【linux-2.6.31】探秘 Linux SysRq 魔法键

Translated By : openspace

Date : 2009-12-18

用作 sysrq.c 的文档

什么是 SysRq 魔法键
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

它是“魔法”组合键；不管内核在做什么事情，只要没有被完全锁住，使用这些组合键可以让内核立即做出回应。

如何激活 SysRq 魔法键
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

要在配置内核的时候将'**Magic SysRq key (CONFIG_MAGIC_SYSRQ)**'设置为"yes"。当运行编译了 SysRq 的内核时，**/proc/sys/kernel/sysrq** 控制可以通过 SysRq 键调用的功能。缺省，该文件值为 1，表示允许所有的 SysRq 请求（早一点的版本中 SysRq 缺省为禁止的，你需要自己来激活它，而现在则不需要这样做了）。下面是/proc/sys/kernel/sysrq 中的可能取值：

```
        0 - 完全禁止 sysrq
        1 - 激活 sysrq 所有功能
       >1 - 允许的 sysrq 功能的位掩码（下面有更加详细的描述）：
            2 - 激活控制台日志级别控制
            4 - 激活键盘(SAK, unraw)控制
            8 - 激活调试进程的 dump 信息等
           16 - 激活 sync 命令
           32 - 激活只读 remount
           64 - 激活进程的信号(term, kill, oom-kill)功能
          128 - 允许 reboot/poweroff
          256 - 允许所有实时任务的 nice 操作
```

可以通过下列命令设置文件中的值：

```
echo "number" >/proc/sys/kernel/sysrq
```

注意/proc/sys/kernel/sysrq 的值只能影响通过键盘调用的操作。通过**/proc/sysrq-trigger** 调用的操作总是允许的（要求用户有管理权限）。

如何使用 SysRq 魔法键

~~~~~~~~~~~~~~~~~~~~~~

x86 平台

按下组合键'**ALT-SysRq-<command key>**'。注意，一些键盘上可能没有标记'SysRq'的键。'SysRq'键也是'Print Screen'键；还有一些键盘不支持一次按下这么多键，所以最好依次"press Alt"、"press SysRq"、"release SysRq"、"press <command key>"，然后释放

SPARC 平台

按下'**ALT-STOP-<command key>**'

串口控制台(仅用于 PC 标准串口)

发送 BREAK，然后在 5 秒内敲入命令键。发送两次 BREAK 被看做一个普通的 BREAK

PowerPC 平台

按下'**ALT - Print Screen (or F13) - <command key>**'，Print Screen (or F13) - <command key>可能就够用了

其他平台

如果你知道适合于其它体系结构的组合键，请告诉我，这样我可以将它们添加到这里

适于所有平台

向**/proc/sysrq-trigger** 写入字符。例如：

```
echo t > /proc/sysrq-trigger
```

命令键有哪些

~~~~~~~~~~~~

'b'

立即重启系统，不会同步或者卸载磁盘

'c'

通过解引用 NULL 指针导致系统崩溃，会产生 crashdump，如果之前这样配置了的话

'd'

显示所有持有的锁

'e'

向所有进程发送 SIGTERM，init 除外

'f'

调用 oom_kill 杀死一个内存消耗大的进程

'g'

kgdb 在 ppc 和 sh 平台上使用

'h'

显示帮助信息（实际上其它任何键都可以，但是'h'更易于记忆）

'i'

向所有进程发送 SIGKILL，init 除外

'j'

强制性"Just thaw it"——用于通过 FIFREEZE ioctl 冻结的文件系统

'k'

安全文件键(SAK)杀死当前虚拟终端的所有程序。注意：参考后面有关 SAK 的信息

'l'

显示所有活跃 CPU 的栈的反向跟踪信息

'm'

转储当前内存状态到控制台

'n'

用于让所有 RT 进程支持 nice 操作

'o'

关闭系统（如果已经这样配置并且系统支持的话）

'p'

