https://habr.com/ru/post/358184/

litwr29 мая 2018 в 23:00

Использование файлов-последовательностей ядра Linux

C *Разработка под Linux *

Характерная черта современного программирования в использовании глобальной сети как источника справочной информации, в частности, источника шаблонов для решения неизвестных или малоизвестных конкретному программисту проблем. Такой подход экономит массу времени и часто даёт вполне качественный результат. Однако, решения, выложенные в сети, хотя обычно правильны, но не всегда учитывают всех тонкостей решения той или иной проблемы, что приводит к появлению в кодах участков, которые обычно работают правильно, но при не совсем стандартных обстоятельствах становятся источниками неприятных неожиданностей.

Рассмотрим тему использования файлов-последовательностей в ядре Linux. Такие файлы считаются самым удобным механизмом для печати из режима ядра. Но на практике использовать их правильно гораздо сложнее, чем можно об этом подумать.

В сети доступно множество материалов по этой теме. Самый лучший – это исходники самого ядра с достаточно подробными комментариями. Проблема этого источника информации в его объёме. Если не знать в точности, что искать, то лучше, имея лишь ограниченное время, и не пробовать. У меня, когда возник интерес к теме, Google нашёл несколько на первый взгляд отличных источников информации: известную книгу The Linux Kernel Module Programming Guide и серию статей Роба Дея (Rob Day) по нужной теме. Источники не новые, но весьма солидные.

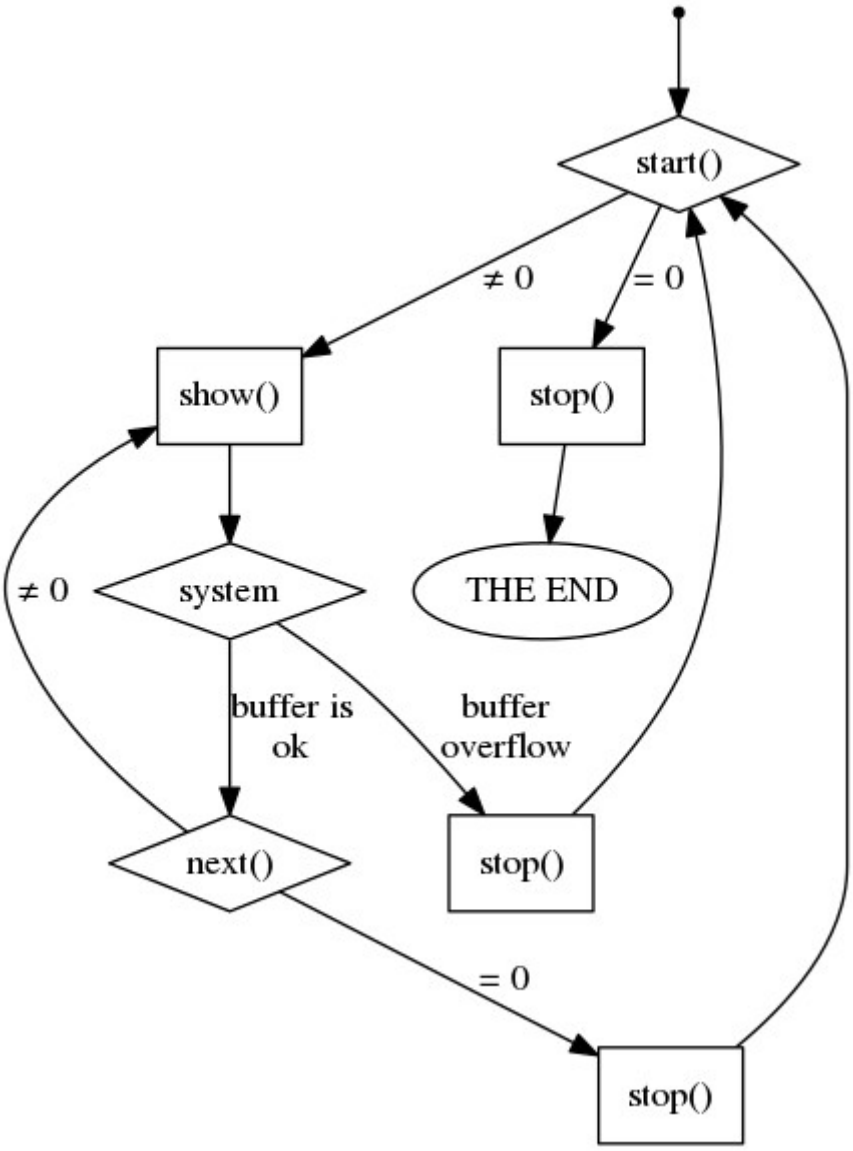
Рассмотрим сначала по-подробнее, когда естественно использовать файлы-последовательности. Самая типичная ситуация – это создание своего файла в файловой системе /proc. Читая файлы этой системы, можно получать самую разнообразную информацию об используемой аппаратуре, её драйверах, об оперативной памяти, о процессах и т. п.

