concept of the stack frame when first introduced , the method describes the printing of the call stack .

Stack frame

Understanding the concept of stack frames is very helpful for us to view the stack dump information in the crash log .

// Brief description of the following two web pages

// Understand APCS-ARM procedure call standard (<http://blog.csdn.net/sunny04/article/details/40456259>) : metaphor to describe the push and pop process of ARM registers

// Introduction to (<http://www.linuxidc.com/Linux/2013-03/81247.htm>) ARM FP register and frame pointer (<http://www.linuxidc.com/Linux/2013-03/81247.htm>) : schematic diagram of the structure of the stack frame

Personal understanding of stack frame principle (add @2019.02.28):

We currently EBP value , you can manually derive the entire call stack , derivation is :

- Step1: Assign the current ebp to esp
- Step2: Pop out a data from the current esp and store it in ebp
- Step3: Then from esp then pop a data , this data is eip (note that it is the next instruction eip, if you want to calculate the call stack is currently being executed , may need to eip subtracting a value , obtained on an instruction Address . Different instructions have different lengths , so the amount of subtraction is not a fixed value .)
- Repeat the above process , can be obtained throughout the call stack

Implementation principle of Linux kernel print stack (add @2019.02.28): Implementation of dump_stack() in the kernel , and simulate dump_stack() in user mode (https://blog.csdn.net/jasonchen_gbd/article/details/44066815)

那dump_stack的工作流程就很清楚了。我就不帖代码了，因为基本上都是体系结构相关的操作。

需要说明的一个地方是，通过函数的地址来打印函数名是通过格式控制符%pS来打印的：

```
printk("[<%p>] %pS\n", (void *) ip, (void *) ip);
```

在内核代码树的lib/vsprintf.c中的pointer函数中，说明了printk中的%pS的意思：

```
1 | case 'S':  
2 |     return symbol_string(buf, end, ptr, spec, *fmt);
```

即'S'表示打印符号名，而这个符号名是kallsyms里获取的。

可以看一下kernel/kallsyms.c中的kallsyms_lookup()函数，它负责通过地址找到函数名，分为两部分：

1. 如果地址在编译内核时得到的地址范围内，就查找kallsyms_names数组来获得函数名。
2. 如果这个地址是某个内核模块中的函数，则在模块加载后的地址表中查找。

kallsyms_lookup()最终返回字符串“函数名+offset/size[mod]”，交给printk打印。

关于内核符号表kallsyms_names可参考我的另一篇文章[点击打开链接](#)。

Printing method

dump_stack

The kernel provides an interface to print the call stack . You only need #include <linux/printk.h> to print the call stack using the dump_stack() function .

dump_stack can print out the correspondence between the kernel symbol table and the function name and address of the kernel module symbol table . For example, if we add a dump_stack() to the kernel module , we can get the following information :

```
[21526.804155] Hardware name: Generic AM33XX (Flattened Device Tree)  
[21526.804185] [<c0015e8d>] (unwind_backtrace) from [<c0012579>] (show_stack+0x11/0x14)  
[21526.804197] [<c0012579>] (show_stack) from [<c03ddf7d>] (dump_stack+0x5d/0x6c)  
[21526.804215] [<c03ddf7d>] (dump_stack) from [<bf83a043>] (alloc_page_owner+0x42/0x5c [page_owner_test])  
[21526.804232] [<bf83a043>] (alloc_page_owner [page_owner_test]) from [<bf83c03d>] (hello_init+0x3c/0x67 [page_owner_test])  
[21526.804248] [<bf83c03d>] (hello_init [page_owner_test]) from [<c0009713>] (do_one_initcall+0x9b/0x198)  
[21526.804261] [<c0009713>] (do_one_initcall) from [<c00fb215>] (do_init_module+0x4d/0x310)  
[21526.804279] [<c00fb215>] (do_init_module) from [<c00a116b>] (load_module+0x16eb/0x1b80)  
[21526.804289] [<c00a116b>] (load_module) from [<c00a17c7>] (SyS_finit_module+0x77/0x9c)  
[21526.804303] [<c00a17c7>] (SyS_finit_module) from [<c000ed21>] (ret_fast_syscall+0x1/0x52)
```

WARN_ON

Warn_on (the include / Generic-ASM / bug.h) encapsulating the dump_stack (), it can also be used to print the call stack .

BUG_ON

BUG_ON (include/asm-generic/bug.h) will call panic(), and the panic function will call dump_stack() to print out the call stack .

But panic will make a kernel panic stuck , therefore caution BUG_ON.

3 query debugging

In Chapter 2, we introduced print debugging . Print debugging is very common , but it also has its shortcomings . Many print debugging uses serial port to print information , but serial port is serial transmission , the speed is very slow , a printk may be It takes about 10ms .

Too much printing information will affect the performance of the system . Therefore, when releasing the code , the printing information is generally reduced as much as possible . For the Linux kernel , after the kernel is started and entered into the file system , no information should be printed under normal circumstances. .

In view of the above shortcomings , we introduce another debugging method : query debugging .

The so-called query debugging means that when I need certain debugging information , the kernel code will generate this information .

The general query debugging has the following types :

3.1 sysfs

For the time relationship , I won't go into details about these kinds of things , and I will improve them later .

3.2 proc

proc belong to one of the virtual file system , we often by cat / proc / xxx some of the information to query the kernel , such as cat / proc / meminfo.

Many subsystems in the kernel use proc, especially some core systems such as process scheduling and memory management . If you want to view information through /proc , you need to do the following two actions :

1. When compiling the kernel , turn on the switch CONFIG_PROC_FS < Many kernels turn on this switch by default when compiling >
1. After the system is running , use the following command mount proc < Many file systems will automatically mount proc>
sudo mount -t proc none /proc

Then , cat / proc / xxx can .

If we own in the preparation of the drive , but also want to use the proc, how to do it ?

In fact, very short answer , since the proc file system , even if it is virtual , the file system will deal with 2 main actions : create files or create folders .

The **header file** provided by proc to other modules is : include/linux/proc_fs.h

The **implementation file** of proc is : fs/proc/generic.c

If you want to use proc, you only need to #incude the above header file in your driver , then prepare the file_operations structure , and then call the API provided by proc to create the file .

Take /proc/meminfo as an example to see how to write the code :

```
static const struct file_operations meminfo_proc_fops = {  
    . open      = meminfo_proc_open ,
```

```

        . read      = seq_read ,
        . llseek    = seq_lseek ,
        . release    = single_release ,
};

static int __init proc_meminfo_init ( void )
{
    proc_create ( "meminfo" , 0 , NULL , & meminfo_proc_fops );
    return 0 ;
}

fs_initcall ( proc_meminfo_init );

```

In addition to creating files , you can also create folders , or create files in a folder . For more APIs , please see the source code .

3.3 debugfs

Debugfs is one of the virtual file systems . You can use debugfs to provide some debugging information to the user space .

Many subsystems in the kernel have already used debugfs. If we want to view this information , we need to do two actions :

1. When compiling the kernel , turn on the switch CONFIG_DEBUG_FS . <Many kernels have this switch turned on by default when compiling >
1. After the system is running , use the following command mount debugfs


```
sudo mount -t debugfs debugfs /sys/kernel/debug/
```

Then , you can in / sys / kernel / debug / see a lot of files or folders directory , cat these files , you can get a debugging information you need .

If we own in the preparation of the drive , but also want to use debugfs to show some debug information to the user space , how to do it ?

In fact, very short answer , since debugfs file system , even if it is virtual , the file system will deal with 2 main actions : create files or create folders .

The **header file** provided by debugfs to other modules is : include/linux/debugfs.h

The **implementation file** of debugfs is : fs/debugfs/inode.c

If you want to use debugfs, you only need to #include the above header file in your driver , then prepare the file_operations structure , and then call the API provided by debugfs to create the file .

Take the GPIO subsystem as an example . The GPIO subsystem creates a file under debugfs with the name " gpio " .

```

static const struct file_operations gpiolib_operations = {
    . owner      = THIS_MODULE ,
    . open       = gpiolib_open ,
    . read       = seq_read ,
    . llseek     = seq_lseek ,
    . release    = seq_release ,
};

```

```
static int __init gpiolib_debugfs_init ( void )
{
    /* /sys/kernel/debug/gpio */
    ( void ) debugfs_create_file ( "gpio" , S_IFREG | S_IRUGO ,
        NULL , NULL , & gpiolib_operations );
    return 0 ;
}
subsys_initcall ( gpiolib_debugfs_init );
```

The code demonstrates how to create a file , when you mount the debugfs, then cat / sys / kernel / debug / gpio, cat operation is called to open & read function , read the function will return to the debugging information to the user space .

As for how to implement file_operations , it is your own business .

In addition to creating files , you can also create folders , or create files in a folder . For more APIs , please see the source code .

3.4 ioctl

3.5 ecall

e call is a system made by HiSilicon , which can view the value of any register in the user space or call any function in the kernel .

The system is divided into a kernel part and an application program . The kernel part [hisi-easy-shell.c](https://github.com/Johnliuxin/HUAWEI-P9-Lite_OpenSource/blob/master/drivers/hisi/mntn/hisi-easy-shell.c) (https://github.com/Johnliuxin/HUAWEI-P9-Lite_OpenSource/blob/master/drivers/hisi/mntn/hisi-easy-shell.c) will eventually provide a device node , and the application program (no code yet) interacts with this device node through ioctl .

3.6 SysRq

Official website : <https://www.kernel.org/doc/html/latest/admin-guide/sysrq.html> (https://www.kernel.org/doc/html/latest/admin-guide/sysrq.html)

The introduction of commands on IBM is more detailed : <https://www.ibm.com/developerworks/cn/linux/l-cn-sysrq/> (https://www.ibm.com/developerworks/cn/linux/l-cn-sysrq/)

C o m m a n d	Function
b	Will immediately reboot the system without syncing or unmounting your disks.