转储当前寄存器和标志信息到控制台

'q'

转储所有 CPU 的 armed hrtimers 列表（不是普通的 timer_list 定时器）以及关于 clockevent 设备的详细信息

'r'

关闭键盘 raw 模式，设置为 XLATE

's'

尝试同步所有挂载的文件系统

't'

将当前任务的列表和信息转储到控制台

'u'

尝试将所有挂载的文件系统重新挂载为只读

'v'

转储 Voyager SMP processor 信息到控制台

'w'

转储处于 uninterruptable 阻塞状态的任务

'x'

由 ppc/powerpc 平台上的 xmon 接口使用

'z'

转储 ftrace 缓冲区

'0'-'9'

设置控制台日志级别，控制哪些内核信息可以打印到控制台。（例如'0'表示只有类似 PANIC 或者 OOPS 的紧急信息才会发送到控制台）

SysRq 键用于什么场合

~~~~~~~~~~~~~~~~~~~~~~

un'R'aw 在 X server 或者一个 svgalib 程序崩溃的时候非常有用。

sa'K'(安全访问键)可用于在登录时确保没有 trojan 程序窃取你的密码。它会杀死指定控制台上的所有进程，这样你看到的登录提示来自于 init，而不是 trojan 程序。重要提示：它并不是一个类似于 c2 兼容系统中的真正的 SAK，不要混淆。有人发现把它用作(System Attention Key)可用于退出一个阻碍你切换控制台的程序。（例如 X 或者一个 svgalib 程序）

re'B'oot 用于不能关机的场合，但是需要先执行'S'ync 和'U'mount 操作。

'C'rash 用于在系统 hung 的时候手工触发一个 crashdump。注意，如果没有 dump 机制那么仅仅是触发一个崩溃。

'S'ync 用于在系统锁住的时候同步磁盘，以减少数据丢失和修复。注意直到从屏幕上看到"OK"或者"Done"时 sync 才发生。（如果内核冲突，你可能得不到 OK 或者 Done 信息）

'U'mount 同'S'ync 一样有用。我在系统锁住的时候通常会'S'ync、'U'mount，然后 re'B'oot。这减少了我的 fsck 操作。通用，直到你看到屏幕上显示"OK"或者"Done"信息 unmount(remount 为只读)才发生。

日志级别**'0'-'9'**用于你的屏幕上充满内核消息而你有不希望看到的时候。选择'0'会阻止所有信息输出到控制台（但是这些消息仍然会记录下来，如果 syslogd/klogd 正在运行的话），非常紧急的内核消息除外。

t'E'rm 和 **k'I'll** 用于一些进程脱离你的控制而你无法将它们杀死，而它们会生成其它进程的场合。

"'J'ust thaw it"用于因为 FIFREEZE ioctl 冻结一个（可能是 root）文件系统而导致系统没有反应的场合。

有时候使用 SysRq 后似乎卡住了，这时该怎么做

~~~~~~

我也遇到过这种情况。我发现敲入键盘两边的 shift、alt 和 control，然后再敲入一个无效的 sysrq 序列可以修复这个问题。（例如，类似 alt-sysrq-z）。切换到另一个虚拟控制台(ALT+Fn)然后再切换回来应该也行得通。

敲入 SysRq，但是什么事情也没有发生，这是怎么回事

~~~~~~
~~~~~~

有些键盘发送不同的 SysRq 扫描码，而不是预定义的 0x54。如果 SysRq 在键盘上不能正常工作，执行'**showkey -s**'查看正确的扫描码序列。然后使用'**setkeycodes <sequence> 84**'将该序列定义为 SysRq 码（84 是 0x54 的十进制表示）。最后将该命令写到一个 boot 脚本中。顺便说一下，退出'showkey'的方法是 10 秒内不敲入任何命令。

