[内核日志及printk结构浅析](http://blog.chinaunix.net/uid-26993600-id-3252420.html) 2012-06-24 16:09:35

分类：

原文地址：[内核日志及printk结构浅析](http://blog.chinaunix.net/uid-20543672-id-3211832.html) 作者：[tekkamanninja](http://blog.chinaunix.net/uid/20543672.html)

    这段时间复习了一下内核调试系统，注意看了一下printk的实现以及内核日志的相关知识，这里做一下总结。

**一、printk概述**

    对于做Linux内核开发的人来说，printk实在是再熟悉不过了。内核启动时显示的各种信息大部分都是通过她来实现的，在做内核驱动调试的时候大部分时候使用她就足矣。她之所以用得如此广泛，一个是由于她使用方便，还有一个重要的原因是她的健壮性。它使用范围很广，几乎是内核的任何地方都能调用它。你既可以在中断上下文、进程上下中调用她，也可以在任何持有锁时调用她，更可以在SMP系统中调用她，且调用时连锁都不必使用。这样好的适应性来源于她的设计，一个由三个指针控制的简单“ring buffer”。

   **注意上面说到的是：“几乎”在内核的任何地方都可以使用。**那什么地方使用会有“问题”？**那就是在系统启动过程的早期，终端初始化之前的某些地方虽然可以使用，但是在终端和控制台被初始化之前所有信息都被缓存在printk的简单的ring buffer（环形缓冲区）中，直到终端和控制台被初始化之后，所有缓存信息都被一并输出。**

     如果你要调试的是启动过程最开始的部分（如setup\_arch()），可以依靠此时能够工作的硬件设备（如串口）与外界通信，使用printk()的变体early\_printk()函数。她在启动过程初期就具有在终端上打印的能力，功能与prink()类似，区别在于：

（1）函数名

（2）能够更早地工作（输出信息）

（3）她有自己的小缓存（一般为512B）

（4）一次性输出到硬件设备，不再以ring buffer的形式保留信息。

      但该函数在一些构架上无法实现，所以这种办法缺少可移植性。（大多数构架都可以，包括x86和arm）。

     所以，除非要在启动初期在终端上输出，否则我们认为printk()在任何情况下都能工作。这点从内核的启动代码中就可以看出，在已进入start\_kernel不久就通过printk打印内核版本信息了:**printk(KERN\_NOTICE "%s", linux\_banner);**。

**二、printk的使用**

   printk()和C库中的printf()在使用上最主要的区别就是 printk()指定了日志级别。

**（1）日志等级**

内核根据日志级别来判断是否在终端（console）上打印消息：内核把级别比某个特定值低的所有消息显示在终端（console）上。但是所有信息都会记录在printk的“ring buffer”中。

  printk有8个loglevel,定义在<linux/kernel.h>中：

1. #define KERN\_EMERG "<0>" /\* 系统不可使用 \*/
2. #define KERN\_ALERT "<1>" /\* 需要立即采取行动 \*/
3. #define KERN\_CRIT "<2>" /\* 严重情况 \*/
4. #define KERN\_ERR "<3>" /\* 错误情况 \*/
5. #define KERN\_WARNING "<4>" /\* 警告情况 \*/
6. #define KERN\_NOTICE "<5>" /\* 正常情况, 但是值得注意 \*/
7. #define KERN\_INFO "<6>" /\* 信息型消息 \*/
8. #define KERN\_DEBUG "<7>" /\* 调试级别的信息 \*/
9. /\* 使用默认内核日志级别 \*/
10. #define KERN\_DEFAULT "<d>"
11. /\*
12. \* 标注为一个“连续”的日志打印输出行(只能用于一个
13. \* 没有用 \n封闭的行之后). 只能用于启动初期的 core/arch 代码
14. \* (否则续行是非SMP的安全).
15. \*/
16. #define KERN\_CONT "<c>"