C o m m a n d	Function
c	Will perform a system crash by a NULL pointer dereference. A crashdump will be taken if configured.
d	Shows all locks that are held.
e	Send a SIGTERM to all processes, except for init.
f	Will call the oom killer to kill a memory hog process, but do not panic if nothing can be killed.
g	Used by kgdb (kernel debugger)
h	Will display help (actually any other key than those listed here will display help. but h is easy to remember 😊)
i	Send a SIGKILL to all processes, except for init.
j	Forcibly “Just thaw it” – filesystems frozen by the FIFREEZE ioctl.
k	Secure Access Key (SAK) Kills all programs on the current virtual console. NOTE: See important comments below in SAK section.
l	Shows a stack backtrace for all active CPUs.
m	Will dump current memory info to your console.
n	Used to make RT tasks nice-able
o	Will shut your system off (if configured and supported).
p	Will dump the current registers and flags to your console.
q	Will dump per CPU lists of all armed hrtimers (but NOT regular timer_list timers) and detailed information about all clockevent devices.
r	Turns off keyboard raw mode and sets it to XLATE.
s	Will attempt to sync all mounted filesystems.
t	Will dump a list of current tasks and their information to your console.
u	Will attempt to remount all mounted filesystems read-only.
v	Forcefully restores framebuffer console

C o m m a n d	Function
v	Causes ETM buffer dump [ARM-specific]
w	Dumps tasks that are in uninterruptable (blocked) state.
x	Used by xmon interface on ppc/powerpc platforms. Show global PMU Registers on sparc64. Dump all TLB entries on MIPS.
y	Show global CPU Registers [SPARC-64 specific]
z	Dump the ftrace buffer
0 – 9	Sets the console log level, controlling which kernel messages will be printed to your console. (0 , for example would make it so that only emergency messages like PANICs or OOPSes would make it to your console.)

4 Disassembly

The main purpose of disassembly is to find the corresponding code through the abnormal address . This chapter introduces some related concepts and tools .

4.1 ELF

ELF stands for Executive and Linkable Format. It is a file format that can be used to describe object files (that is, obj files generated by compilation), executable files (that is, executable programs generated by linking) and library files .

When the system fails , it is often the address information that was shown to us when the error occurred. What we need is to convert the address information to the corresponding C/C++ code , and then analyze the code to find the cause of the error .

To better understand the meaning of the information of these addresses , we need to have a little background knowledge of ELF , so we first briefly introduce the ELF file format .

ELF has been a standard format under Linux for a long time, replacing the early a.out format. A special advantage of ELF is that the same file format can be used on almost all architectures supported by the kernel. This not only simplifies the creation of user-space tool programs, but also simplifies the programming of the kernel itself. For example, the design of the loader.

But the same file format does not mean that there is binary compatibility between programs on different systems. For example, FreeBSD and Linux both use ELF as the binary format. Although the two methods of organizing data in files are the same, there are still differences in the system call mechanism and the semantics of the system call. This is also the reason why FreeBSD programs cannot run under Linux without an intermediate emulation layer (and vice versa) .

In addition, binary programs cannot be exchanged between different architectures (for example, Linux binary programs compiled for Alpha CPU cannot be executed on Sparc Linux), because the underlying architecture is completely different. However, due to the existence of ELF , the relevant information of the program itself and the way in which the various parts of the program are encoded in the binary file are the same for all architectures.

Linux uses ELF not only for user space applications and libraries, but also for building modules. The kernel itself is also in ELF format. ELF is an open format, and its specifications are freely available.

4.1.1 Example code

In the later introduction of , we will explain the principles of the edge , the edge demonstration examples . The demonstration will need to compile C code that , so we here a short list of C code that , to prepare for later use .

```
#include<stdio.h>

int add ( int a , int b ) {
    printf ( "Numbers are added together\n" );
    return a + b ;
}

int main () {
    int a , b ;
    a = 3 ;
    b = 4 ;

    int ret = add ( a , b );
    printf ( "Result: %d\n" , ret );

    return 0 ;
}
```

Then we use gcc to compile this code to generate an object file and an executable file :

```
gcc -c main.c -> main.o ( object file )
```

```
gcc -o main main.o -> main ( executable file )
```

4.1.2 ELF layout and structure

Refer to "In-depth Linux Kernel Architecture Appendix E " to complete this chapter

ELF header : readelf - h

You can use readelf -h or file xxx to determine whether a file is in ELF format .

Refer to "In-depth Linux Kernel Architecture Appendix E " to complete this chapter

Program header table : readelf - l

Section / Segment : readelf - S

Section / segment , also called section.

Symbol table : readelf -s / objdump -t / nm

We introduced on an ELF various existing sections, symbol table also belongs to section one (.symtab) .

The symbol table is an important part of each ELF file , because it stores the global variables and functions used by the program . These global variables and functions include those defined by the program itself and those defined in the external libraries used by the program .

For the global variables and functions defined by the program itself , since their runtime addresses are determined during the linking period , it conforms to the one-to-one correspondence between these variables and functions and their addresses stored in the table .

For the global variables and functions defined in the external library , if it is statically linked , then the one-to-one correspondence with the address can also be stored in the compliance table ; if it is a dynamic link (such as the .so library , it is a redirectable ELF Format file) , because it dynamically determines the operating address of the function during operation (using *ld-linux.so*) , only the referenced variable / function name is stored in the symbol table , and the one-to-one correspondence with the address cannot be determined .

Let's look at an example to deepen our understanding (using the main.o and main compiled in the previous article)
.


The nm tool can generate a list of all symbols defined and used by the program , let's take a look at main.o first :

```
liuxin@ubuntu:~/gcc-g$ nm main.o
0000000000000000 T add
0000000000000022 T main
                 U printf
                 U puts
```

- 左列给出了符号的值，即，符号在目标文件中定义的位置（注意这个值并不是运行时的地址，因为链接动作还没有在这个时间进行，而运行地址只能链接后得到确认）。
- 例子包括两种不同类型的符号，程序自身实现的 add / main 函数存储在文本段（缩写 T 指示），而未定义引用 / puts printf 的 U 指示。未定义引用值没有符号。

We use nm to see the tool main . There will be more symbols in the executable file . But because most of them are automatically generated by the compiler , for running internal system , the following examples given only appear in both target Symbols in the file :

```
liuxin@ubuntu:~/gcc-g$ nm main
0000000000400566 T add
.....
0000000000400598 T main
                 U printf@@GLIBC_2.2.5
                 U puts@@GLIBC_2.2.5
.....
```

-  address listed on the left is the address during function operation .
- printf/puts is still undefined , which means that dynamic linking was used during linking , but some information was added at the same time , indicating the name and minimum version of the GNU standard library that can provide functions .

readelf -s / objdump -t can provide similar information , you can use **readelf -s main.o** and **readelf -s main** to view , I won't go into details here .

How does ELF implement the symbol table mechanism? The following .3 th section for receiving the relevant data.

- .symtab determines the association between the name of the symbol and its value. But not the name of the symbol appearing directly as a string, and is expressed as an index of an array of strings
- .strtab saves an array of strings
- .hash saves a hash table to help quickly find symbols

I won't go into more details here . If you are interested, you can study it online .

The key points I want to express here are :

Due to the existence of this section of the table , we can locate the function by address ! When the program crashes , we can generally get the address pointed to by the PC pointer at the time of the crash . Through this address , we can use **addr2line/nm/objdump** to get the corresponding Function name . The specific method will be introduced later .

But there is still a shortcoming , we can only get the function name , but we can't know which line in which c file the function is . If there are multiple c files in the system , the situation will be very bad .

For example, the kernel crashes , and then you locate the name of the crashed function , and then find the corresponding c file in the vast code , which is also very time-consuming .

There is a solution , just add the **-g** option when compiling gcc . For details, see section 4.1.4 .

4.1.3 ELF data structure in the kernel

Reference "in-depth Linux kernel architecture Appendix E " , do not intend to fine writing , there is time to add

4.1.4 gcc -g -rdynamic & strip

-g & strip

" 4.1.2 Symbol Table" one of the last to explain why use **-g** option , it is recommended to look back at "symbol table" .

When we use **gcc -g** command to compile the code , will ELF increases file inside 5 debug sections, which are :
.debug_aranges , **.debug_info** , **.debug_abbrev** , **.debug_line** , **.debug_str** .

You can compile the example code of Section 4.1.1 with **gcc -o main main.c** and **gcc -g -o main-g main.c** respectively , and then use **readelf -S main** and **readelf -S main-g** to compare the sections of the two .

Extra 5 debug sections in , **.debug_line** this section inside the store is the function " address during operation " and " file path and line number where the function " correspondence between .

You can use **readelf -wl main-g** View **.debug_line** this section details of . (**Readelf -debug-dump** can view 5 debug section full content) .

With `.debug_line` this section, we can easily locate the source file and line number Error path through procedural failures address . Mating " symbol table " in this section locate the name of the function , we can basically lock error Code .

strip command to remove the above-mentioned 5 Ge sections, such as :

`strip main-g`

After this processing , `main-g` is exactly the same as `main` ! One use of this command is that we can use during debugging -g compile the code , with the time of publication mirrored strip to remove the mirror in `.debug_XXX` sections.

– Rdynamic & s trip

-g is a compile option , -rdynamic is a link option , it will instruct the linker to add all symbols (not just the external symbols used by the program) to the dynamic symbol table (that is, the `.dynsym` table) , So that functions like `dlopen()` or `backtrace()` (this series of functions use symbols in the `.dynsym` table) can be used .

-rdynamic details please refer to : <http://www.cnblogs.com/LiuYanYGZ/p/5550544.html>
(<http://www.cnblogs.com/LiuYanYGZ/p/5550544.html>)

I don't look into the time relationship in depth here . Note that strip cannot be deleted . The content in this section of `dynsym` !

4.2 addr2line

Error location method :

`addr2line -fe main 40057f` : main is an executable file , followed by the value of the PC register when the error occurred .

- executable program can be a user-space program , it may be kernel `vmlinux`
- If the -g option is added when compiling , the above command will display the error function name and the file location and number of lines where the function is located . Otherwise, only the function name can be displayed .
- Note that the `addr2line` corresponding to the compiler must be used , for example, if `gcc` is `$CROSS_COMPILE-gcc`, you need to use `$CROSS_COMPILE-addr2line`
- `addr2line --help` can view its more options

4.3 objdump

The `objdump` command is a command to disassemble object files or executable files under Linux . It has other functions . The following takes the ELF format executable file `test` as an example to introduce in detail :

- `objdump -f test`
Display the file header information of `test`
- `objdump -d test`
Disassembly `test` to those in need of instruction execution section
- `objdump -D test`
Similar to -d , but disassemble all sections in `test`
- `objdump -h test`
Display `test` of Section Header information

- `objdump -x test`
Display all header information of test
- `objdump -s test`
In addition to displaying all the header information of the test , it also displays their corresponding hexadecimal file codes

How to disassemble any binary file ?

We can do this :

`objdump -D -b binary -m i386 a.bin`

-D means to disassemble all files , -b means binary , -m means instruction set architecture , a.bin is the binary file we want to disassemble .

`objdump -m` to see more support instruction set architecture , such as i386: x86-64, i8086 and so on .

We can also specify the big-endian or Little-endian (-EB or -EL) , we can specify from one location to start disassembling and so on . So `objdump` command is very powerful!

Also , pay attention to using `objdump` corresponding to the compiler .

