|  |  |
| --- | --- |
|  | **Министерство науки и высшего образования Российской Федерации**  **Федеральное государственное бюджетное образовательное учреждение**  **высшего образования**  **«Московский государственный технический университет**  **имени Н.Э. Баумана**  **(национальный исследовательский университет)»**  **(МГТУ им. Н.Э. Баумана)** |

ФАКУЛЬТЕТ «Информатика и системы управления»

КАФЕДРА «Программное обеспечение ЭВМ и информационные технологии»

**ОТЧЕТ**

*к лабораторной работе №4*

*По курсу: «Операционные системы»*

*На тему: «Файловая система /proc»*

Студентка ИУ7-65Б

Оберган Т.М

Преподаватель

Рязанова Н.Ю.

*2020 г.*

Оглавление

[Часть 1 3](#_Toc38234409)

[/proc/[pid]/cmdline 3](#_Toc38234410)

[Листинг программы: 3](#_Toc38234411)

[Результат работы программы: 3](#_Toc38234412)

[/proc/[pid]/environ 4](#_Toc38234413)

[Листинг программы: 4](#_Toc38234414)

[Результат работы программы: 4](#_Toc38234415)

[/proc/[pid]/stat 6](#_Toc38234416)

[Листинг программы: 6](#_Toc38234417)

[Результат работы программы: 9](#_Toc38234418)

[/proc/[pid]/fd/ 10](#_Toc38234419)

[Листинг программы: 10](#_Toc38234420)

[Результат работы программы: 11](#_Toc38234421)

[Часть 2 11](#_Toc38234422)

[Загружаемый модуль ядра 11](#_Toc38234423)

[Листинг программы: 11](#_Toc38234424)

[Результат работы программы: 13](#_Toc38234425)

# Часть 1

## /proc/[pid]/cmdline

### Листинг программы:

#include <stdio.h>

#include <unistd.h>

#define BUFFSIZE 0x1000

**int** **main**(**int** argc, **char**\* argv[])

{

**char** buf[BUFFSIZE];

**int** len;

FILE\* f = fopen("/proc/self/cmdline","r");

len = fread(buf, 1, BUFFSIZE, f);

buf[len-1] = 0;

printf("pid: %d\ncmdline:%s\n", getpid(), buf);

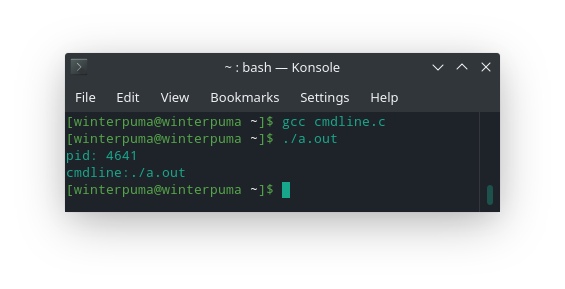
fclose(f);

**return** 0;

}

### Результат работы программы:

Содержит полную командную строку процесса. В случае, если процесс находится в состоянии зомби, файл пуст.



## /proc/[pid]/environ

### Листинг программы:

#include <stdio.h>

#define BUFFSIZE 0x1000

**int** **main**(**int** argc, **char**\* argv[])

{

**char** buf[BUFFSIZE];

**int** len;

**int** i;

FILE\* f;

f = fopen("/proc/self/environ","r");

**while**((len = fread(buf, 1, BUFFSIZE, f)) > 0)

{

**for**(i = 0; i < len; i++)

**if**(buf[i] == 0)

buf[i] = 10;

buf[len -1] = 10;

printf("%s", buf);

