

Глава 3. Файловая система /proc.

Файловая система `proc` представляет собой интерфейс к нескольким структурам данных ядра, которые работают также как и файловая система. Вместо того, чтобы каждый раз обращаться в `/dev/kmem` и искать путь к определению местонахождения какой-либо информации, все приложения читают файлы и каталоги из `/proc`. Таким образом все адреса структур данных ядра заносятся в `/proc` во время компиляции ядра, и программы использующие `proc` не могут перекомпилироваться после этого.

Существует возможность поддерживать файловую систему `proc` вне `/proc`, но при этом она теряет эффективность, поэтому в данном труде эта возможность не рассматривается.

3.1 Каталоги и файлы /proc.

Эта часть довольно сильно урезана, однако на данный момент авторы не могут предложить ничего более существенного.

В `/proc` существует подкаталог для каждого запускаемого процесса, названный по номеру `pid` процесса. Эти директории более подробно описаны ниже. Также в `/proc` присутствует несколько других каталогов и файлов:

`self` Этот файл имеет отношение к процессам имеющим доступ к файловой системе `proc`, и идентифицированным в директориях названных по `id` процессов осуществляющих контроль.

`kmsg` Этот файл используется системным вызовом `syslog()` для регистрации сообщений ядра. Чтение этого файла может осуществляться лишь одним процессом имеющим привилегию `superuser`. Этот файл не доступен для чтения при регистрации с помощью вызова `syslog()`.

`loadavg` Этот файл содержит числа подобно:

```
0.13 0.14 0.05
```

Эти числа являются результатом команд `uptime` и подобных, показывающих среднее число процессов пытающихся запуститься в одно и то же время за последнюю минуту, последние пять минут и последние пятнадцать.

`meminfo` Файл содержит обзор выходной информации программы `free`. Содержание его имеет следующий вид:

```
total:  used:  free:  shared:  buffers:
Mem: 7528448 7344128 184320 2637824 1949696
Swap: 8024064 1474560 6549504
```

Помните что данные числа представлены в байтах! Linus написала версию `free` осуществляющую вывод как в байтах, так и в кидобайтах в зависимости от ключа (`-b` или `-k`). Она находится в пакете `procs` в `tsx-11.mit.edu`. Также помните, что что своп-файлы используются нераздельно - все пространство памяти доступное для своппинга суммируется.

`uptime` Файл содержит время работы системы в целом и идеализированное время затрачиваемое системой на один процесс. Оба числа представлены в виде десятичных дробей с точностью до сотых секунды. Точность до двух цифр после запятой не гарантируется на всех архитектурах, однако на всех подпрограммах Linux даются достаточно точно используя удобные 100-Гц часы. Этот файл выглядит следующим образом: 604.33 205.45 В этом случае система функционирует 604.33 секунды, а время затрачиваемое на идеальный процесс равно 204.45 секунд.

`kcore` Этот файл представляет физическую память данной системы, в формате аналогичном "основному файлу" (`core file`). Он может быть использован отладчиком для проверки значений переменных ядра. Длина файла равна длине физической памяти плюс 4кб под заголовок.

`stat` Файл `stat` отображает статистику данной системы в формате ASCII. Пример:

```
cpu 5470 0 3764 193792
disk 0 0 0 0
page 11584 937
```

```
swap 255 618
intr 239978
ctxt 20932
btime 767808289
```

Значения строк:

cpu	Четыре числа сообщают о количестве тиков за время работы системы в пользовательском режиме, в пользовательском режиме с низким приоритетом, в системном режиме, и с идеальной задачей. Последнее число является стократным увеличением второго значения в файле uptime.
disk	Четыре компоненты dk_drive в структуре kernel_stat в данный момент незаняты.
page	Количество страниц введенных и исключенных системой.
swap	Количество своп-страниц введенных и исключенных системой.
intr	Количество прерываний установленных при загрузке системы.
ctxt	Номер подтекста выключающий систему.
btime	Время в секундах отсчитываемое сначала суток.
modules	Список модулей ядра в формате ASCII. Формат файла изменяется от версии к версии, поэтому пример здесь неприводится. Окончательно формат установится, видимо со стабилизацией интерфейса самих модулей.
malloc	Этот файл присутствует в случае, если во время компиляции ядра была описана строка CONFIG_DEBUG_MALLOC.
version	Файл содержит строку идентифицирующую версию работающего в данный момент Linux.
Linux version 1.1.40 (johnson@nigel) (gss version 2.5.8) #3 Sat Aug 6	Строка содержит версию Linux, имя пользователя и владельца осуществлявшего компиляцию ядра, версию gcc, количество предыдущих компиляций владельцем, дата последней компиляции.
net	Этот каталог содержит три файла, каждый из которых представляет статус части уровня работы с сетями в Linux. Эти файлы представляют двоичные структуры и они визуально нечитабельны, однако стандартный набор сетевых программ использует их. Двоичные структуры читаемые из этих файлов определены в . Файлы называются следующим образом:
unix	
arp	
route	
dev	
raw	
tcp	
udp	