4.4 nm & ldd




`ldd` is a tool used to analyze the dynamic library that the program needs to rely on when running ; `nm` is a tool used to view the related content of the symbol table in the specified program . You can view more information through -help .

With the help of " 4.1.1 Example Code" , take a look at the output of the two :

ldd main

```
liuxin@ubuntu:~/gcc-g$ ldd main
linux-vdso.so.1 => (0x00007ffe14610000)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007fad10476000)
/lib64/ld-linux-x86-64.so.2 (0x000055cc8d37f000)
```

In the above example , the results of `ldd` can be divided into three columns :

-  first column : what libraries the program needs to rely on
-  second column : the library provided by the system corresponding to the library required by the program
-  third column: the starting address of the library loading

Through the above information, we can get the following information

- by comparing the first and second columns , we can analyze a program library and system relies actually provided , matches
- By observing the third column , we can know the starting position of the symbol in the current library in the address space of the corresponding process

nm

```

liuxin@ubuntu:~/gcc-g$ nm main
0000000000400566 T add
0000000000601040 B __bss_start
0000000000601040 b completed.7585
0000000000601030 D __data_start
0000000000601030 w data_start
00000000004004a0 t deregister_tm_clones
0000000000400520 t __do_global_ctors_aux
0000000000600e18 t __do_global_ctors_aux_fini_array_entry
0000000000601038 D __dso_handle
0000000000600e28 d _DYNAMIC
0000000000601040 D _edata
0000000000601048 B _end
0000000000400654 T _fini
0000000000400540 t frame_dummy
0000000000600e10 t __frame_dummy_init_array_entry
00000000004007d8 r __FRAME_END__
0000000000601000 d _GLOBAL_OFFSET_TABLE_
000000000040068c r __GNU_EH_FRAME_HDR
0000000000400400 T _init
0000000000600e18 t __init_array_end
0000000000600e10 t __init_array_start
0000000000400660 R _IO_stdin_used
0000000000600e20 d __JCR_END__
0000000000600e20 d __JCR_LIST__
0000000000400650 T __libc_csu_fini
00000000004005e0 T __libc_csu_init
0000000000400598 T main
00000000004004e0 t register_tm_clones
0000000000400470 T _start
0000000000601040 D __TMC_END__

```

The format of the content is as follows :

- *ℳ first column : the address of the current symbol
- *ℳ second column : the type of the current symbol (for the description of the type, interested friends can read it in man nm)
- *ℳ third column : the name of the current symbol

nm for our program so what specific help it , I think mainly the following aspects

- Determine whether the specified symbol is defined in the specified program (commonly used method: nm -C proc | grep symbol)
- solving compiled undefined reference errors , as well as mutiple definition of error
- view an address of a symbol , as well as the approximate location of the process space (BSS, Data, text areas , particularly the second column by the class type is determined)

4.5 How to locate the known PC address

Reference " 4.2 addr2line " , using addr2line -fe program pcaddress to obtain pc of code bits corresponding to the address counter .

4.6 How to locate function+offset/size

objdump + addr2line

In the process of a system crash , we get the following information , the crashed module is tejaxpci.ko:

The specific location is : shtej_spanconfig+0x98/0x123

Use first after

```
#objdump -S tejaxpci.ko
```

Get shtej_spanconfig address 0x248a, plus 0x98 is , 0x2522.

After getting the address of 0x2522 , run

```
#addr2line -fe tejaxpci.ko 0x2522
```

You can locate the corresponding line of code .

If you encounter a situation where objdump cannot be used , you can use the following command to check the symbol list in the bin file in ELF format

```
readelf -s output/debug/vmlinux | grep xxx
```

nm + objdump

The error message shows that PC is at snd_soc_dai_set_sysclk+0x10/0x84

0x10 : indicates the offset position of the error ; 0x84 indicates the size of the snd_soc_dai_set_sysclk function

First find the location of the snd_soc_dai_set_sysclk function

```
1. $ arm-linux-gnueabi-nm vmlinux | grep snd_soc_dai_set_sysclk
1. c04116bc T snd_soc_dai_set_sysclk
```

Then objdump out this function

```
1. arm-linux-gnueabi-objdump -S vmlinux -start-address=0xc04116bc -stop-address=0xc04116bc >
~/temp/soc
```

Next, check the vim ~/temp/soc file and find the location of 0xc04116bc+0x10

4.7 How to locate the crash in the so library

Crash log example

```
// Register stack information
[ 2 ][ 18.572091 ] Pid : 678 , comm : minismarthub
[ 2 ][ 18.577125 ] CPU : 2 Tainted : PO ( 3.8.13 #1 )
[ 2 ][ 18.582943 ] PC is at 0x150ac58
[ 2 ][ 18.586367 ] LR is at 0xa96a3460
[ 2 ][ 18.589892 ] pc : [< 0150ac58 >] lr : [< a96a3460 >] psr :
60000010
[ 2 ][ 18.589892 ] sp : 9aa7eb48 ip : 0150ac50 fp : 9aa7eb64
[ 2 ][ 18.602218 ] r10 : 00000000 r9 : 00004ae0 r8 : 00000001
[ 2 ][ 18.607854 ] r7 : a055d478 r6 : a055d6a4 r5 : a96f85d4 r4 : 00000b64
[ 2 ][ 18.614815 ] r3 : 0000272b r2 : ffffffff r1 : 00000001 r0 : 00000b64

//dump map information
[ 2 ][ 18.747704 ] _____
[ 2 ][ 18.754763 ] * dump maps on pid ( 678 - minismarthub )
[ 2 ][ 18.760043 ] _____
... ..
[ 2 ][ 21.700779 ] a963f000 - a977c000 r - xp 00000000 b3 : 0f 495 /
mtd_rwarea / libMiniSmartHubApp . So // the address allocated by the
libMiniSmartHubApp.so code, pay attention to the starting address a963f000
```

```
... ..
```

```
addr2line -Cfe exeAPP 0xa96a3460 // exeAPP is an executable program
??
??:0
```

Here there is a problem , 0xa96a3460 the corresponding function is a so inside the function , so the direct use addr2line not print out the name of the function . The reason is so inside address when the executable file is loaded , can be reallocate 's , this address It is dynamic and cannot be determined during compilation , so it is useless to directly use the runtime address .

How to solve ? Use addr2line or gdb can , described below, respectively .

addr2line

The idea is very simple , to convert the run-address so the internal offset , then you can use addr2line up .

For example , the " the dump Map Information " understood , libMiniSmartHubApp.so is loaded into a963f000-a977c000 between , 0xa96a3460 this address is within the range , so that it is libMiniSmartHubApp.so inside a function . $0xa96a3460 - a963f000 = 0x64460$ is the function in so offset inside .

```
addr2line - Cfe libMiniSmartHubApp . so 0x64460
CMNMiniHubView::StartViewTimer()
/home/sundh/GolfP/AP_MM/AP_MiniSmartHub/Src/MNMiniHubView.cpp:1689
```

You can get the function name .

gdb

If you want to use gdb, just type the following command to let gdb load so.

```
( gdb ) set solib - search - path / path / to / the / so
```

Then you can use the gdb command to debug normally , for example

```
( gdb ) bt
```

4.8 oops analysis and positioning

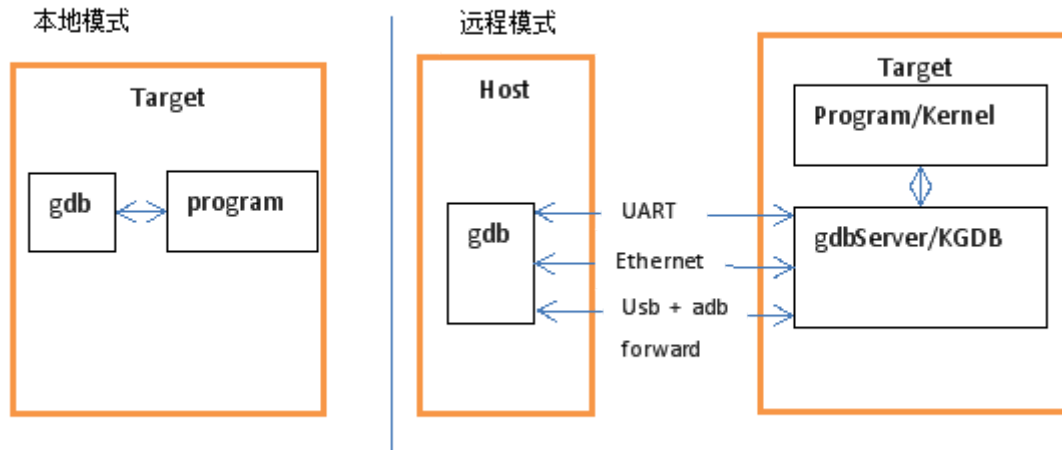
Oops displays the state of the processor when the error occurs, including the contents of the CPU register, the location of the page descriptor table, and some incomprehensible information. These messages are generated by the printk statement in the failure handling function (arch/*/kernel/traps.c) . The more important information is the instruction pointer (EIP), which is the address of the error instruction .

This problem positioned generally will be enabled kernel `DEBUG_INFO = the Y` (i.e. open `-g` option) , and then recompiling the kernel run , then in accordance with the error address , by the above-described embodiment disassembly positioning error codes position .

5 G db & KGDB

[Note add @ 1018-10-19] There is now an updated debugger LLDB , which has a tendency to replace gdb .

From a macro say , gdb class debugging tools of the operating mode can be divided into local and remote control modes : Local mode means that gdb and the debugger running on the same machine ; remote mode refers to gdb and the commissioning procedures in different of Running on the machine , the two communicate via UART or Ethernet .



From the point of view of debugging tools for , it can be divided into static method and dynamic method : Static method under , gdb can

- Carry out some disassembly things , such as " 5.1. 3 gdb disassembly "
- analysis core dump file , such as " 5.1. 4 gdb analysis core dump "

Under the dynamic method , gdb can

- trace debugging a new program , you can single-step , set breakpoints, print variable values and so on , just as before with MDK debugging bare the same program , see " 5. 1.2 gdb local debugging "
- Attach to a running program and debug it (subsequent supplement)

No matter what kind of method , are inseparable from gdb command , and therefore " 5.1.1 gdb command " a first introduced gdb 's commands .

5.1 gdb

5.1.1 gdb command

```
liuxin@ubuntu:~/gcc-g$ gdb : enter gdb debugging
```

```
(gdb) help : List all the major categories of commands
```

```
(gdb) help stack : View the specific commands under a certain category , for example, view the stack category .
```

Note that gdb supports tab completion !