Казалось бы распечатка чего бы то ни было – это самое простое из всего, что есть в программировании. Но работа в режиме ядра ОС накладывает множество ограничений, которые могут показаться совершенно невообразимыми разработчику прикладного ПО. В режиме ядра размер буфера печати ограничен размером страницы виртуальной памяти. Для архитектуры x86 – это четыре килобайта. Поэтому хорошая программа при распечатке данных большего объёма должна сначала

добиться максимального заполнения этого буфера, потом его распечатать, а потом опять повторять эту итерацию до полного исчерпания данных для печати. Можно, конечно, печатать посимвольно, что сильно бы все упростило, но у нас речь о хороших программах.

Названные выше источники оказались несколько хуже, чем ожидалось. В книге, например, некоторая информация оказалась вообще неверной и именно это подвинуло меня написать эту заметку. Неверной оказалась самая лёгкая для восприятия и использования информация, приведённая в виде схемы. Использование такой схемы может привести к серьёзным ошибкам. Хотя приведённый в книге пример работает правильно и следует именно этой схеме. Так получается из-за того, что в этом примере при обращении к /proc/iter печатается за один раз лишь несколько байт. Если же использовать его как шаблон для печати текстов, размером более страницы памяти, то возникнут сюрпризы. Упомянутая выше серия статей явных ошибок не содержит, но не сообщает о некоторых важные для понимания темы деталях.

Итак, рассмотрим сначала правильную схему того, как работают с файлом-последовательностью.



Для работы с таким файлом нужно создать функции start, stop, next и show. Названия этих функций могут быть любыми, выбрал наиболее короткие и соответствующие по смыслу. При наличие таких функций и их правильном подключении к системам ядра они начинают работать автоматически при обращении к связанному с ними файлу в каталоге /proc. Наиболее запутывающим выглядит использование функции stop, которая может вызываться в трёх разных контекстах. Её вызов после start означает окончание работы по распечатке данных. Её вызов после show означает, что при выполнении последней операции печати в буфер (обычно для этого используется функция seq_printf) случилось переполнение страничного буфера и эта операция печати была отменена. А её вызов после next – это самый интересный случай, который происходит, когда заканчивается печать в буфер одних данных и нужно либо завершать работу, либо использовать новые данные. Например, предположим, что наш файл в каталоге /proc при обращении к нему выдаёт сначала некоторую информацию по блочным устройствам, а затем по символьным. Сначала функция start инициализирует печать

по блочным устройствам, а функция next и, возможно, show используют данные этой инициализации для последующей печати шаг за шагом нужной информации по блочным устройствам. Когда всё будет готово, после последнего вызова next и будет сделан рассматриваемый случай вызова stop, после которого вызывается start, который на этот раз должен уже проинициализировать дальнейшую распечатку по символьным устройствам.