- К сожалению, автор не располагает подробной информацией об устройстве файлов, поэтому в данной книге оно не описывается.

Каждый из подкаталогов прцессов (прнумерованных и имеющих собственный каталог) имеет свой набор файлов и подкаталогов. В подобном подкаталоге присутствует следующий набор файлов:

cmdlin	Содержит полную командную строку процесса, если он полностью не выгружен или убит. В любом из последних двух случаев файл пуст и чтение его приводит к тому-же результату, что и чтение пустой строки. Этот файл содержит в конце нулевой символ.
e	
cwd	Компановка текущего каталога данного процесса. Для обнаружения cwd процесса 20, сделайте следующее: (cd /proc/20/cwd; pwd)
environ	Файл содержит требования процесса. В файле отсутствуют переводы строки: в конце файла и между записями находятся нулевые символы. Для вывода требований процесса 10 вы должны сделать: cat /proc/10/environ tr "\000" "\n"
exe	Компановка запускаемого прцесса. Вы можете набрать: /proc/10/exe для перезапуска процесса 10 с любыми изменениями.
fd	Подкаталог содержащий запись каждого файла открытого процесса, названного именем дескриптора, и скомпанованного как фактический файл. Программы работающие с файлами, но не использующие

стандартный ввод-вывод, могут быть переопределены с использованием флагов -i (определение входного файла), -o (определение выходного файла): ... | foobar -i /proc/self/fd/0 -o /proc/self/fd/1 |... Помните, что это не будет работать в программах осуществляющих поиск файлов, так как файлы в каталоге fd поиску не поддаются.

maps Файл содержащий список распределенных кусков памяти, используемых процессом. Общедоступные библиотеки распределены в памяти таким образом, что на каждую из них отводится один отрезок памяти. Некоторые процессы также используют память для других целей.

Пример:

```
00000000 - 00013000 r-xs 00000400 03:03 12164
00013000 - 00014000 gwxr 00013400 03:03 12164
00014000 - 0001c000 gwxr 00000000 00:00 0
bffff000 - c0000000 gwxr 00000000 00:00 0
```

Первое поле записи определяет начало диапазона распределенного куска памяти.

Второе поле определяет конец диапазона отрезка.

Третье поле содержит флаги:

```
r - читаемый кусок, - нет.
w - записываемый, - нет.
x - запускаемый, - нет.
s - общедоступный, r - частного пользования.
```

Четвертое поле - смещение от которого происходит распределение.

Пятое поле отображает основной номер:подномер устройства распределяемого файла.

Пятое поле показывает число inode распределяемого файла.

me Этот файл не идентичен устройству mem, несмотря на то, что они имеют одинаковый номер устройств.

m Устройство /dev/mem - физическая память перед выполнением переадресации, здесь mem - память доступная процессу. В данный момент она не может быть перераспределена (mmap()), поскольку в ядре нет функции общего перераспределения.

root указатель на корневой каталог процесса. Полезен для программ использующих chroot(), таких как ftpd.

stat Файл содержит массу статусной информации о процессе. Здесь в порядке представления в файле описаны поля и их формат чтения функцией scanf():

```
pid %d    id процесса.
comm (%s)  Имя запускаемого файла в круглых скобках. Из него
            видно использует-ли процесс своппинг.
state %c   один из символов из набора "RSDZT", где:
            R - запуск
            S - заморозка в ожидании прерывания
            W - заморозка с запрещением прерывания (в частности
            для своппинга)
            Z - исключение процесса
            T - приостановка в определенном состоянии
ppid %d    pid процесса
pgrp %d    pgrp процесса
session %d
tty %d     используемая процессом tty.
tpgid %d   pgrp процесса который управляет tty соединенным
            с текущим процессом.
flags %u   Флаги процесса. Каждый флаг имеет набор битов
minflt %u  Количество малых сбоев работы процесса, которые не
            требуют загрузки с диска страницы памяти.
cminflt %u Количество малых сбоев в работе процесса и его сыновей
majflt %u  Количество существенных сбоев в работе процесса,
```