Commonly used gdb commands are as follows :

- l(list) , display the source code, and you can see the corresponding line number ;
 - b < line number >
 - b < function name >
 - b *< function name >
 - b *< code address >
- Short for Breakpoint , set breakpoint . You can use "line number", "function name" and "execution address" to specify the location of the breakpoint .
- The " * " symbol is added in front of the function name to set the breakpoint at the " prolog code generated by the compiler " .
- The actual effect is to stop before entering the function .
- I do not understand if any , can 4.1.1 applet section , comparison b add and b * add differences .
- d [number] : Shorthand for Delete breakpoint , delete a breakpoint with a specified number , or delete all breakpoints . The number of breakpoints increases from 1 .
 - p(print)x, x is the variable name, which means to print the value of variable x
 - r(run), which means to continue execution to the position of the breakpoint
 - n(next), which means to execute the next step
 - c(continue), which means to continue execution
 - q(quit) , which means to quit gdb

5.1.2 gdb local debugging

Here is a simple example , so familiar gdb debugger of the basic method .

You can try it on any linux machine , first download the compiler program :

```
git clone https://gitlab.com/study-kernel/debugging_techniques/gdb.git (https://gitlab.com/study-kernel/debugging_techniques/gdb.git)
cd gdb & make
```

Then run gdb to debug :

```
root@embest:~/gdb# gdb ./main
GNU gdb (Debian 7.12-6) 7.12.0.20161007-git
Copyright (C) 2016 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 "arm-linux-gnueabi".
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 ./main...done.
```

start runs to the main entrance :

```
(gdb) start
Temporary breakpoint 1 at 0x103fe: file main.c, line 24.
Starting program: /root/gdb/main

Temporary breakpoint 1, main (argc=1, argv=0xbeffffda4) at main.c:24
24 printf("main function\n");
```

Next single step execution :

```
(gdb) next
main function
25 ... _ coredump_case1();
```

bt print call stack :

```
(gdb) bt
#0 main (argc=1, argv=0xbffffda4) at main.c:25
```

5.1.3 gdb disassembly

Locate the code location by the abnormal address :

Suppose the kernel prints the following error . For example :

Call Trace: [<800a73b8>] do_vfs_ioctl+0x88/0x5c8

If the kernel enables CONFIG_DEBUG_INFO when compiling , we can use the following command to try to locate

```
# gdb vmlinux
```

```
(gdb) list *(0x800a73b8)
```

```
(gdb) list *(do_vfs_ioctl+0x88)
```

Both of the above methods will work .

Follow-up supplement for other situations

5.1.4 gdb analysis coredump

5.1.3.1 core dump

<http://blog.csdn.net/tenfyguo/article/details/8159176/> (<http://blog.csdn.net/tenfyguo/article/details/8159176/>)

What is coredump

We often hear people when it comes to program core out , needs to be positioned to address , said here mostly refers to **the application** , due to various abnormalities or bug cause abnormal exit or abort during operation , and meet certain conditions (Why do you say that certain conditions need to be met here? The following will analyze) A file called core will be generated .

Typically , Core file will contain the run-time program memory , register state , stack pointer , memory management information and various function call stack information , we can understand the current state of the work program is to generate the first file storage . Many programs will generate a core file when there is an error . Through the analysis of this file , we can locate the corresponding stack call when the program exits abnormally , find out the problem and solve it in time .

The storage location of the coredump file

The default storage location of the core file is in the same directory as the corresponding executable program . The file name is core . You can see the location of the core file through the following command :

```
cat /proc/sys/kernel/core_pattern
```

The default value is core

Note : this refers to the current working directory is created under the process . Normally associated with the program under the same path . However, if the program calls chdir function , it is possible to change the current working directory . Then core file is created in chdir specified Under the path . Many programs crashed , but we couldn't find where the core file was placed , which is related to the chdir function . Of course, the program crashes may not necessarily generate the core file .

You can change the storage location of the coredump file with the following command . If you want to generate the core file to the /data/coredump/core directory :

```
echo "/data/coredump/core" > / proc / sys / kernel / core_pattern
```

By default , the core file generated by the kernel during coredump is placed in the same directory as the program , and the file name is fixed as core . Obviously , if multiple programs generate core files , or the same program crashes multiple times , It will overwrite the same core file repeatedly , so it is necessary to name the core files generated by different programs separately .

By modifying the parameters of the kernel , we can specify the dynamic file name of the coredump file generated by the kernel . For example , use the following command to make the kernel generate a core dump file in the format of core.filename.pid :

```
echo "/data/coredump/core.%e.%p" >/ proc / sys / kernel / core_pattern
```

After this configuration , the generated core file will contain the name of the crashed program and its process ID. The %e and %p above will be replaced with the program file name and process ID.

It should be noted , there is a kernel with coredump -related settings , that is, / proc / SYS / Kernel / core_uses_pid . If the contents of this file is configured to 1 , even if core_pattern not set % the p- , the last generation of core The process ID will still be added to the dump file name .

c ore_pattern format

There are many variables that can be used in the core_pattern template , see the list below :

%%	Single % character
%p	Process ID of the dumped process
%u	The dump actual user process ID
%g	The dump actual group process ID
%s	The signal that caused this core dump
%t	core dump time (by the 1970 Nian 1 Yue 1 from date of seconds)
%h	CPU name
%e	Program file name

Summary of some conditions that produce coredum

ulimit -c

To generate coredump conditions , you first need to confirm the ulimit - c of the current session . If it is 0, the corresponding coredump will not be generated and need to be modified and set .

➤ ulimit -c unlimited (coredump can be generated and is not limited by size)

- `ulimit -c 4` (`ulimit -c [size]` can set the size , the unit of size here is blocks, generally 1block=512bytes . I actually measured it , the minimum must be set to 4 and above to generate a coredump file)

Common causes of program coredump

1 : Memory access out of bounds

- Due to the use of the wrong subscript , the array access is out of bounds .
- `strlen` searching for a string , rely on the end of the string to judge whether the string is over , but the string does not use the end of normal .
- use `strcpy`, `strcat`, `sprintf`, `strcmp`, `strcasemp` other string manipulation functions , the target string read / write burst . Should use `strncpy`, `strncpy`, `strncat`, `strlcat`, `snprintf`, `strncmp`, `strncasemp` the like to read and write functions to prevent cross-border .

2 : Multithreaded programs use thread-unsafe functions

3 : Data read and written by multiple threads is not protected by lock

For global data that will be accessed by multiple threads at the same time , you should pay attention to locking protection , otherwise it is easy to cause coredump

4 : Illegal pointer

- Use a null pointer
- Use pointer conversion at will .
A pointer pointing to a section of memory , unless this memory previously allocated to a certain structure or to determine the type , or this type of structure or array , or do not convert it to this type of structure or pointer , but this should be The memory is copied to a structure or type of this type, and then the structure or type is accessed . This is because if the starting address of this memory is not aligned according to this structure or type , then it is easy to access it due to bus error and core dump
- Stack overflow
Do not use large local variables (because local variables are allocated on the stack) , so likely to cause a stack overflow , stack and heap structural damage system , leading to inexplicable error occurs

Description of several situations where core files will not be generated

The core file will not be generated if

- (a) the process was set-user-ID and the current user is not the owner of the program file, or
- (b) the process was set-group-ID and the current user is not the group owner of the file,
- (c) the user does not have permission to write in the current working directory,
- (d) the file already exists and the user does not have permission to write to it, or
- (e) the file is too big (recall the `RLIMIT_CORE` limit in Section 7.11). The permissions of the core file (assuming that the file doesn't already exist) are usually user-read and user-write, although Mac OS X sets only user-read.

How to determine whether a file is a coredump file

In class unix under the system , coredump file format itself is also a major ELF format , therefore , we can readelf judge command .

```

TEG_23_76_sles10_64:/data/coredump # readelf -h core
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                                ELF64
  Data:                                  2's complement, little endian
  Version:                              1 (current)
  OS/ABI:                               UNIX - System V
  ABI Version:                           0
  Type:                                  CORE (Core file)
  Machine:                              Advanced Micro Devices X86-64
  Version:                              0x1
  Entry point address:                   0x0
  Start of program headers:              64 (bytes into file)
  Start of section headers:              0 (bytes into file)
  Flags:                                 0x0
  Size of this header:                   64 (bytes)
  Size of program headers:               56 (bytes)
  Number of program headers:              25
  Size of section headers:               0 (bytes)
  Number of section headers:              0
  Section header string table index:     0

```

You can also quickly judge through the file command :

```

TEG_23_76_sles10_64:/data/coredump # file core
core: ELF 64-bit LSB core file AMD x86-64, version 1 (SYSV), SVR4-style, from 'coremain'

```

Instance

Save it on the BBB board and proceed as follows :

- git clone https://gitlab.com/study-kernel/debugging_techniques/gdb.git (https://gitlab.com/study-kernel/debugging_techniques/gdb.git)
- cd gdb & make
- ulimit -c 4
- ./main

Will get the following result :

```

root@embest:~/gdb# ./main
main function
Segmentation fault (core dumped)

```

In the current directory ls, you will see the core dump file : core

```

root@embest:~/gdb# ls
core  main  main.c  main.d  main.o  Makefile

```

5.1.3.2 gdb debug core dump

Core dump is equivalent to a snapshot of the program at a certain point in time . Through core dump, gdb can reproduce the running situation of the program when an error occurs .

If there is no core dump file , when gdb loads an executable program, it is equivalent to the " main() " of the program , you have to run step by step to the place where the error occurred . Of course , the error at this time and the core dump error may be It's different , because many bugs are random .

So , best to have core dump file , the collapse of the cadaver left , in order to analyze anatomy back!

If you get the core dump file (with the help of the example in " core dump " , assuming that the dumped file is named core and the corresponding executable program is main) , the debugging steps are as follows :

gdb main core

```
root@embest:~/coredump# gdb main core
GNU gdb (Debian 7.7.1+dfsg-5) 7.7.1
Copyright (C) 2014 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 "arm-linux-gnueabi".
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 main...done.
BFD: Warning: /root/coredump/core is truncated: expected core file size >= 225280, found: 2120.
[New LWP 599]
Failed to read a valid object file image from memory.
Core was generated by './main'.
Program terminated with signal SIGSEGV, segmentation fault.
#0 0x000103e8 in case_1 () at main.c:4
4      *ptr=0;
(gdb) bt
Python Exception <class 'gdb.MemoryError'> Cannot access memory at address 0xbe9edcb4:
#0 0x000103e8 in case_1 () at main.c:4
#1 0x00010408 in main () at main.c:10
(gdb)
```

- ♣ soon as gdb is started , we can see the function at the time of the error and the file and line number where it is located .
- ☞ bt command can display the call stack to the error location

Of course, this is just a very simple example , the complex may not be obvious at a glance , but you can use this idea to slowly debug .