     如果使用时没有指定日志等级，内核会选用DEFAULT\_MESSAGE\_LOGLEVEL，这个定义位于kernel/printk.c：

1. /\* printk's without a loglevel use this.. \*/
2. #define DEFAULT\_MESSAGE\_LOGLEVEL CONFIG\_DEFAULT\_MESSAGE\_LOGLEVEL

    可以看出，这个等级是可以在内核配置时指定,这种机制是从2.6.39开始有的，如果你不去特别配置，那么默认为<4>,也就是KERN\_WARNING。

    内核将最重要的记录等级 KERN\_EMERG定为“<O>”，将无关紧要的调试记录等级“KERN\_DEBUG”定为“<7>”。

    内核用这些宏指定日志等级和当前终端的记录等级console\_loglevel来决定是不是向终端上打印，使用示例如下：

1. printk(KERN\_EMERG "log\_level:%s\n", KERN\_EMERG);

    当编译预处理完成之后，前例中的代码实际被编译成成如下格式：

1. printk( "<0>" "log\_level:%s\n", KERN\_EMERG);

     给一个printk()什么日志等级完全取决于你。那些正式、且需要保持的消息应该根据信息的性质给出相应的日志等级。但那些为了解决一个问题临时加得到处都是的调试信息可以有两种做法：

    一种选择是保持终端的默认记录等级不变，给所有调试信息KERN CRIT或更低的等级以保证信息一定会被输出。

    另一种方法则相反，给所有调试信息KERN DEBUG等级，而调整终端的默认记录等级为7，也可以输出所有调试信息。

    两种方法各有利弊。

    这里说到了调整内核的默认的日志级别，在我3年半前的学习笔记[《 Linux设备驱动程序学习（2）-调试技术 》](http://blog.chinaunix.net/uid-20543672-id-94292.html)中有介绍，可以通过 /proc/sys/kernel/printk来改变，或者C程序调用syslog系统调用来实现。但是现在的glibc的函数接口改了，由于 syslog 这个词使用过于广泛，这个函数的名称被修改成 klogctl，所以原来博文中的代码无法使用了，以下是新版的 console\_loglevel代码：

1. #include <stdio.h>
2. #include <stdlib.h>
3. #include <string.h>
4. #include <errno.h>
5. //#define \_\_LIBRARY\_\_ /\* \_syscall3 and friends are only available through this \*/
6. //#include <linux/unistd.h>
7. /\* define the system call, to override the library function \*/
8. //\_syscall3(int, syslog, int, type, char \*, bufp, int, len);
9. int main(int argc, char \*\*argv)
10. {
11. int level;
12. if (argc == 2) {
13. level = atoi(argv[1]); /\* the chosen console \*/
14. } else {
15. fprintf(stderr, "%s: need a single arg\n", argv[0]);
16. exit(1);
17. }
18. if (**klogctl**(8, NULL, level) < 0) {
19. fprintf(stderr, "%s: syslog(setlevel): %s\n",
20. argv[0], strerror(errno));
21. exit(1);
22. }
23. exit(0);
24. }

**（2）相关辅助宏**

    如果确定printk所需要的日志等级，每次都要在其中添加以上宏，似乎有点麻烦了。所以内核黑客们定义了一些宏来方便printk的使用，这些宏在内核代码中也是随处可见：