如何向一个模块中添加 SysRQ 键事件？

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

要注册一个基本的函数，必须先包含头文件'**include/linux/sysrq.h**'，该文件定义了必要的接口。然后，必须创建一个 **sysrq_key_op** 结构，设置：A) 使用的键处理函数；B) 一个 help_msg 字符串，用于 SysRQ 打印帮助信息的时候输出；C) 一个 action_msg 字符串，用于在处理函数被调用前输出。处理器函数必须遵循'sysrq.h'中定义的原型。

创建 sysrq_key_op 后，调用内核函数 register_sysrq_key(int key, struct sysrq_key_op *op_p)，该函数会将'op_p'所指的操作注册到表中键值'key'处，如果表中的对应位置为空的话。模块卸载时，必须调用 unregister_sysrq_key(int key, struct sysrq_key_op *op_p)，该函数会移除键值'key'中的'op_p'指向的操作结构，如果操作结构注册在对应项中的话。这是为了防止对应项在你注册后被覆盖。

Magic SysRQ 系统通过注册在一个键操作查找表中注册键操作结构运作，该表在'**drivers/char/sysrq.c**'中定义。该键值表中有些操作是在编译时注册的，但是该表是可变的，访问该表可以通过导出的两个函数：

register_sysrq_key 和 **unregister_sysrq_key**

当然，不要在表中留下非法指针。例如，当调用 register_sysrq_key()的模块退出时，必须调用 unregister_sysrq_key()清理它使用的 sysrq 键表的表项。表中的 Null 指针是安全的。

如果出于某些原因，需要从 **handle_sysrq** 调用的一个函数中调用 handle_sysrq，那么必须意识到当前处于一个锁之中（还处于一个中断处理函数中，不能睡眠！），所以要通过调用**__handle_sysrq_nolock** 来代替。

当敲入 SysRq 组合键的时候只有 header 显示在控制台上

~~~~~~

Sysrq 输出与其他控制台输出一样，都受相同的控制台日志级别控制。这意味着如果内核像发行版内核那样'quiet'启动，那么输出可能不会出现在控制台上，即便它出现在 dmesg 缓冲区中，并且可以通过 dmesg 命令或者/proc/kmsg 访问。不同的是，sysrq 命令的 header 会传递给所有读取控制台的进程，看起来就像当前日志级别处于最高状态。如果只输出了 header，基本上可以确定内核的日志级别非常低。要让输出显示到控制台，需要临时提升控制台的日志级别，使用 alt-sysrq-8 或者：

```
echo 8 > /proc/sysrq-trigger
```

记住，在触发 sysrq 命令后将日志级别返回到正常状态

有问题的时候向谁咨询

~~~~~~~~~~~~~~~~~~~~~~~~

我会尽快解答关于注册系统的问题。

- Crutcher
~~~~~~~~~~~~~~~~~~~~~~~~

致谢

~~~

Mydraal <vulpyne@vulpyne.net> 编写

Adam Sulmicki <adam@cfar.umd.edu> 进行了更新

Jeremy M. Dolan <jmd@turbogeek.org> 2001/01/28 10:15:59 进行了更新

Crutcher Dunnavant <crutcher+kernel@datastacks.com> 添加了新的信息
~~~

【原文】Documentation/sysrq.txt

```
Linux Magic System Request Key Hacks
Documentation for sysrq.c

*  What is the magic SysRq key?
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

It is a 'magical' key combo you can hit which the kernel will respond to
regardless of whatever else it is doing, unless it is completely locked up.

