<http://www.cnblogs.com/hoys/archive/2011/04/10/2011270.html>

relayfs是一个快速的转发（relay）数据的文件系统，它以其功能而得名。它为那些需要从内核空间转发大量数据到用户空间的工具和应用提供了快速有效的转发机制。

Channel是relayfs文件系统定义的一个主要概念，每一个channel由一组内核缓存组成，每一个CPU有一个对应于该channel 的内核缓存，每一个内核缓存用一个在relayfs文件系统中的文件文件表示，内核使用relayfs提供的写函数把需要转发给用户空间的数据快速地写入当前CPU上的channel内核缓存，用户空间应用通过标准的文件I/O函数在对应的channel文件中可以快速地取得这些被转发出的数据mmap 来。写入到channel中的数据的格式完全取决于内核中创建channel的模块或子系统。

relayfs的用户空间API：

relayfs实现了四个标准的文件I/O函数，open、mmap、poll和close.

open()，打开一个channel在某一个CPU上的缓存对应的文件。

mmap()，把打开的channel缓存映射到调用者进程的内存空间。

read ()，读取channel缓存，随后的读操作将看不到被该函数消耗的字节，如果channel的操作模式为非覆盖写，那么用户空间应用在有内核模块写时仍 可以读取，但是如果channel的操作模式为覆盖式，那么在读操作期间如果有内核模块进行写，结果将无法预知，因此对于覆盖式写的channel，用户 应当在确认在channel的写完全结束后再进行读。

poll()，用于通知用户空间应用转发数据跨越了子缓存的边界，支持的轮询标志有POLLIN、POLLRDNORM和POLLERR。

close()，关闭open函数返回的文件描述符，如果没有进程或内核模块打开该channel缓存，close函数将释放该channel缓存。

注意：用户态应用在使用上述API时必须保证已经挂载了relayfs文件系统，但内核在创建和使用channel时不需要relayfs已经挂载。下面命令将把relayfs文件系统挂载到/mnt/relay。

mount -t relayfs relayfs /mnt/relay

relayfs内核API：

relayfs提供给内核的API包括四类：channel管理、写函数、回调函数和辅助函数。

Channel管理函数包括：

relay\_open(base\_filename, parent, subbuf\_size, n\_subbufs, overwrite, callbacks)

relay\_close(chan)

relay\_flush(chan)

relay\_reset(chan)

relayfs\_create\_dir(name, parent)

relayfs\_remove\_dir(dentry)

relay\_commit(buf, reserved, count)

relay\_subbufs\_consumed(chan, cpu, subbufs\_consumed)

写函数包括：

relay\_write(chan, data, length)

\_\_relay\_write(chan, data, length)

relay\_reserve(chan, length)

回调函数包括：

subbuf\_start(buf, subbuf, prev\_subbuf\_idx, prev\_subbuf)

buf\_mapped(buf, filp)

buf\_unmapped(buf, filp)

辅助函数包括：

relay\_buf\_full(buf)

subbuf\_start\_reserve(buf, length)

前面已经讲过，每一个channel由一组channel缓存组成，每个CPU对应一个该channel的缓存，每一个缓存又由一个或多个子缓存组成，每一个缓存是子缓存组成的一个环型缓存。

函数relay\_open用于创建一个channel并分配对应于每一个CPU的缓存，用户空间应用通过在relayfs文件系统中对应的文件可以 访问channel缓存，参数base\_filename用于指定channel的文件名，relay\_open函数将在relayfs文件系统中创建 base\_filename0..base\_filenameN-1，即每一个CPU对应一个channel文件，其中N为CPU数，缺省情况下，这些文件将建立在relayfs文件系统的根目录下，但如果参数parent非空，该函数将把channel文件创建于parent目录下，parent目录使 用函数relay\_create\_dir创建，函数relay\_remove\_dir用于删除由函数relay\_create\_dir创建的目录，谁创建的目录，谁就负责在不用时负责删除。参数subbuf\_size用于指定channel缓存中每一个子缓存的大小，参数n\_subbufs用于指定 channel缓存包含的子缓存数，因此实际的channel缓存大小为(subbuf\_size x n\_subbufs)，参数overwrite用于指定该channel的操作模式，relayfs提供了两种写模式，一种是覆盖式写，另一种是非覆盖式 写。使用哪一种模式完全取决于函数subbuf\_start的实现，覆盖写将在缓存已满的情况下无条件地继续从缓存的开始写数据，而不管这些数据是否已经 被用户应用读取，因此写操作决不失败。在非覆盖写模式下，如果缓存满了，写将失败，但内核将在用户空间应用读取缓存数据时通过函数 relay\_subbufs\_consumed()通知relayfs。如果用户空间应用没来得及消耗缓存中的数据或缓存已满，两种模式都将导致数据丢失，唯一的区别是，前者丢失数据在缓存开头，而后者丢失数据在缓存末尾。一旦内核再次调用函数relay\_subbufs\_consumed()，已满的缓存将不再满，因而可以继续写该缓存。当缓存满了以后，relayfs将调用回调函数buf\_full()来通知内核模块或子系统。当新的数据太大无法写 入当前子缓存剩余的空间时，relayfs将调用回调函数subbuf\_start()来通知内核模块或子系统将需要使用新的子缓存。内核模块需要在该回调函数中实现下述功能：