1. #ifndef pr\_fmt
2. #define pr\_fmt(fmt) fmt
3. #endif
4. #define pr\_emerg(fmt, ...) \
5. printk(KERN\_EMERG pr\_fmt(fmt), ##\_\_VA\_ARGS\_\_)
6. #define pr\_alert(fmt, ...) \
7. printk(KERN\_ALERT pr\_fmt(fmt), ##\_\_VA\_ARGS\_\_)
8. #define pr\_crit(fmt, ...) \
9. printk(KERN\_CRIT pr\_fmt(fmt), ##\_\_VA\_ARGS\_\_)
10. #define pr\_err(fmt, ...) \
11. printk(KERN\_ERR pr\_fmt(fmt), ##\_\_VA\_ARGS\_\_)
12. #define pr\_warning(fmt, ...) \
13. printk(KERN\_WARNING pr\_fmt(fmt), ##\_\_VA\_ARGS\_\_)
14. #define pr\_warn pr\_warning
15. #define pr\_notice(fmt, ...) \
16. printk(KERN\_NOTICE pr\_fmt(fmt), ##\_\_VA\_ARGS\_\_)
17. #define pr\_info(fmt, ...) \
18. printk(KERN\_INFO pr\_fmt(fmt), ##\_\_VA\_ARGS\_\_)
19. #define pr\_cont(fmt, ...) \
20. printk(KERN\_CONT fmt, ##\_\_VA\_ARGS\_\_)
21. /\* 除非定义了DEBUG ，否则pr\_devel()不产生任何代码 \*/
22. #ifdef DEBUG
23. #define pr\_devel(fmt, ...) \
24. printk(KERN\_DEBUG pr\_fmt(fmt), ##\_\_VA\_ARGS\_\_)
25. #else
26. #define pr\_devel(fmt, ...) \
27. no\_printk(KERN\_DEBUG pr\_fmt(fmt), ##\_\_VA\_ARGS\_\_)
28. #endif
29. /\* 如果你在写一个驱动，请使用dev\_dbg \*/
30. #if defined(DEBUG)
31. #define pr\_debug(fmt, ...) \
32. printk(KERN\_DEBUG pr\_fmt(fmt), ##\_\_VA\_ARGS\_\_)
33. #elif defined(CONFIG\_DYNAMIC\_DEBUG)
34. /\* dynamic\_pr\_debug() uses pr\_fmt() internally so we don't need it here \*/
35. #define pr\_debug(fmt, ...) \
36. dynamic\_pr\_debug(fmt, ##\_\_VA\_ARGS\_\_)
37. #else
38. #define pr\_debug(fmt, ...) \
39. no\_printk(KERN\_DEBUG pr\_fmt(fmt), ##\_\_VA\_ARGS\_\_)
40. #endif

     从上面的代码大家应该就可以知道这些宏的使用了。值得注意的是：pr\_devel和 pr\_debug这些宏只有在定义了DEBUG之后才会产生实际的printk代码，这样方便了内核开发：在代码中使用这些宏，当调试结束，只要简单地#undef DEBUG就可以消除这些调试使用的代码，无需真正地去删除调试输出代码。

**（3）输出速率控制**

   在调试的时候，有时某些部分可能printk会产生大量输出, 导致系统无法正常工作，并可能使系统日志ring buffer溢出（旧的信息被快速覆盖）。特别地，当使用一个慢速控制台设备(如串口), 过量输出也能拖慢系统。这样反而难于发现系统出问题的地方。所以你应当非常注意：正常操作时不应当打印任何东西，打印的输出应当是指示需要注意的异常，并小心不要做过头。

   在某些情况下, 最好的做法是设置一个标志变量表示：已经打印过这个了，以后不再打印任何这个信息。而对于打印速率的控制内核已经提供了一个现成的宏:

1. #define printk\_ratelimit() \_\_printk\_ratelimit(\_\_func\_\_)

     这个函数应当在你认为打印一个可能会出现大量重复的消息之前调用，如果这个函数返回非零值, 继续打印你的消息, 否则跳过它。典型的调用如这样:

1. if (printk\_ratelimit())
2. printk(KERN\_NOTICE "The printer is still on fire\n");

  printk\_ratelimit通过跟踪发向控制台的消息的数量和时间来工作。当输出超过一个限度, printk\_ratelimit 开始返回 0 使消息被丢弃。我们可以通过修改 ：

1. /proc/sys/kern/printk\_ratelimit( 可以看作一个监测周期，在这个周期内只能发出下面的控制量的信息)
2. /proc/sys/kernel/printk\_ratelimit\_burst(以上周期内的最大消息数，如果超过了printk\_ratelimit()返回0)

来控制消息的输出.