требующих подкачки страницы памяти.
 cmajflt %u Количество существенных сбоев процесса и его сыновей.
 utime %d Количество тиков, со времени распределения работы процесса в пространстве пользователя.
 stime %d Количество тиков, со времени распределения работы процесса в пространстве ядра.
 cutime %d Количество тиков, со времени распределения работы процесса и его сыновей в пространстве пользователя.
 cstime %d Количество тиков, со времени распределения работы процесса и его сыновей в пространстве ядра.
 counter %d Текущий максимальный размер в тиках следующего периода работы процесса, в случае его непосредственной деятельности, количество тиков до завершения деятельности.
 priority %d стандартное UN*X-е значение плюс пятнадцать. Это число не может быть отрицательным в ядре.
 timeout %u Время в тиках, следующего перерыва в работе процесса.
 it_real_value %u
 Период времени в тиках, по истечении которого процессу передается сигнал SIGALARM (будильник).
 start_time %d
 Время отсчитываемое от момента загрузки системы, по истечении которого начинает работу процесс.
 vsize %u Размер виртуальной памяти.
 rss %u Установленный размер резидентной памяти - количество страниц используемых процессом, содержащихся в реальной памяти минус три страницы занятые под управление. Сюда входят стековые страницы и информационные. Свop-страницы, страницы загрузки запросов не входят в данное число.
 rlim %u Предел размера процесса. По усмотрению 2Гб.
 start_code %u
 Адрес выше которого может выполняться текст программы.
 end_code %u Адрес ниже которого может выполняться текст программы.
 start_stack %u
 Адрес начала стека.
 kstk_esp %u Текущее значение указателя на 32-битный стек, получаемый в стековой странице ядра для процесса.
 kstk_eip %u Текущее значение указателя на 32-битную инструкцию, получаемую в стековой странице ядра для процесса.
 signal %d Побитовая таблица задержки сигналов (обычно 0)
 blocked %d Побитовая таблица блокируемых сигналов (обычно 0,2)
 sigignore %d
 Побитовая таблица игнорируемых сигналов.
 sigcatch %d Побитовая таблица полученных сигналов.
 wchan %u "Канал" в котором процесс находится в состоянии ожидания. Это адрес системного вызова, который можно посмотреть в списке имен, если вам нужно получить строковое значение имени.

statm Этот файл содержит специальную статусную информацию, занимающую немного больше места, нежели информация в stat, и используемую достаточно редко, чтобы выделить ее в отдельный файл. Для создания каждого поля в этом файле, файловая система rps должна просматривать каждый из 0x300 составляющих в каталоге страниц и вычислять их текущее состояние.

Описание полей:

size %d Общее число страниц, распределенное под процесс в виртуальной памяти, вне зависимости физическая она или логическая.
resident %d Общее число страниц физической памяти используемых процессом. Это поле должно быть численно равно полю rss в файле stat, однако метод подсчета значения отличается от примитивного чтения структуры процесса.
trs %d Размер текста в резидентной памяти - общее количество страниц текста(кода), принадлежащих

процессу, находящихся в области физической памяти. Не включает в себя страницы с общими библиотеками.

- lrs %d** Размер резидентной памяти выделенный под библиотеки - общее количество страниц, содержащих библиотеки, находящихся в верхней памяти.
- drs %d** Размер резидентной области используемой процессом в физической памяти.
- dt %d** Количество доступных страниц памяти.

3.2 Структура файловой системы /proc.

Файловая система proc интересна тем, что в реальной структуре каталогов не существует файлов. Функции, которые поводят гигантское количество операции по чтению файла, получению страницы и заполнению ее, выводу результата в пространство памяти пользователя, помещаются в определенные vfs-структуры.

Одним из интереснейших свойств файловой системы proc, является описание каталогов процессов. По существу, каждый каталог процесса имеет свой номер inode своего PID помещенный 16 бит в 32 - битный номер больше 0x0000ffff.

Внутри каталогов номер inode перезаписывается, так как верхние 16 бит номера маскируется выбором каталога.

Другим не менее интересным свойством, отличающим proc от других файловых систем в которых используется одна структура file_operations для всей файловой системы, введены различные структуры file_operations записываемые в компонент файловой структуры f_ops вбирающий в себя функции нужные для просмотра конкретного каталога или файла.