*  How do I enable the magic SysRq key?
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

You need to say "yes" to 'Magic SysRq key (CONFIG_MAGIC_SYSRQ)' when
configuring the kernel. When running a kernel with SysRq compiled in,
/proc/sys/kernel/sysrq controls the functions allowed to be invoked via
the SysRq key. By default the file contains 1 which means that every
possible SysRq request is allowed (in older versions SysRq was disabled
by default, and you were required to specifically enable it at run-time
but this is not the case any more). Here is the list of possible values
in /proc/sys/kernel/sysrq:
   0 - disable sysrq completely
   1 - enable all functions of sysrq
  >1 - bitmask of allowed sysrq functions (see below for detailed function
       description):
          2 - enable control of console logging level
          4 - enable control of keyboard (SAK, unraw)
          8 - enable debugging dumps of processes etc.
         16 - enable sync command
         32 - enable remount read-only
         64 - enable signalling of processes (term, kill, oom-kill)
        128 - allow reboot/poweroff
        256 - allow nicing of all RT tasks

You can set the value in the file by the following command:
    echo "number" >/proc/sys/kernel/sysrq

Note that the value of /proc/sys/kernel/sysrq influences only the invocation
via a keyboard. Invocation of any operation via /proc/sysrq-trigger is always
allowed (by a user with admin privileges).

*  How do I use the magic SysRq key?
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

On x86   - You press the key combo 'ALT-SysRq-<command key>'. Note - Some
           keyboards may not have a key labeled 'SysRq'. The 'SysRq' key is
           also known as the 'Print Screen' key. Also some keyboards cannot
          handle so many keys being pressed at the same time, so you might
          have better luck with "press Alt", "press SysRq", "release SysRq",
          "press <command key>", release everything.

On SPARC - You press 'ALT-STOP-<command key>', I believe.

On the serial console (PC style standard serial ports only) -
           You send a BREAK, then within 5 seconds a command key. Sending
```

BREAK twice is interpreted as a normal BREAK.

On PowerPC - Press 'ALT - Print Screen (or F13) - <command key>,
Print Screen (or F13) - <command key> may suffice.

On other - If you know of the key combos for other architectures, please
let me know so I can add them to this section.

On all - write a character to /proc/sysrq-trigger. e.g.:

```
echo t > /proc/sysrq-trigger
```

* What are the 'command' keys?
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

'b' - Will immediately reboot the system without syncing or unmounting
your disks.

'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 oom_kill to kill a memory hog process.

'g' - Used by kgdb on ppc and sh platforms.

'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'   - Dumps Voyager SMP processor info to your console.

'w'   - Dumps tasks that are in uninterruptable (blocked) state.

'x'   - Used by xmon interface on ppc/powerpc platforms.

'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.)

* Okay, so what can I use them for?

Well, un'R'aw is very handy when your X server or a svgalib program crashes.

sa'K' (Secure Access Key) is useful when you want to be sure there is no trojan program running at console which could grab your password when you would try to login. It will kill all programs on given console, thus letting you make sure that the login prompt you see is actually the one from init, not some trojan program.