5.1.5 gdb remote debugging

gdb remote debugging generally used IDE above . For example Android official of NDK, NDK provides an eclipse of a plug-in , for editing, compiling code ; in addition NDK also provides gdb and gdbserver, gdbserver in running on the phone , gdb in PC on cooperation eclipse runs , which can provide users with a graphical debugging interface .

In our usual debugging process , if the scene requires , we can also use the remote mode . The following shows some of the necessary settings for remote debugging .

Use remote debugging , gcc , gdb , gdbserver which the three best supporting .

We at BBB running debian , for example , first in the PC on download gcc compiler : <https://releases.linaro.org/components/toolchain/binaries/5.3-2016.02/arm-linux-gnueabi/> (https://releases.linaro.org/components/toolchain/binaries/5.3-2016.02/arm-linux-gnueabi/) , according to PC 's model , select The corresponding compressed package (for example, gcc-linaro-5.3-2016.02-x86_64_arm-linux-gnueabi.tar.xz (https://releases.linaro.org/components/toolchain/binaries/5.3-2016.02/arm-linux-gnueabi/gcc-linaro-5.3-2016.02-x86_64_arm-linux-gnueabi.tar.xz)). After decompressing the compressed package , enter the bin directory :

```
root@ubuntu:~/home/liuxin/gcc-linaro-5.3-2016.02-x86_64_arm-linux-gnueabi/bin# ls
arm-linux-gnueabi-addr2line  arm-linux-gnueabi-elfedit  arm-linux-gnueabi-gcc-ranlib  arm-linux-gnueabi-ld  arm-linux-gnueabi-readelf
arm-linux-gnueabi-ar        arm-linux-gnueabi-gcov     arm-linux-gnueabi-gcov-tool  arm-linux-gnueabi-ld.bfd  arm-linux-gnueabi-size
arm-linux-gnueabi-as        arm-linux-gnueabi-gcc      arm-linux-gnueabi-gfortran   arm-linux-gnueabi-ld.gold  arm-linux-gnueabi-strings
arm-linux-gnueabi-c++filt   arm-linux-gnueabi-gcc-ar   arm-linux-gnueabi-gprof      arm-linux-gnueabi-objcopy  arm-linux-gnueabi-strip
arm-linux-gnueabi-cpp       arm-linux-gnueabi-gcc-nm   arm-linux-gnueabi-ranlib     arm-linux-gnueabi-objdump  gdbserver
arm-linux-gnueabi-gdb       arm-linux-gnueabi-gdbserver
```

You can see , gcc-Linar O provides what we need gcc , gdb and gdbserver, they are " supporting " the .

Then the PC on download and cross compiler :

- `git clone https://gitlab.com/study-kernel/debugging_techniques/gdb.git` (https://gitlab.com/study-kernel/debugging_techniques/gdb.git) (https://gitlab.com/study-kernel/debugging_techniques/gdb.git)
- `export CROSS_COMPILE=/path/to/ gcc-linaro-5.3-2016.02-x86_64_arm-linux-gnueabi/bin / arm-linux-gnueabi-`
- `cd gdb && make`
- compiled out of the main and compressed inside gdbserver get to the BBB of the board

Then run `./gdbserver` — help on the BBB board , and you can roughly know how to use gdbserver by looking at it .

Here is an example of network communication :

```
root@embest:~# ./gdbserver 192.168.2.110:1234 main
Process main created; pid = 1781
Listening on port 1234
```

192.168.2.110 is the PC 's IP, port number 1234 free to take .

Then the PC on the run `arm-linux-gnueabi-gdb` and target remote command :

```
root@ubuntu:~/home/liuxin/gcc-linaro-5.3-2016.02-x86_64_arm-linux-gnueabi/bin# ./arm-linux-gnueabi-gdb
GNU gdb (Linaro GDB 2016.02) 7.10.1.20160210-cvs
Copyright (C) 2015 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 "--host=x86_64-unknown-linux-gnu --target=arm-linux-gnueabi".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://bugs.launchpad.net/gcc-linaro>.
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".
(gdb) target remote 192.168.2.105:1234
Remote debugging using 192.168.2.105:1234
Reading /root/main from remote target...
warning: File transfers from remote targets can be slow. use "set sysroot" to access files locally instead.
Reading /root/main from remote target...
Reading symbols from target:/root/main...done.
Reading /lib/ld-linux-armhf.so.3 from remote target...
Reading /lib/ld-linux-armhf.so.3 from remote target...
Reading symbols from target:/lib/ld-linux-armhf.so.3...Reading /lib/867309c1cc14ae392683da6c69aa2e46749280.debug from remote target...
Reading /lib/.debug/867309c1cc14ae392683da6c69aa2e46749280.debug from remote target...
(no debugging symbols found)...done.
0xb6fd7980 in ?? () from target:/lib/ld-linux-armhf.so.3
```

192.168.2.105 is the IP of the board, and the port number must be consistent with the port number specified by gdbserver .

At this point , the PC and the board have established remote debugging through the network . At this time , the gdbserver on the board will print the following sentence :

```
root@embest:~# ./gdbserver 192.168.2.110:1234 main
Process main created; pid = 1781
Listening on port 1234
Remote debugging from host 192.168.2.110
```

Finally , you can run list , continue and other commands on the PC to debug the remote program .

5.2 kdb & kgdb

[Write in front of , behind the new discovery may delete this paragraph]

In their experiments the process , found kdb and kgdb in debugging kernel terms not very obvious advantages , the main reason is to enter debug state must stop the kernel is running , but once the kernel is after stop , want to continue to continue to run , seemingly not succeed .

Currently the only conceivable scenario is useful when running a kernel crash , for example, appear oops / fault, at enabling kdb / kgdb of the case , the kernel will automatically enter kdb debugging state , at this time you can view the memory, register, call stack, etc. .

Maybe there are more useful scenarios that I didn't find . I will encounter actual problems later. I will consider whether it can be solved with kdb/kgdb.

[END]

gdb is for debugging an application , corresponding to the kdb and kgdb are used to debug the kernel .

kdb similar " local mode GDB", only target the machine can . By can debug commands using the serial port or a keyboard that provides , these commands include the investigation change memory, registers , print the call stack , setting breakpoints and so on .

kgdb is similar to " Remote Mode of gdb " , need target and HOST two machines . kgdb running on target of the kernel , host (e.g. the PC) running gdb command . TARGET and the host through the serial or network communication .

And kdb comparison , kgdb advantage is that you can provide source-level debugging . In addition kgdb in part also run kdb of command .

At present, kdb and kgdb have been merged to the main line of the kernel . For the history of merge, please refer to https://kgdb.wiki.kernel.org/index.php/Main_Page#Quick_History.

(https://kgdb.wiki.kernel.org/index.php/Main_Page#Quick_History) .

But the mainline kernel only supported via the serial port connection gdb and kgdb, because the author felt some problems through the network , refer to : [What is the status of kgdboe and the mainline](https://kgdb.wiki.kernel.org/index.php/KDB_FAQ#What_is_the_status_of_kgdboe_and_the_mainline.3F) (https://kgdb.wiki.kernel.org/index.php/KDB_FAQ#What_is_the_status_of_kgdboe_and_the_mainline.3F) . If you want to try your own network of communication , may refer to " RefLink " in third party methods provided . (https://kgdb.wiki.kernel.org/index.php/KDB_FAQ#What_is_the_status_of_kgdboe_and_the_mainline.3F)

5.2.1 Configure the kernel to enable kdb&kgdb

Although kdb and kgdb are two debugging methods , they can be switched to each other , and KGDB must be enabled to enable KDB . Therefore , it is recommended to enable both kdb and kgdb in the kernel , as follows :

CONFIG_KGDB=y	Join KGDB support
CONFIG_KGDB_SERIAL_CONSOLE=y	Make KGDB communicate with the host through the serial port (open this option, the default will open CONFIG_CONSOLE_POLL and CONFIG_MAGIC_SYSRQ)
CONFIG_KGDB_KDB=y	Join KDB support
CONFIG_DEBUG_RODATA=n	Turn this off, you can set a breakpoint in the read-only area
CONFIG_FRAME_POINTER=y	Enable KDB to print more stack information
CONFIG_KALLSYMS	Add symbol information
CONFIG_KDB_KEYBOARD=y	If you communicate with KDB through the keyboard of the target version, you need to turn on this, and the keyboard cannot be a USB interface

CONFIG_DEBUG _KERNEL=y	Contains driver debugging information
CONFIG_DEBUG _INFO=y	Make the kernel contain basic debugging information

In addition , BBB 's board should have enabled the options above .

5.2.2 Open and use kdb

kdb you can by bootargs turn , also can start after the completion of the kernel sysfs open .

5.2.2.1 Open via bootargs

It may be in bootargs which add the following contents :

kgdboc=[kms][[,]kbd][[,]serial_device][,baud]

- kgdboc of meaning "kgdb over console"
- kms = Kernel Mode Setting , the main purpose is to set the display system as a temporary console. Generally, you don't use it , so do n't pay attention to it .
- kbd = Keyboard
- serial_device and baud represent the serial port and baud rate to be used

For example , we configured this on the BBB board : kgdboc=ttyS0,115200

There is another parameter to explain : **kgdbwait**

It means that after the kernel completes some necessary initializations , it enters the kdb debugging state , which can be used to debug situations where the kernel cannot start normally . Typical usage is : bootargs= XXX kgdboc=ttyS0,115200 kgdbwait

When we BBB of the board using kgdboc and kgdbwait time , you can see the following output :

```
[ 2.828176] console [ttyS0] enabled
[ 2.832914] KGDB: Registered I/O driver kgdboc
[ 2.847988] KGDB: waiting for connection from remote gdb...

Entering kdb (current=0xdd0d4000, pid 1) on processor 0 due to keyboard Entry
[0]kdb> █
```

At this point you can use kdb debugging a , input help can query kdb various providers of command .

5.2.2.2 Open via sysfs

After the kernel is started , kdb can be enabled / disabled in the following ways :

Enable kgdboc on ttyS0 : echo ttyS0> /sys/module/kgdboc/parameters/kgdboc

Disable kgdboc : echo ""> /sys/module/kgdboc/parameters/kgdboc

NOTE: At this point do not need to specify the baud rate , since the kernel boot process has been configured well .

5.2.2.3 stop kernel

kdb / kgdb after opening to let the kernel debugger to stop running to enter the state , there are several ways to stop the kernel : Use kgdbwait AS A the Boot argument , using sysrq-G , or run the kernel until the exception, for example, oops or fault.