3.3 Программирование файловой системы /proc.

Предупреждение: Текст фрагментов программ, представленных здесь, может отличаться от исходников вашего ядра, так как файловая система /proc видоизменилась со времени создания этой книги, и видимо, будет видоизменяться далее. Структура root_dir со времени написания данного труда увеличилась вдвое.

В отличие от других файловых систем, в proc не все номера inode уникальны. Некоторые файлы определены в структурах

```
static struct proc_dir_entry root_dir[] = {
    { 1,1,"." },
    { 1,2,".." },
    { 2,7,"loadavg" },
    { 3,6,"uptime" },
    { 4,7,"meminfo" },
    { 5,4,"kmsg" },
    { 6,7,"version" },
    { 7,4,"self" }, /* смена номера inode */
    { 8,4,"net" }
};
```

Некоторые файлы динамически создаются во время чтения файловой системы. Все каталоги процесса имеют номера inode, чей идентификационный номер помещается в 16 бит, но файлы в этих каталогах переиспользуют малые номера inode (1-10), помещаемые во время работы процесса в pid процесса. Это происходит в inode.c с помощью аккуратного переопределения структур inode_operations.

Большинство файлов в корневом каталоге и в каждом подкаталоге процесса, доступных только для чтения используют простейший интерфейс поддерживаемый структурой array_inode_operations, находящейся в array.c.

Такие каталоги, как /proc/net, имеют свой номер inode. К примеру сам каталог net имеет номер 8. Файлы внутри этих каталогов имеют номера со 128 по 160, определенные в inode.c и для просмотра и записи таких файлов нужно специальное разрешение.

Внесение файла является несложной задачей, и остается в качестве упражнения читателю. Если предположить, что каталог в который вносится файл не динамический, как к примеру каталоги процессов, приведем следующий алгоритм:

1. Выберите уникальный диапазон номеров inode, дающий вам приемлимый участок памяти для помещения. Зытем справа от строки:
2. `if (!pid) { /* в каталоге /proc/ */`
3. `сделайте запись идентичную следующей`
4. `if ((ino>=128) && (ino<=160) { /*Файлы внутри /proc/net*/`
5. `inode->i_mode = S_IFREG | 0444`
6. `inode->i_op = &proc_net_inode_operations;`
7. `return;`
- `}`

изменив ее для операции нужной вам. В частности, если вы работаете в диапазоне 200-256 и ваши файлы имеют номера inode 200,201,202, ваши каталоги имеют номера 204 и 205, а номер inode 206 имеет имеющийся у вас файл читаемый лишь из корневого каталога, ваша запись будет выглядеть следующим образом:

```
if ((ino >= 200)&&(ino <= 256)) { /* Файлы в /proc/foo */
    switch (ino) {
        case 204:
        case 205:
            inode->i_mode = S_IFDIR | 0555;
            inode->i_op = &proc_foo_inode_operations;
            break;
        case 206:
            inode->i_mode = S_IFREG | 0400;
            inode->i_op = &proc_foo_inode_operations;
            break;
        default:
            inode->i_mode = S_IFREG | 0444;
            inode->i_op = &proc_foo_inode_operations;
            break;
    }
    return;
}
```

8. Найдите место определения файлов. Если ваши файлы помещаются в подкаталог каталога /proc, вам надо найти следующие строки в файле root.c:
9. `static struct proc_dir_entry root_dir[] = {`
10. `{ 1,1,"",`
11. `{ 1,2,"..",`
12. `{ 2,7,"loadavg" },`
13. `{ 3,6,"uptime" },`
14. `{ 4,7,"meminfo" },`
15. `{ 5,4,"kmsg" },`
16. `{ 6,7,"version" }`
17. `{ 7,4,"self" }, /* смена inode */`
18. `{ 8,4,"net" }`
- `};`

Затем вам следут подставить в эту запись после строки:

```
{ 8,4,"net" }
```

подставиь строку:

```
{ 9,3,"foo" }
```

Таким образом, вы предусматриваете новый каталог в inode.c, и текст:

```

if (!pid) { /* not a process directory but in /proc/ */
inode->i_mode = S_IFREG | 0444;
inode->i_op = &proc_array_inode_operations;
switch (ino)
case 5:
inode->i_op = &proc_array_inode_operations;
break;
case 8: /* for the net directory */
inode->i_mode = S_IFDIR | 0555;
inode->i_op = &proc_net_inode_operations;
break;
default:
break;
return;
}