初始化新的子缓存；

如果1正确，完成当前子缓存；

如果2正确，返回是否正确完成子缓存切换；

在非覆盖写模式下，回调函数subbuf\_start()应该如下实现：

static int subbuf\_start(struct rchan\_buf \*buf, void \*subbuf, void \*prev\_subbuf, unsigned intprev\_padding){

if (prev\_subbuf)

\*((unsigned \*)prev\_subbuf) = prev\_padding;

if (relay\_buf\_full(buf))

return 0;

subbuf\_start\_reserve(buf, sizeof(unsigned int));

return 1;

}

如果当前缓存满，即所有的子缓存都没读取，该函数返回0，指示子缓存切换没有成功。当子缓存通过函数relay\_subbufs\_consumed ()被读取后，读取者将负责通知relayfs，函数relay\_buf\_full()在已经有读者读取子缓存数据后返回0，在这种情况下，子缓存切换成 功进行。

在覆盖写模式下， subbuf\_start()的实现与非覆盖模式类似：

static int subbuf\_start(struct rchan\_buf \*buf, void \*subbuf, void \*prev\_subbuf, unsigned int prev\_padding)

{

if (prev\_subbuf)

\*((unsigned \*)prev\_subbuf) = prev\_padding;

subbuf\_start\_reserve(buf, sizeof(unsigned int));

return 1;

}

只是不做relay\_buf\_full()检查，因为此模式下，缓存是环行的，可以无条件地写。因此在此模式下，子缓存切换必定成功，函数 relay\_subbufs\_consumed() 也无须调用。如果channel写者没有定义subbuf\_start()，缺省的实现将被使用。 可以通过在回调函数subbuf\_start()中调用辅助函数subbuf\_start\_reserve()在子缓存中预留头空间，预留空间可以保存任 何需要的信息，如上面例子中，预留空间用于保存子缓存填充字节数，在subbuf\_start()实现中，前一个子缓存的填充值被设置。前一个子缓存的填 充值和指向前一个子缓存的指针一道作为subbuf\_start()的参数传递给subbuf\_start()，只有在子缓存完成后，才能知道填充值。 subbuf\_start()也被在channel创建时分配每一个channel缓存的第一个子缓存时调用，以便预留头空间，但在这种情况下，前一个子 缓存指针为NULL。

内核模块使用函数relay\_write()或\_\_relay\_write()往channel缓存中写需要转发的数据，它们的区别是前者失效了本 地中断，而后者只抢占失效，因此前者可以在任何内核上下文安全使用，而后者应当在没有任何中断上下文将写channel缓存的情况下使用。这两个函数没有 返回值，因此用户不能直接确定写操作是否失败，在缓存满且写模式为非覆盖模式时，relayfs将通过回调函数buf\_full来通知内核模块。

函数relay\_reserve()用于在channel缓存中预留一段空间以便以后写入，在那些没有临时缓存而直接写入channel缓存的内核 模块可能需要该函数，使用该函数的内核模块在实际写这段预留的空间时可以通过调用relay\_commit()来通知relayfs。当所有预留的空间全 部写完并通过relay\_commit通知relayfs后，relayfs将调用回调函数deliver()通知内核模块一个完整的子缓存已经填满。由 于预留空间的操作并不在写channel的内核模块完全控制之下，因此relay\_reserve()不能很好地保护缓存，因此当内核模块调用 relay\_reserve()时必须采取恰当的同步机制。