The way through sysrq -g is as follows :

echo g> /proc/sysrq-trigger

The running effects on the BBB board are as follows :

```
root@embest:~# echo g > /proc/sysrq-trigger
[ 678.651070] sysrq: SysRq : DEBUG
Entering kdb (current=0xdd63f480, pid 491) on processor 0 due to Keyboard Entry
[0]kdb> █
```

5.2.3 Turn on and use k g db

Open kgdb required to the target on entering kdb debug state , and then enter the command " kgdb " switching to kgdb debug mode :

```
[0]kdb> kgdb
Entering please attach debugger or use $D#44+ or $3#33
```

Note: Enter "\$3#33" to switch from kgdb mode to kdb mode .

The next step requires the host on the run gdb and through the serial port with the target to establish the connection , as follows :

```
arm-linux-gnueabi-gdb ./vmlinux
(gdb) target remote /dev/ttyUSB0
```

After the connection is established , you can debug the kernel like a normal application .

5.2.4 R efLink

Official wiki: https://kgdb.wiki.kernel.org/index.php/Main_Page
(https://kgdb.wiki.kernel.org/index.php/Main_Page)

Official document : <https://www.kernel.org/pub/linux/kernel/people/jwessel/kdb/>
(<https://www.kernel.org/pub/linux/kernel/people/jwessel/kdb/>)

- [Running kdb commands from gdb](https://www.kernel.org/pub/linux/kernel/people/jwessel/kdb/ch06s02.html)
(<https://www.kernel.org/pub/linux/kernel/people/jwessel/kdb/ch06s02.html>)
(<https://www.kernel.org/pub/linux/kernel/people/jwessel/kdb/ch06s02.html>)
- The [kgdbcon](https://www.kernel.org/pub/linux/kernel/people/jwessel/kdb/kgdbcon.html) (<https://www.kernel.org/pub/linux/kernel/people/jwessel/kdb/kgdbcon.html>) feature allows you to see printk() messages inside gdb while gdb is connected to the kernel
(<https://www.kernel.org/pub/linux/kernel/people/jwessel/kdb/kgdbcon.html>)

kgdboe : <http://sysprogs.com/VisualKernel/kgdboe/> (<http://sysprogs.com/VisualKernel/kgdboe/>)

Example of using kgdb on samsung : <http://www.cnblogs.com/Ph-one/p/6432717.html>
(<http://www.cnblogs.com/Ph-one/p/6432717.html>) (<http://www.cnblogs.com/Ph-one/p/6432717.html>)

6 LLDB

LLDB is llvm new generation debugger , to replace gdb, its official website : <http://lldb.llvm.org/> (http://lldb.llvm.org/).

For a comparison of the functions provided by LLDB and GDB , see : <http://lldb.llvm.org/lldb-gdb.html> (http://lldb.llvm.org/lldb-gdb.html).

For how to use LLDB, please refer to the official tutorial : <http://lldb.llvm.org/tutorial.html> (http://lldb.llvm.org/tutorial.html)

7 probe technology

Probe technology is based on the original code does not change , prying running status code , it is like a multimeter check circuit the same .

7.1 Kprobes

Kprobes technology is used in the kernel layer .

7.1.1 Introduction

kprobe is a tool that dynamically collects debugging and performance information. It is derived from the Dprobe project. It is a non-destructive tool. Users can use it to track almost any function or executed instruction as well as some asynchronous events (such as timer) . Its basic workflow is: the user specifies a detection point and associates a user-defined processing function to the detection point. When the kernel executes to the detection point, the corresponding associated function is executed, and then continues to execute the normal code path .

Kprobe implements three types of probe points : kprobes, jprobes and kretprobes (also called return probe points) .

- ✓ kprobes are probe points that can be inserted into any instruction position of the kernel
- ✓ jprobes can only be inserted into the entrance of a kernel function
- ✓ kretprobes is executed when the specified kernel function returns

Generally, the kprobe program is used as a kernel module. The initialization function of the module is responsible for installing detection points, and the exit function uninstalls those installed detection points. kprobe provides interface functions (APIs) to install or uninstall probe points.

7.1.2 Principle

Kprobes

The essence of kprobes is to use the breakpoint instruction of the CPU : 386 and x86_64 have a special breakpoint instruction int3, when the instruction is executed, a breakpoint interrupt is triggered to execute the corresponding processing function ; the ARM architecture does not have a special breakpoint instruction , So use undefined instructions to simulate breakpoint instructions .

When installing a kprobes probe point, kprobe first backs up the probed instruction, and then uses the breakpoint instruction (that is , the int3 instruction in i386 and x86_64) to replace the first one or several bytes of the probed instruction. When the CPU executes the detection point , it will execute the trap operation due to running the breakpoint instruction , which will save the CPU register and call the corresponding trap processing function , and the trap processing function will call the corresponding notifier_call_chain (an asynchronous work in the kernel) For all the notifier functions registered in the mechanism) , kprobe realizes the detection processing by registering the processing function associated with the detection point to the notifier_call_chain corresponding to the trap .

When kprobe registration notifier that, when executed, it first performs associated detection point pre_handler function, and the corresponding kprobe struct and registers saved as a parameter of the function, then, kprobe set CPU as a single-step mode , and the single-step the sniffer backup , single-step mode causes the CPU to run a complete command after the shot breakpoint interrupt , the interrupt handler , kprobe execution post_handler .

After all these operations are completed , the CPU operating mode is restored , and then the instruction stream following the detected instruction will be executed normally.

In the ARM architecture in , is replaced with an undefined instruction INT3, when executed, to the undefined instruction , triggers the corresponding interrupt , the corresponding interrupt handler is do_undefinstr (http://lxr.free-electrons.com/ident?v=4.4;i=do_undefinstr) , do_undefinstr further call call_undef_hook , then call kprobes_arm_break_hook-> Fn , then execution kprobe_handler .

Jprobes

jprobe can seamlessly access the parameters of the probed function . jprobe handler should be detected and functions have the same prototype , the function and the processing function end must call kprobe functions provided jprobe_return () .

jprobe dependent kprobe, equivalent to using kprobe set a breakpoint , the breakpoint location is a function of the entry detection , when the implementation of the break point when, kprobe backup CPU portions of the stack and registers, and instruction pointer modify jprobe handler, When the jprobe processing function is executed , the register and stack contents are exactly the same as the actual probed function, so it can access the function parameters without any special processing. When the processing function is executed to the end, it calls jprobe_return() , That will cause the registers and stack to be restored to the state when the probe point was executed, so that the probed function can be run normally. It should be noted that the parameters of the probed function may be passed through the stack or through registers, but jprobe can work in both cases because it backs up both the stack and the registers. Of course, the premise is that jprobe must process the function prototype. Exactly the same as the detected function.

Kretprobe s

Kretprobe also uses kprobes to implement. When the user calls register_kretprobe() , kprobe establishes a detection point at the entry of the probed function. When the detection point is reached, kprobe saves the return address of the probed function and replaces the return address as The address of a trampoline , kprobe defines the trampoline when it is initialized and registers a kprobe for the trampoline . When the probed function executes its return instruction, control is passed to the trampoline , so kprobe has registered the processing function corresponding to the trampoline Will be executed, and the processing function will call the processing function associated with the kretprobe by the user . After the processing is completed, the instruction register is set to point to the function return address that has been backed up, so the original function return is executed normally.

The return address of the probed function is stored in a variable of type `kretprobe_instance` . The `maxactive` field of the structure `kretprobe` specifies the number of instances of the probed function that can be simultaneously probed. The function `register_kretprobe()` will pre-allocate a specified number of `kretprobe_instance` . If the probed function is non-recursive and the spinlock has been held during the call , then `maxactive` is 1 is sufficient; if the probed function is non-recursive and preemption is invalid at runtime, then `maxactive` is `NR_CPUS` . ; If `maxactive` is set to be less than or equal to 0, it is set to the default value (if preemption is enabled, `CONFIG_PREEMPT` is configured , the default value is the maximum of 10 and $2*NR_CPUS$, otherwise the default value is `NR_CPUS`).

If `maxactive` is set too small, the execution of some detection points may be lost, but it does not affect the normal operation of the system . The `nmissed` field in the structure `kretprobe` will record the number of executions of the lost detection point, which is registered when the return detection point is registered Set to 0 , it will increase by 1 every time when the probe function is executed and no `kretprobe_instance` is available .

7.1.3 API

Kprobes

Header file : `include/ linux/kprobes.h`

struct kprobe	Comment
<code>struct hlist_node hlist</code>	All registered kprobes will be added to the <code>kprobe_table</code> hash table, and <code>hlist</code> members are used to link to a certain slot
<code>struct list_head list</code>	If multiple kprobes are registered at the same location , these kprobes will form a queue . The head of the queue is a special kprobe instance , and the list members are used to link to this queue . When the probe point is triggered , the kprobe at the head of the queue The handlers registered in the instance will traverse the handlers registered in the queue one by one
<code>unsigned long nmissed</code>	Record the number of times the current probe has not been processed
<code>kprobe_opcode_t *addr</code>	This member has two functions : One is that the user specifies the base address of the detection point before registering (plus the offset to get the real address), the base address can be queried in <code>System.map</code> through the symbolic name . The other is to save the actual address of the detection point after registration . Before registration , this can not be specified , by the kprobes to initialize . If not specified , you must specify the location of the probe symbol information (<code>symbol_name</code>), for example, the function name
<code>const char *symbol_name</code>	Symbol Name probing point . From the <code>System.map</code> get into the symbolic name . The name and address cannot be specified at the same time , otherwise an <code>EINVAL</code> error will be returned during registration .
<code>unsigned int offset</code>	The offset of the detection point relative to the <code>addr</code> address . <code>addr/symbol_name</code> can only be accurate to the function name . With offset, you can set a breakpoint anywhere inside the function
<code>kprobe_pre_handler_t pre_handler</code>	This interface after the break exception trigger , is invoked before the start of the original single-step instructions

kprobe_post_handler_t post_handler	Will be called after stepping through the original instruction
kprobe_fault_handler_t fault_handler	Specify an error handler , when performing pre_handler , post_handler when an error occurs and is detected during the function , it will be called
kprobe_break_handler_t break_handler	<p>In the call to probe handler (such as pre_handler Interface) triggering an exception breakpoint when invoking the interface .</p> <p>Break exception is handled by interrupt gate , automatically before calling the appropriate handler for the interrupt disabling . Interrupt though not received maskable interrupt off the case , but the CPU thrown exception or NMI or will receive , so It is possible that nesting of breakpoint exception handling may occur , which is used in the implementation of jprobes .</p>
kprobe_opcode_t opcode	Original instruction , saved before being replaced by breakpoint instruction (int 3 instruction under X86)
struct arch_specific_insn ainsn	A copy of the original instructions of the probe point is saved . The instructions copied here are more instructions than those stored in the opcode , and the size of the copy is MAX_INSN_SIZE * sizeof(kprobe_opcode_t)
u32 flags	<p>Flag detection point , preferably the value KPROBE_FLAG_GONE and KPROBE_FLAG_DISABLED .</p> <p>If the KPROBE_FLAG_GONE flag is set , it means that the breakpoint instruction has been removed ;</p> <p>If KPROBE_FLAG_DISABLED is set , it means that only the probe is registered , but it is not enabled , which means that the interface of the probe will not be called when the breakpoint is triggered.</p>