   在<linux/kernel.h>中还定义了其他的宏，比如printk\_ratelimited(fmt, ...)等，有兴趣的朋友就去文件中看看便知，很好理解的。

**（4）最后特别提醒：**

**1、虽然printk很健壮，但是看了源码你就知道，这个函数的效率很低：做字符拷贝时一次只拷贝一个字节，且去调用console输出可能还产生中断。所以如果你的驱动在功能调试完成以后做性能测试或者发布的时候千万记得尽量减少printk输出，做到仅在出错时输出少量信息。否则往console输出无用信息影响性能。**我刚开始学驱动的时候就犯过这样的白痴错误，在测试CAN驱动性能的时候居然printk出信息来核对，结果直接宕机。

**2、printk的临时缓存printk\_buf只有1K，所有一次printk函数只能记录<1K的信息到log buffer，并且printk使用的“ring buffer”**

**三、printk的内核实现**

    对于Linux的printk内核日志，常常被称为kernel ring buffer，这是由于printk的缓存实现就是使用了一个简单的ring buffer（环形缓冲区）。但是这里需要注意的是，不要和内核trace系统ring buffer混淆，虽然他们都是为了跟踪调试，但是trace系统的ring buffer实现更加完善复杂，而printk使用的ring buffer则非常简单，其实就定义了一个字符数组：

1. static char \_\_log\_buf[\_\_LOG\_BUF\_LEN];

并使用了一套指针来管理：

1. /\*
2. \* 在指向log\_buf时并没有用log\_buf\_len做限制 - 所以他们
3. \* 在作为下标使用前必须用掩码处理（去除CONFIG\_LOG\_BUF\_SHIFT以上的高位）
4. \*/
5. static unsigned log\_start; /\* log\_buf中的索引: 指向由syslog()读取的下一个字符 \*/
6. static unsigned con\_start; /\* log\_buf中的索引: 指向发送到console的下一个字符 \*/
7. static unsigned log\_end; /\* log\_buf中的索引:最近写入的字符地址 + 1 \*/

    具体的实现CU中已经有一位博友写过了，我这里就不再啰嗦了，我转载备份了一下：[**《**printk实现分析**》**](http://blog.chinaunix.net/uid-20543672-id-3217096.html)