当内核模块结束对channel的使用后需要调用relay\_close() 来关闭channel，如果没有任何用户在引用该channel，它将和对应的缓存全部被释放。

函数relay\_flush()强制在所有的channel缓存上做一个子缓存切换，它在channel被关闭前使用来终止和处理最后的子缓存。

函数relay\_reset()用于将一个channel恢复到初始状态，因而不必释放现存的内存映射并重新分配新的channel缓存就可以使用channel，但是该调用只有在该channel没有任何用户在写的情况下才可以安全使用。

回调函数buf\_mapped() 在channel缓存被映射到用户空间时被调用。

回调函数buf\_unmapped()在释放该映射时被调用。内核模块可以通过它们触发一些内核操作，如开始或结束channel写操作。

在源代码包中给出了一个使用relayfs的示例程序relayfs\_exam.c，它只包含一个内核模块，对于复杂的使用，需要应用程序配合。该模块实现了类似于文章中seq\_file示例实现的功能。

当然为了使用relayfs，用户必须让内核支持relayfs，并且要mount它，下面是作者系统上的使用该模块的输出信息：

$ mkdir -p /relayfs  
$ insmod ./relayfs-exam.ko  
$ mount -t relayfs relayfs /relayfs  
$ cat /relayfs/example0  
…  
$  
   
relayfs是一种比较复杂的内核态与用户态的数据交换方式，本例子程序只提供了一个较简单的使用方式，对于复杂的使用，请参考relayfs用例页面<http://relayfs.sourceforge.net/examples.html>。

//kernel module: relayfs-exam.c

#include <linux/module.h>

#include <linux/relayfs\_fs.h>

#include <linux/string.h>

#include <linux/sched.h>

#define WRITE\_PERIOD (HZ \* 60)

static struct rchan \* chan;

static size\_t subbuf\_size = 65536;

static size\_t n\_subbufs = 4;

static char buffer[256];

void relayfs\_exam\_write(unsigned long data);

static DEFINE\_TIMER(relayfs\_exam\_timer, relayfs\_exam\_write, 0, 0);

void relayfs\_exam\_write(unsigned long data)

{

int len;

task\_t \* p = NULL;

len = sprintf(buffer, "Current all the processes:\n");

len += sprintf(buffer + len, "process name\t\tpid\n");

relay\_write(chan, buffer, len);

for\_each\_process(p) {

len = sprintf(buffer, "%s\t\t%d\n", p->comm, p->pid);

relay\_write(chan, buffer, len);

}

len = sprintf(buffer, "\n\n");

relay\_write(chan, buffer, len);

relayfs\_exam\_timer.expires = jiffies + WRITE\_PERIOD;

add\_timer(&relayfs\_exam\_timer);

}

/\*

\* subbuf\_start() relayfs callback.

\*

\* Defined so that we can 1) reserve padding counts in the sub-buffers, and

\* 2) keep a count of events dropped due to the buffer-full condition.

\*/

static int subbuf\_start(struct rchan\_buf \*buf,

void \*subbuf,

void \*prev\_subbuf,

unsigned int prev\_padding)

{

if (prev\_subbuf)

\*((unsigned \*)prev\_subbuf) = prev\_padding;

if (relay\_buf\_full(buf))

return 0;

subbuf\_start\_reserve(buf, sizeof(unsigned int));

return 1;

}

/\*

\* relayfs callbacks

\*/

static struct rchan\_callbacks relayfs\_callbacks =

{

.subbuf\_start = subbuf\_start,

};

/\*\*

\* module init - creates channel management control files

\*

\* Returns 0 on success, negative otherwise.

\*/

static int init(void)

{

chan = relay\_open("example", NULL, subbuf\_size,

n\_subbufs, &relayfs\_callbacks);

if (!chan) {

printk("relay channel creation failed.\n");

return 1;

}

relayfs\_exam\_timer.expires = jiffies + WRITE\_PERIOD;

add\_timer(&relayfs\_exam\_timer);

return 0;

}

static void cleanup(void)

{

del\_timer\_sync(&relayfs\_exam\_timer);

if (chan) {

relay\_close(chan);

chan = NULL;

}

}

module\_init(init);

module\_exit(cleanup);

MODULE\_LICENSE("GPL");