API

API kprobe	Comment
int register_kprobe(struct kprobe *p)	<p>The parameter of this function is a pointer of type struct kprobe</p> <p>Before calling this function is registered , the user must first set the struct kprobe respective fields , kprobe of 3 treatments do not have to function all the definitions , any processing function can be set to NULL</p> <p>This function returns 0 when it succeeds , otherwise it returns a negative error code</p>
void unregister_kprobe(struct kprobe *p)	Logout breakpoint
int register_kprobes(struct kprobe **kps, int num)	Register multiple kprobes at once
void unregister_kprobes(struct kprobe **kps, int num)	Log out multiple kprobes at once

Jprobes

Header file : include/ linux/kprobes.h

struct jprobe	Comment
struct kprobe kp	Kprobe introduced above

void *entry	<p>Registered jprobe before , we need to define a struct jprobe type variable and set its kp.addr and entry fields .</p> <p>kp.addr specifies the location of the probe point , it must be the address of the first instruction of the probed function .</p> <p>entry specifies the processing function of the probe point. The parameter list and return type of the processing function should be exactly the same as the probed function , and it must call jprobe_return() just before returning . If the probed function is declared as asm linkage ,fastcall or influence parameters any other form of transmission , then the corresponding handler must also be declared with corresponding form .</p>
-------------	--

API

API jprobe	Comment
int register_jprobe(struct jprobe *p)	<p>The registration function registers a jprobes type detection point in jp->kp.addr . When the kernel runs to the detection point , the function specified by jp->entry will be executed .</p> <p>If successful , the function returns 0 , otherwise it returns a negative error code .</p>
void unregister_jprobe(struct jprobe *p)	Log out of jprobe
int register_jprobes(struct jprobe **jps, int num)	Register multiple jprobes
void unregister_jprobes(struct jprobe **jps, int num)	Unregister multiple jprobes

Kretprobes

Header file : include/ linux/kprobes.h

struct kretprobe	Comment
struct kprobe kp	Kprobe introduced above
kretprobe_handler_t handler	<p>Before registering kretprobe, the user must define a variable of struct kretprobe and set its kp.addr , handler and maxactive fields .</p> <p>kp.addr specifies the location of the detection point</p> <p>When the probed function returns , the handler function will be executed</p>
kretprobe_handler_t entry_handler	When the entry is detected to perform the function , it executes entry_handler function
int maxactive	maxactive specifies the maximum number of processing function instances that can run at the same time , it should be set appropriately , otherwise some operations of the detection point may be lost . For details, see " 4.10.2 Kretprobes "
int nmissd	Record the number of executions of lost detection points
size_t data_size	
struct hlist_head free_instances	
raw_spinlock_t lock	

struct kretprobe_instance	Comment
struct hlist_node hlist	

struct kretprobe *rp	Point to the corresponding kretprobe
kprobe_opcode_t *ret_addr	Indicates the return address
struct task_struct *task	
char data[0]	

API

API Kretprobe	Comment
int register_kretprobe(struct kretprobe *rp)	If successful , the function returns 0 , otherwise it returns a negative error code .

7.1.4 Features and limitations

Kprobe allows you to register multiple kprobes at the same address , but you cannot have multiple jprobes on the address at the same time .

Generally , users can register probe points at any location in the kernel , especially for interrupt handling functions , but there are some exceptions . If users try to implement kprobe code (including kernel/kprobes.c and arch/*/kernel /kprobes.c and do_page_fault and notifier_call_chain) to register the probe point , register_*probe will return -EINVAL.

If you register a probe point for an inline function , kprobe cannot guarantee that all instances of the function will register the probe point , because gcc may implicitly inline a function . Therefore , keep in mind that the user may not see the expected Execution of detection points .

A detection point can modify the context handler function be detected , such as to modify the kernel data structures , registers and the like . Thus , the kprobe be used to mount bug resolution code injection or some errors or test code .

If a function calls another detection process detection points , the processing function of the detection point will not run , but it missed number will be added to a . A plurality of multiple instances of the same function or probe point processing functions capable of processing different CPU Run at the same time .

Except for registration and uninstallation , kprobe will not use mutexe or allocate memory .

The detection point processing function is invalid and preempt at runtime . Depending on the specific architecture , the detection point processing function may also be interrupted when it is running . Therefore , for any detection point processing function , do not use any kernel that causes sleep or process scheduling. Function (such as trying to get a semaphore)

kretprobe is replaced by the return address for a predefined trampoline address to achieve , and therefore the stack back and gcc built-in functions __builtin_return_address () call will return trampoline address instead of the real detected function's return address .

If the number of calls to a function is not the same as the number of returns , then the kretprobe probe point registered on the function may produce unexpected results (do_exit() is a typical example , but do_execve() and do_fork() are no problem)

When entering or exiting a function , if the CPU is running on a stack that is not owned by the current task , the kretprobe detection of the function may produce unexpected results , so kprobe does not support the return detection of `__switch_to()` on `x86_64` . , If the user registers a detection point to it , the registration function will return `-EINVAL` .

7.1.5 Example

https://gitlab.com/study-kernel/debugging_techniques/kprobes
kernel/debugging_techniques/kprobes)

(https://gitlab.com/study-kernel/debugging_techniques/kprobes)

The README has instructions for using examples .

7.2 Ptrace

Ptrace technology is used in user space .

7.2.1 Introduction

`ptrace()` is a system call . Its core is to allow one process (tracking process) to modify the memory and registers of another process (tracked process) , thereby changing the behavior of the tracked process .

Note : `ptrace()` is highly dependent on the underlying hardware . Programs that use `ptrace` are usually not easy to port between clock architectures .

7.2.2 API

Android Bionic in , the `ptrace` following prototype (the Linux under `glibc` have similar implementation , `ptrace` ordinary linux can also be used on the system) : http://androidxref.com/7.1.1_r6/xref/bionic/libc/bionic/ptrace.cpp (http://androidxref.com/7.1.1_r6/xref/bionic/libc/bionic/ptrace.cpp)

```
#include <sys/ptrace.h>
extern "C" long __ptrace ( int req , pid_t pid , void * addr , void * data );
```

We can see that `ptrace` has 4 parameters :

- `req` decides what `ptrace` does
- `pid` is the ID of the tracked process
- `data` stores the data that will be read / written from the place where the process space offset is `addr`

The most important thing here is the `req` parameter . Let 's take a look at what values `req` can take and the corresponding meaning : <https://linux.die.net/man/2/ptrace> (<https://linux.die.net/man/2/ptrace>)

[Note] I planned to use a table to explain the meaning of each `req` parameter in detail , but since I haven't actually tried `ptrace` , I won't give too much explanation here . The details can be parameterized in the above link . You can add more explanations if you need it later .

7.2.3 Function hook

Function hook of a specific scenario is : assuming that there is a process A, A call to the libc. SO this library provides the func1 function . Suppose I wanted to modify func1 is achieved , the how to do ? Which way is to directly modify libc in func1 of the code , then recompile libc.so, so do harm will change , " the official Code ." Another one way is to ourselves to achieve a function func2, the function compiledlibx.so, then using Pt Race technology , the during operation of the process A in the func1 replaced libx.so of func2.

To achieve the above scenario , it requires two steps :

The first step , need to libx.so injected into the process A 's address space . Reference : <http://www.cnblogs.com/jiayy/p/4283766.html> (http://www.cnblogs.com/jiayy/p/4283766.html)

The first step two , need to acquire func1 in process A in the memory address , and then using ptrace modify this address , the func1 replaced as func2, reference : <http://www.cnblogs.com/jiayy/p/4283990.html> (http://www.cnblogs.com/jiayy/p/4283990.html)

[Note] Because there is no actual hands tried ptrace, therefore here is not to do too much explanation , details can be parameterized link above . Follow-up need may supplement .

7.3 Watchpoint

Breakpoints , divided into software breakpoints and hardware breakpoints .

The principle of software breakpoint is generally to modify the RAM, first back up the monitored address , and then replace the content of the address with a certain characteristic value . When the code runs to this point, an exception will be triggered , and the content of the backup before running in the exception handling .

Hardware breakpoints rely on specific hardware CPU, the principle is generally to CPU of a register to be written is monitored address , CPU is compared with the register during operation , when an equal address triggers an exception . In the process, we can make an exception Things you want to do , such as dump stack to see who accesses this address again , this feature can be used to debug memory stepping problems .

7.3.1 Software breakpoints and hardware breakpoints

Here is a detailed introduction to their differences and advantages and disadvantages .

http://www.360doc.com/content/17/0823/11/17136639_681458868.shtml

(http://www.360doc.com/content/17/0823/11/17136639_681458868.shtml)

Another

link:

http://processors.wiki.ti.com/index.php/Data_Breakpoint/Watchpoint#Difference_between_Software_and_Hardware_Breakpoints

(http://processors.wiki.ti.com/index.php/Data_Breakpoint/Watchpoint#Difference_between_Software_and_Hardware_Breakpoints)

7.3.2 Realization of hardware breakpoints

First of all , the hardware needs to support the hardware breakpoint function , generally x86, x64, ARM64, ARM (like A8, A9, A15 and later) are all supported .

Secondly , the kernel in 2.6.37 After adding a hardware breakpoint subsystem (https://blog.csdn.net/_xiao/article/details/40619797) :

- pairs , providing register_user_hw_breakpoint (https://elixir.bootlin.com/linux/v5.0.2/source/kernel/events/hw_breakpoint.c#L454) and register_wide_hw_breakpoint (https://elixir.bootlin.com/linux/v5.0.2/source/kernel/events/hw_breakpoint.c#L554) , respectively, for the user space and kernel space to set hardware breakpoints .
(https://elixir.bootlin.com/linux/v5.0.2/source/kernel/events/hw_breakpoint.c#L454)
(https://elixir.bootlin.com/linux/v5.0.2/source/kernel/events/hw_breakpoint.c#L554)
- Next , arch/xxx/kernel/ hw_breakpoint.c implements the method of accessing the hardware registers on each architecture .