**四、用户空间访问内核日志**

    用户空间访问和控制内核日志有两个接口：

   （1）通过glibc的klogctl函数接口调用内核的syslog系统调用

   （2）通过fs/proc/kmsg.c内核模块中导出的procfs接口：/proc/kmsg文件。

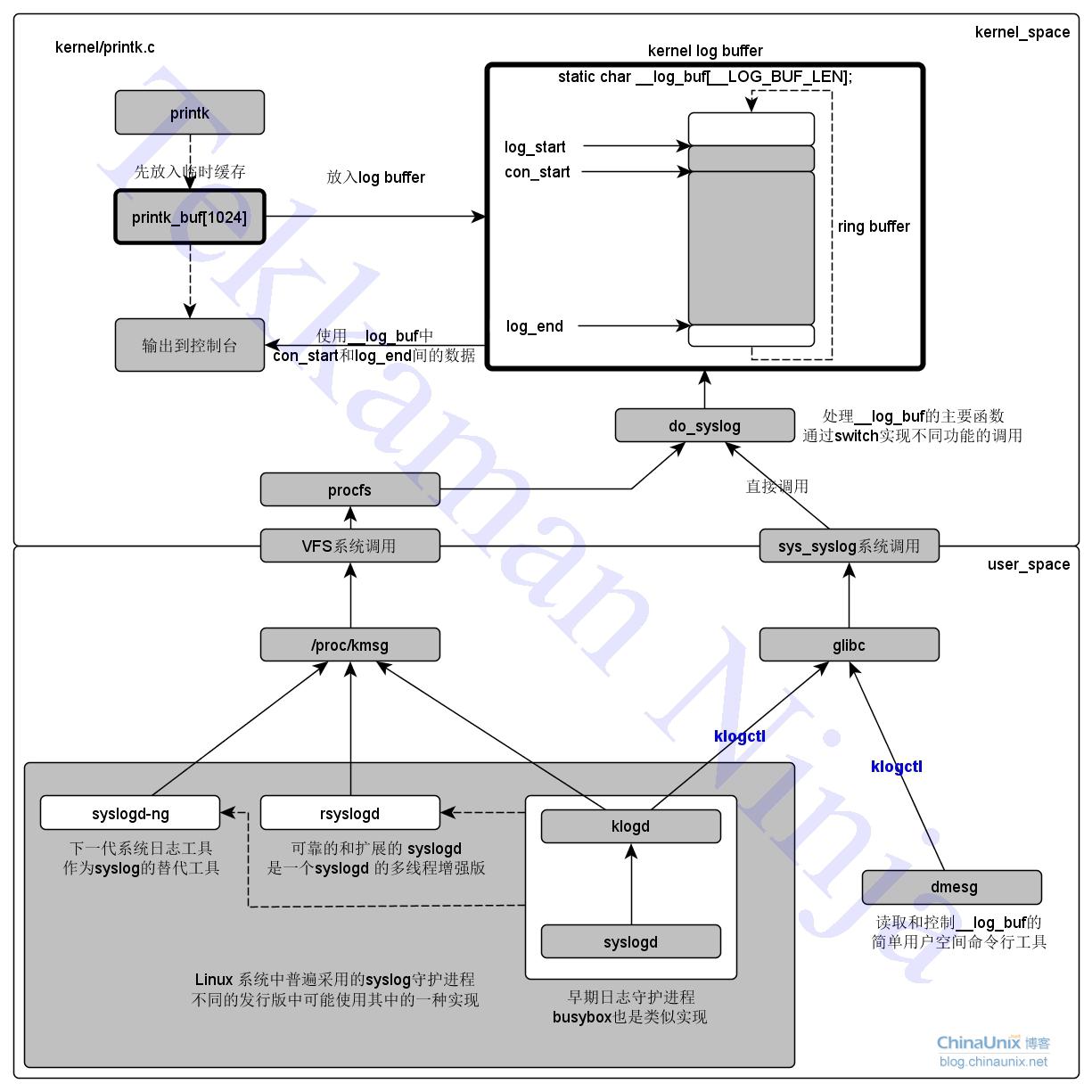
    他们其实最终都调用了/kernel/printk.c中的do\_syslog函数，实现对\_\_log\_buf的访问及相关变量的修改。

    但值得注意的是：**从/proc/kmsg中获取数据，那么\_\_log\_buf中被读取过的数据就不再保留（也就是会修改log\_start指针）, 然而 syslog 系统调用返回日志数据并保留数据给其他进程。**读取/proc文件是 klogd的默认做法。dmesg命令可用来查看缓存的内容并保留它，其实它是将\_\_log\_buf中的所有内容返回给stdout，并不管它是否已经被读取过。

    这里我还是推荐大家 RTFSC – Read The Fucking Source Code，自己看这些代码比什么都强，我这里就只引个路。

    在用户空间有专门用于记录系统日志的程序，统称为“syslog守护进程”。早期及现在的大部分嵌入式系统使用的是klogd+syslogd组合，现在大多数发行版都使用rsyslogd或者syslogd-ng了。这些用户空间的程序我这里就不分析了，我不擅长，运维的可能比较清楚。我只知道一下他们大致的调用关系就好。

    这里我用一张图来总结一下内核printk和日志系统的总体结构：

[](http://blog.chinaunix.net/attachment/201205/19/20543672_1337444365LWQY.jpg)

**极力推荐阅读：**[**《内核日志：API 及实现----从内核到用户空间的日志》**](http://www.ibm.com/developerworks/cn/linux/l-kernel-logging-apis/index.html)