Приведу немного измененный пример (содержимое файла evens.c) из статьи Роба Дея. В ней пришлось заменить вызов функции, которая отсутствует в современных ядрах, на её актуальный эквивалент. Комментарии на английском сменил на на русском.

```
#include <linux/module.h>
#include <linux/moduleparam.h>
#include <linux/init.h>
#include <linux/kernel.h>
#include <linux/proc_fs.h>
#include <linux/fs.h>
#include <linux/seq_file.h>
#include <linux/slab.h>

static int limit = 10; //значение по умолчанию, его можно менять здесь
module_param(limit, int, S_IRUGO); //или, передавая параметр, при загрузке
модуля

static int* even_ptr; //усложняющий пример указатель, демонстрирующий, как
работать
        //с динамической памятью

/**
 * start
 */
static void *ct_seq_start(struct seq_file *s, loff_t *pos) {
     printk(KERN_INFO "Entering start(), pos = %Ld, seq-file pos = %lu.\n",
               *pos, s->count);

     if (*pos >= limit) {     // are we done?
         printk(KERN_INFO "Apparently, we're done.\n");
         return NULL;
     }
     //Выделяем память для числа, которое будет хранить наше растущее четное
значение
     even_ptr = kmalloc(sizeof(int), GFP_KERNEL);

     if (!even_ptr)     //фатальная ошибка
         return NULL;

     printk(KERN_INFO "In start(), even_ptr = %pX.\n", even_ptr);
     *even_ptr = (*pos)*2;
     return even_ptr;
}

/**
```

```
 * show
 */
static int ct_seq_show(struct seq_file *s, void *v) {
     printk(KERN_INFO "In show(), even = %d.\n", *(int*)v);
     seq_printf(s, "The current value of the even number is %d\n", *(int*)v);
     return 0;
}

/**
 * next
 */
static void *ct_seq_next(struct seq_file *s, void *v, loff_t *pos) {
     printk(KERN_INFO "In next(), v = %pX, pos = %Ld, seq-file pos = %lu.\n",
            v, *pos, s->count);

     (*pos)++;              //увеличиваем счётчик
     if (*pos >= limit)     //заканчиваем?
          return NULL;

     *(int*)v += 2;         //переходим к следующему чётному

     return v;
 }

/**
 * stop
 */
static void ct_seq_stop(struct seq_file *s, void *v) {
     printk(KERN_INFO "Entering stop().\n");

     if (v)
         printk(KERN_INFO "v is %pX.\n", v);
     else
         printk(KERN_INFO "v is null.\n");

     printk(KERN_INFO "In stop(), even_ptr = %pX.\n", even_ptr);

     if (even_ptr) {
         printk(KERN_INFO "Freeing and clearing even_ptr.\n");
         kfree(even_ptr);
         even_ptr = NULL;
     } else
         printk(KERN_INFO "even_ptr is already null.\n");
}

/**
 * Эта структура собирает функции для управления последовательным чтением
 */
static struct seq_operations ct_seq_ops = {
     .start = ct_seq_start,
     .next  = ct_seq_next,
     .stop  = ct_seq_stop,
     .show  = ct_seq_show
};

/**
 * Эта функция вызывается при открытии файла из /proc
 */
static int ct_open(struct inode *inode, struct file *file) {
     return seq_open(file, &ct_seq_ops);
```

```
};

/**
 * Эта структура собирает функции для управления файлом из /proc
 */
static struct file_operations ct_file_ops = {
     .owner   = THIS_MODULE,
     .open    = ct_open,
     .read    = seq_read,
     .llseek  = seq_lseek,
     .release = seq_release
};

/**
 * Эта функция вызывается, когда этот модуль загружается в ядро
 */
static int __init ct_init(void) {
     proc_create("evens", 0, NULL, &ct_file_ops);
     return 0;
}

/**
 * Эта функция вызывается, когда этот модуль удаляют из ядра
 */
static void __exit ct_exit(void) {
     remove_proc_entry("evens", NULL);
}

module_init(ct_init);
module_exit(ct_exit);

MODULE_LICENSE("GPL");
```

Функции для работы с файлом-последовательностей используют два указателя с пересекающейся функциональностью (это также несколько запутывающий момент). Один из них должен указывать на текущий объект для распечатки в буфер функцией show – v в программе, а другой (pos) обычно используется для указания на счётчик.

Для тех, кто возможно впервые захочет запустить свою программу в режиме ядра, привожу пример Makefile для успешной сборки. Конечно, для успешной сборки необходимо наличие в системе заголовков исходников ядра Linux.

```
obj-m += evens.o

all:
	make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules

clean:
	make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

Подключение к ядру производится командой sudo insmod evens.ko, проверка функциональности появившегося после этого файла /proc/evens командой cat /proc/evens, чтение журнала событий, поясняющих работу системы, командой sudo cat /var/log/messages.

Чтобы случилось переполнение страничного буфера нужно задать параметру limit значение побольше, например, 200. Это значение можно как вписать в текст программы, так и использовать при загрузке модуля – sudo insmod evens.ko limit=200.

Анализ лога может пояснить остающиеся неясными моменты. Например, можно заметить, что перед вызовом stop после next или start система зануляет v. Также можно заметить, что перед вызовом start после stop система распечатывает содержимое буфера.

Буду признателен, если кто-нибудь сообщит об обнаруженных в моей заметке неточностях или о том, что ещё следовало бы упомянуть.

Исходники можно также взять здесь.