Finally , from the perspective of use :

- samples/hw_breakpoint/ data_breakpoint.c (https://elixir.bootlin.com/linux/v5.0.2/source/samples/hw_breakpoint/data_breakpoint.c) shows how to set hardware breakpoints in the kernel .
(https://elixir.bootlin.com/linux/v5.0.2/source/samples/hw_breakpoint/data_breakpoint.c)
- Arch / xxx / Kernel / ptrace.c provides an interface to user space , so that we can set hardware breakpoints in user space . (Later demo shows how to pass in user space ptrace set hardware breakpoints).

When the breakpoint is set successfully , once the conditions are matched , the hardware will generate an exception , the kernel will generate an interrupt , and then call back the registered breakpoint processing function in the interrupt processing function .

At the kernel level , we can do what we want in the breakpoint processing function .

How does the user space know that an exception has occurred ? The breakpoint processing function will send a SIGTRAP signal (such as ptrace_hbptriggered (<https://elixir.bootlin.com/linux/v5.0.2/source/arch/arm64/kernel/ptrace.c#L181>)) to the process , and the user process can capture this signal .

The implementation of each system architecture is slightly different , but the principle is the same , see the hardware breakpoint subsystem for (https://blog.csdn.net/_xiao/article/details/40619797) details .

7.3.3 D emo

Please read the README document to learn how to compile and use the Demo.

https://gitlab.com/study-android/memory_related/ptrace_hw_watchpoint (https://gitlab.com/study-android/memory_related/ptrace_hw_watchpoint)

8 Tracer & performance analysis

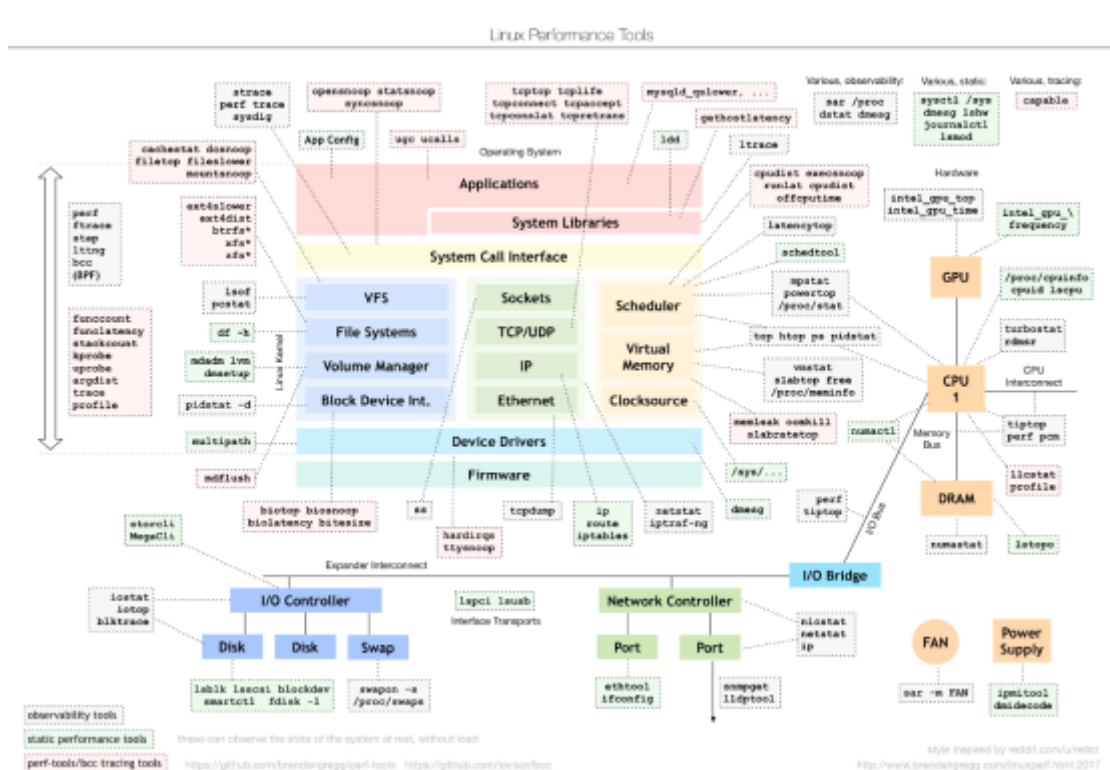
8.1 Tracer Total said

If you want to down-tune the kernel in Linux and capture the performance of the program , then the first thing that comes to mind may be OProfile and Linux Perf. But obviously , open source has a very significant feature that you cannot avoid , that is, you have too many choices. : perf, oprofile, systemtap, dtrace4linux, lttng, kgtp, ktap, sysdig,

ftrace, eBPF. Is it already dazzled ? Then you can't miss this article :

<http://www.brendangregg.com/blog/2015-07-08/choosing-a-linux-tracer.html>
 (http://www.brendangregg.com/blog/2015-07-08/choosing-a-linux-tracer.html)
 The author carefully lists a large number of tool principles and usage tutorials

8.2 Performance Analysis Total said



<http://www.brendangregg.com/linuxperf.html> (http://www.brendangregg.com/linuxperf.html) , the figure is very comprehensive display of the L inux each module can be used analysis tool , click on the web site for details . (Brendan Gregg is Netflix Senior Performance Architect, he where do large-scale computer performance design, analysis and tuning addition, he is. " Systems performance author" and other technology books, has won the 2013 Nian USENIX LISA Award! his github address : <https://github.com/brendangregg> (https://github.com/brendangregg)) .

8.3 strace

strace is mainly used during the process of implementation of systemt calls and the received signal . More and more instructions please refer to : <http://www.cnblogs.com/ggjucheng/archive/2012/01/08/2316692.html> (http://www.cnblogs.com/ggjucheng/archive/2012/01/08/2316692.html)

8.4 ftrace

ftrace is mainly used to track the runtime behavior of the kernel , including the following functions :

- Function tracer and Function graph tracer: trace function calls.
- Schedule switch tracer: track the process scheduling situation.
- Wakeup tracer : Track the scheduling delay of the process, that is , the delay time from the high-priority process from entering the ready state to obtaining the CPU . The tracer is only for real-time processes.

- **Irqsoff tracer** : When the interrupt is disabled, the system cannot respond to external events, such as keyboard and mouse, and the clock cannot generate tick interrupts. This means that the system response delay, irqsoff this tracer to track and record the kernel function which disables interrupts, for which the interrupt disable time is the longest, irqsoff will log marked the first line of the file, so that developers can quickly Locate the culprit that caused the response delay.
- **Preemptoff tracer** : Similar to the previous tracer , the preemptoff tracer tracks and records the functions that prohibit kernel preemption, and clearly shows the kernel functions that have the longest time to prohibit preemption.
- **Preemptirqsoff tracer**: Same as above, to track and record the kernel functions that prohibit interrupts or preemption, and the functions that have been forbidden for the longest time.
- **Branch tracer**: Track the hit rate of likely/unlikely branch prediction in the kernel program . Branch tracer can record the number of successful predictions of these branch statements. So as to provide clues for the optimization program.
- **Hardware branch tracer** : Use the branch tracking capability of the processor to achieve hardware-level instruction jump recording. On x86 , the feature of BTS is mainly used .
- **Initcall tracer** : Record the init call called by the system in the boot phase .
- **Mmiotrace tracer** : record related information of memory map IO .
- **Power tracer** : Record system power management related information.
- **Sysprof tracer** : By default, sysprof tracer samples the kernel every 1 msec and records function calls and stack information.
- **Kernel memory tracer**: The memory tracer is mainly used to track the allocation of slab allocator . Including kfree , kmem_cache_alloc and other API calls, the user program can analyze the internal fragmentation based on the information collected by the tracer , find out the code fragments with the most frequent memory allocation, and so on.
- **Workqueue statistical tracer** : This is a statistic tracer , which counts the work status of all workqueues in the system , such as how many work has been inserted into the workqueue , and how many have been executed. Developers can use this to determine the specific workqueue implementation, such as whether to use single threaded workqueue or per cpu workqueue.
- **Event tracer**: trace system events, such as timer , system call, interrupt, etc.


Not all tracers are listed here , ftrace is currently a very active development area, and new tracers will continue to be added to the kernel.

[ftrace Introduction](https://www.ibm.com/developerworks/cn/linux/l-cn-ftrace/) (https://www.ibm.com/developerworks/cn/linux/l-cn-ftrace/) describes the ftrace main function, use, basic principles .

[Use ftrace debug Linux kernel](https://www.ibm.com/developerworks/cn/linux/l-cn-ftrace1/) (https://www.ibm.com/developerworks/cn/linux/l-cn-ftrace1/) is mainly from menuconfig 's point of view about how to configure the kernel to enable ftrace.

9 Drive Design Guidelines

9.1 printk

- menuconfig in , **CONFIG_MESSAGE_LOGLEVEL_DEFAULT** for configuration does not display indicates printk of loglevel time , the default loglevel is how much
- menuconfig in , **CONFIG_PRINTK_TIME** to configure whether to enable time display , may likewise be / sys / module / printk / parameters / time modified .
-  **printk_ratelimit** API can be used to control the printing speed , and /proc/sys/kernel/printk_ratelimit and /proc/sys/kernel/printk_ratelimit_burst can control the behavior of printk_ratelimit .
- menuconfig in , **CONFIG_LOG_BUF_SHIFT** for configuring log_buf size .

No matter what the loglevel is , the data will be stored in log_buf

- the bootargs inside , by " console = " to **specify printk printed information which is sent to console on**
If " console= " is not specified , it will be printed on the first registered console by default
- can be modified through the / SYS / Module / printk / the Parameters / ignore_loglevel , to ignore loglevel, so **no matter what loglevel, can display information to the console on** .
bootargs inside , can also " ignore_loglevel " to achieve a similar effect .
- by / proc / sys / kernel / printk modified console_loglevel value , thereby controlling which loglevel information may be displayed console on
It may likewise be bootargs the " quiet / Debug; LogLevel = " modify console_loglevel value .
You can also modify the value of console_loglevel through dmesg -n xx .
- be in user space by dmesg read command log_buf data , by default , it will read log_buf all data , not loglevel filtered .

9.2 early_printk

- Open CONFIG_EARLY_PRINTK and CONFIG_DEBUG_LL in menuconfig
- In bootargs passing a parameter : bootargs = "... . **Earlyprintk** ";
- assembly language functions : addruart , waituart , senduart , busyuart these functions , using the UART print information

9.3 early_print

- In menuconfig in , open CONFIG_DEBUG_LL
- assembly language functions : addruart , waituart , senduart , busyuart these functions , using the UART print information