```

СТАНОВИТСЯ

```

if (!pid) { /* not a process directory but in /proc/ */
inode->i_mode = S_IFREG | 0444;
inode->i_op = &proc_array_inode_operations;
switch (ino)
case 5:
inode->i_op = &proc_array_inode_operations;
break;
case 8: /* for the net directory */
inode->i_mode = S_IFDIR | 0555;
inode->i_op = &proc_net_inode_operations;
break;
case 9: /* for the foo directory */
inode->i_mode = S_IFDIR | 0555;
inode->i_op = &proc_foo_inode_operations;
break;
default:
break;
return;
}

```

19. Затем вам нужно обеспечить содержание файла в каталоге foo. Создайте файл `proc/foo.c` следуя указанной модели.

```

20. * linux/fs/proc/foo.c
21. *
22. * Copyright (C) 1993 Linus Torvalds, Michael K. Johnson, and Your N. Here
23. *
24. * proc foo directory handling functions
25. *
26. * inode numbers 200 - 256 are reserved for this directory
27. * (/proc/foo/ and its subdirectories)
28. */
29.
30. #include
31. #include
32. #include
33. #include
34. #include
35.
36. static int proc_readfoo(struct inode *, struct file *, struct dirent *, int);
37. static int proc_lookupfoo(struct inode *, const char *, int, struct inode **);
38. static int proc_read(struct inode * inode, struct file * file,
39. char * buf, int count),
40. static struct file_operations proc_foo_operations = {
41. NULL, /* lseek - default */
42. proc_read, /* read */

```

```

43.         NULL,          /* write - bad */
44.         proc_readfoo,    /* readdir */
45.         NULL,          /* select - default */
46.         NULL,          /* ioctl - default */ /* danlap */
47.         NULL,          /* mmap */
48.         NULL,          /* no special open code */
49.         NULL           /* no special release code */
50.     };
51.
52. /*
53.  * proc directories can do almost nothing..
54.  */
55. struct inode_operations proc_foo_inode_operations = {
56.     &proc_foo_operations, /* default foo directory file-ops */
57.     NULL,                /* create */
58.     proc_lookupfoo,      /* lookup */
59.     NULL,                /* link */
60.     NULL,                /* unlink */
61.     NULL,                /* symlink */
62.     NULL,                /* mkdir */
63.     NULL,                /* rmdir */
64.     NULL,                /* mknod */
65.
66.     NULL,                /* rename */
67.     NULL,                /* readlink */
68.     NULL,                /* follow_link */
69.     NULL,                /* bmap */
70.     NULL,                /* truncate */
71.     NULL                 /* permission */
72. };
73.
74. static struct proc_dir_entry foo_dir[] = {
75.     { 1,2,".." },
76.     { 9,1,"." },
77.     { 200,3,"bar" },
78.     { 201,4,"suds" },
79.     { 202,5,"xyzyzy" },
80.     { 203,3,"baz" },
81.     { 204,4,"dir1" },
82.     { 205,4,"dir2" },
83.     { 206,8,"rootfile" }
84. };
85.
86.
87. #define NR_FOO-DIRENTRY ((sizeof (foo_dir))/(sizeof (foo_dir[0])))
88.
89. unsigned int get_bar(char * buffer);
90. unsigned int get_suds(char * buffer);
91. unsigned int get_xyzyzy(char * buffer);
92. unsigned int get_baz(char * buffer);
93. unsigned int get_rootfile(char * buffer);
94.
95.
96. static int proc_read(struct inode * inode, struct file * file,
97.                     char * buf, int count)
98. {
99.     char * page;
100.    int length;
101.    int end;
102.    unsigned int ino;
103.
104.    if (count < 0)
105.        return -EINVAL;
106.    page = (char *) get_free_page(GFP-KERNEL);