```
IMPORTANT: In its true form it is not a true SAK like the one in a :IMPORTANT
IMPORTANT: c2 compliant system, and it should not be mistaken as   :IMPORTANT
IMPORTANT: such.                                                   :IMPORTANT
```

It seems others find it useful as (System Attention Key) which is useful when you want to exit a program that will not let you switch consoles. (For example, X or a svgalib program.)

re'B'oot is good when you're unable to shut down. But you should also 'S'ync and 'U'mount first.

'C'rash can be used to manually trigger a crashdump when the system is hung. Note that this just triggers a crash if there is no dump mechanism available.

'S'ync is great when your system is locked up, it allows you to sync your disks and will certainly lessen the chance of data loss and fscking. Note that the sync hasn't taken place until you see the "OK" and "Done" appear on the screen. (If the kernel is really in strife, you may not ever get the OK or Done message...)

'U'mount is basically useful in the same ways as 'S'ync. I generally 'S'ync, 'U'mount, then re'B'oot when my system locks. It's saved me many a fsck. Again, the unmount (remount read-only) hasn't taken place until you see the "OK" and "Done" message appear on the screen.

The loglevels '0'-'9' are useful when your console is being flooded with kernel messages you do not want to see. Selecting '0' will prevent all but the most urgent kernel messages from reaching your console. (They will still be logged if syslogd/klogd are alive, though.)

t'E'rm and k'I'll are useful if you have some sort of runaway process you are unable to kill any other way, especially if it's spawning other processes.

"'J'ust thaw it" is useful if your system becomes unresponsive due to a frozen (probably root) filesystem via the FIFREEZE ioctl.

* Sometimes SysRq seems to get 'stuck' after using it, what can I do?

That happens to me, also. I've found that tapping shift, alt, and control on both sides of the keyboard, and hitting an invalid sysrq sequence again will fix the problem. (i.e., something like alt-sysrq-z). Switching to another virtual console (ALT+Fn) and then back again should also help.

* I hit SysRq, but nothing seems to happen, what's wrong?

There are some keyboards that send different scancodes for SysRq than the pre-defined 0x54. So if SysRq doesn't work out of the box for a certain keyboard, run 'showkey -s' to find out the proper scancode sequence. Then use 'setkeycodes <sequence> 84' to define this sequence to the usual SysRq code (84 is decimal for 0x54). It's probably best to put this command in a boot script. Oh, and by the way, you exit 'showkey' by not typing anything for ten seconds.

* I want to add SysRQ key events to a module, how does it work?

In order to register a basic function with the table, you must first include the header 'include/linux/sysrq.h', this will define everything else you need. Next, you must create a sysrq_key_op struct, and populate it with A) the key handler function you will use, B) a help_msg string, that will print when SysRQ prints help, and C) an action_msg string, that will print right before your handler is called. Your handler must conform to the prototype in 'sysrq.h'.

After the sysrq_key_op is created, you can call the kernel function register_sysrq_key(int key, struct sysrq_key_op *op_p); this will register the operation pointed to by 'op_p' at table key 'key', if that slot in the table is blank. At module unload time, you must call the function unregister_sysrq_key(int key, struct sysrq_key_op *op_p), which will remove the key op pointed to by 'op_p' from the key 'key', if and only if it is currently registered in that slot. This is in case the slot has been overwritten since you registered it.

The Magic SysRQ system works by registering key operations against a key op lookup table, which is defined in 'drivers/char/sysrq.c'. This key table has a number of operations registered into it at compile time, but is mutable, and 2 functions are exported for interface to it:

```
        register_sysrq_key and unregister_sysrq_key.
```

Of course, never ever leave an invalid pointer in the table. I.e., when your module that called register_sysrq_key() exits, it must call unregister_sysrq_key() to clean up the sysrq key table entry that it used. Null pointers in the table are always safe. :)

If for some reason you feel the need to call the handle_sysrq function from within a function called by handle_sysrq, you must be aware that you are in a lock (you are also in an interrupt handler, which means don't sleep!), so you must call __handle_sysrq_nolock instead.

* When I hit a SysRq key combination only the header appears on the console?

Sysrq output is subject to the same console loglevel control as all other console output. This means that if the kernel was booted 'quiet' as is common on distro kernels the output may not appear on the actual console, even though it will appear in the dmesg buffer, and be accessible via the dmesg command and to the consumers of /proc/kmsg. As a specific exception the header line from the sysrq command is passed to all console consumers as if the current loglevel was maximum. If only the header is emitted it is almost certain that the kernel loglevel is too low. Should you require the output on the console channel then you will need to temporarily up the console loglevel using alt-sysrq-8 or:

```
    echo 8 > /proc/sysrq-trigger
```

Remember to return the loglevel to normal after triggering the sysrq command you are interested in.

* I have more questions, who can I ask?

And I'll answer any questions about the registration system you got, also responding as soon as possible.
 -Crutcher

* Credits

Written by Mydraal <vulpyne@vulpyne.net>
Updated by Adam Sulmicki <adam@cfar.umd.edu>
Updated by Jeremy M. Dolan <jmd@turbogeek.org> 2001/01/28 10:15:59
Added to by Crutcher Dunnavant <crutcher+kernel@datastacks.com>