```



```

107.     if (!page)
108.         return -ENOMEM;
109.     ino = inode->i_ino;
110.     switch (ino) {
111.         case 200:
112.             length = get_bar(page);
113.             break;
114.         case 201:
115.             length = get_suds(page);
116.             break;
117.         case 202:
118.             length = get_xyzzy(page);
119.             break;
120.         case 203:
121.             length = get_baz(page);
122.             break;
123.         case 206:
124.             length = get_rootfile(page);
125.             break;
126.         default:
127.             free_page((unsigned long) page);
128.             return -EBADF;
129.     }
130.     if (file->f_pos >= length) {
131.         free_page ((unsigned long) page);
132.         return 0;
133.     }
134.     if (count + file->t_pos > length)
135.         count = length - file->f_pos;
136.     end = count + file->f_pos;
137.     memcpy_tofs(buf, page + file->f_pos, count);
138.     free_page((unsigned long) page);
139.     file->f_pos = end;
140.     return count;
141.
142. }
143.
144. static int proc_ lookupfoo(struct inode * dir, const char * name, int len,
145.     struct inode ** result)
146. {
147.     unsigned int pid, ino;
148.     int i;
149.
150.     *result = NULL;
151.     if (!dir)
152.         return -ENOENT;
153.     if (!S_ISDIR(dir->i_mode)) {
154.         iput(dir);
155.         return -ENOENT;
156.     }
157.     ino = dir->i_ino;
158.     i = NR_FOO_DIRENTRY;
159.     while (i-- > 0 && !proc_match(len,name,foo_dir+i))
160.         /* nothing */;
161.     if (i < 0) {
162.         iput(dir);
163.         return -ENOENT;
164.     }
165.     if (!(*result = iget(dir->i_sb,ino))) {
166.         iput(dir);
167.         return -ENOENT;
168.     }
169.     iput(dir);
170.     return 0;

```

```

171.     }
172.
173. static int proc_readfoo(struct inode * inode, struct file * flie,
174.     struct dirent * dirent, int count)
175.
176.     {
177.         struct proc_dir_entry * de;
178.         unsigned int pid, ino;
179.         lnt i,j;
180.
181.         if (!inode || !S_ISDIR(inode->i_mode))
182.             return -EBADF;
183.         ino = inode->i_ino;
184.         if (((unsigned) filp->f_pos) < NR_FOO_DIRENTRY) {
185.             de = foo_dir + filp->f_pos;
186.             filp->f_pos++;
187.             i = de->namelen;
188.             ino = de->low_ino;
189.             put_fs_long(ino, &dirent->d_ino);
190.             put_fs_word(i, &dirent->d_reclen);
191.             put_fs_byte(0, i+dirent->d_name);
192.             j = i;
193.             while (i--)
194.                 put_fs_byte(de->name[i], i+dirent->d_name);
195.             return j;
196.         }
197.         return 0;
198.     }
199.
200.
201. unsigned int get_foo(char * buffer)
202.
203.     {
204.         /* code to find everything goes here */
205.
206.         return sprintf(buffer, "format string ", variables);
207.     }
208.
209.
210. unsigned int get_suds(char * buffer)
211.     {
212.         /* code to find everything goes here */
213.
214.         return sprintf(buffer, "format string", variables);
215.     }
216.
217. unsigned int get_xyzzy(char * buffer)
218.     {
219.         /* code to find everything goes here */
220.
221.         return sprintf(buffer, "format string", valriables);
222.     }
223.
224. unsigned int get_baz(char * buffer)
225.     {
226.         /* code to find everything goes here */
227.
228.         return sprintf(buffer, "format string", variables);
229.     }
230.
231. unsigned int get_rootfile(char * buffer)
232.     {
233.         /* code to find everything goes here */
234.

```

```
235.         return sprintf(buffer, "format string", variables);  
           }
```

Прмечание: Текст функций `proc_lookupfoo()` и `proc_readfoo()` абстрактный, так как они могут использоваться в разных местах.

236. Заполнение каталогов `dir1` и `dir2` остается в качестве упражнения. В большинстве случаев эти каталоги не используются, однако алгоритм представленный здесь может быть перестроен в рекурсивный алгоритм заполнения более глубоких каталогов. Заметим, что в программе сохранены номера `inode` с 200 по 256 для каталога `/proc/foo/` и всех его подкаталогов, так что вы можете использовать незанятые номера `inode` в этом диапазоне для ваших собственных файлов в `dir1` и `dir2`. Программа резервирует диапазон под каждый каталог для ваших будущих расширений. Автор также предпочел собрать всю информацию и требуемые функции в `foo.c` нежели создавать другой файл, если файлы не в `dir1` и в `dir2` не сильно концептуально отличаются от `foo`.
237. Сделайте соответствующие изменения в `fs/proc/имя_файла`. Это также будет достойным упражнением для читателя. Примечание: вышенаписанная программа (`/proc/net/netstat`) была написана по памяти и может оказаться неполной. Если вы обнаружите в ней какие-то несоответствия пожалуйста пришлите аннотацию по адресу johnsonm@sunsite.unc.edu.