}

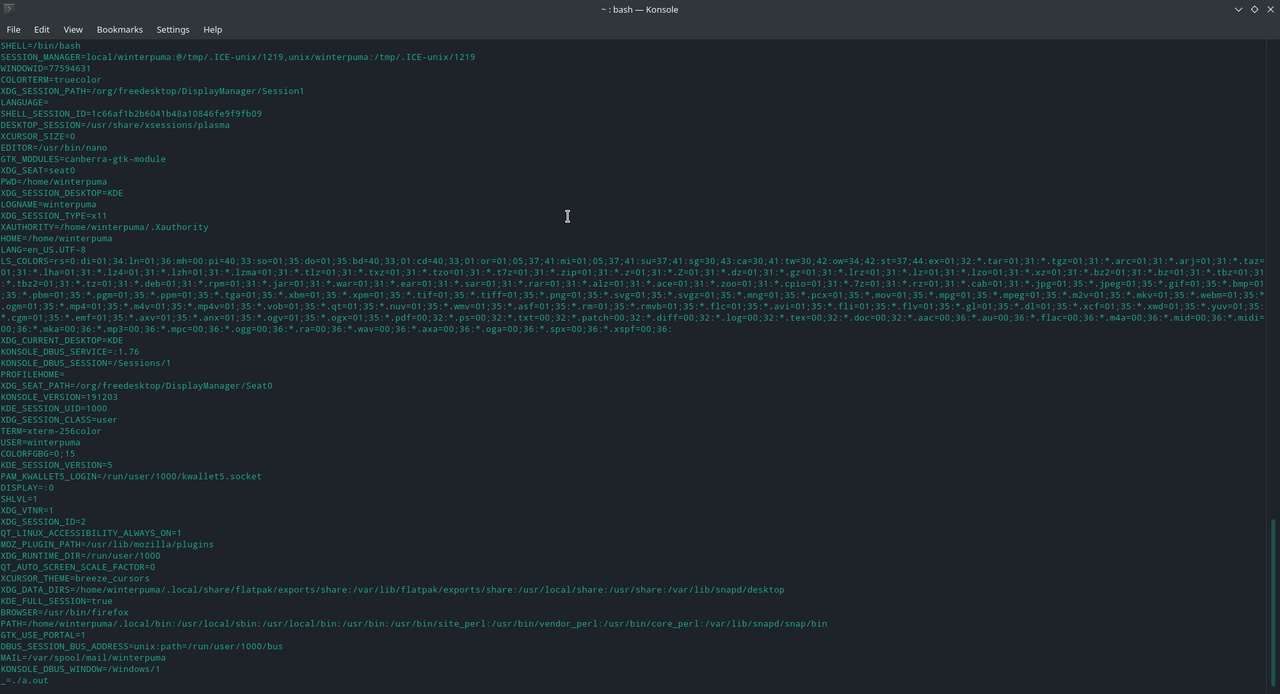
fclose(f);

**return** 0;

}

Окружение — это набор пар ПЕРЕМЕННАЯ=ЗНАЧЕНИЕ, доступный каждому пользовательскому процессу. Иными словами, окружение — это набор переменных окружения.

### Результат работы программы:



Данный файл содержит исходное окружение, которое было установлено при запуске текущего процесса (вызове execve()).

Некоторые переменные окружения:

* LS\_COLORS - используется для определения цветов, с которыми будут выведены имена файлов при вызове ls.
* LESSCLOSE, LESSOPEN – определяют пре- и пост- обработчики файла, который открывается при вызове less.
* XDG\_MENU\_PREFIX, XDG\_VTNR, XDG\_SESSION\_ID, XDG\_SESSION\_TYPE, XDG\_DATA\_DIRS, XDG\_SESSION\_DESKTOP, XDG\_CURRENT\_DESKTOP, XDG\_RUNTIME\_DIR, XDG\_CONFIG\_DIRS, DESKTOP\_SESSION – переменные, необходимые для вызова xdg-open, использующейся для открытия файла или URL в пользовательском приложении.
* LANG – язык и кодировка пользователя.
* DISPLAY – указывает приложениям, куда отобразить графический пользовательский интерфейс.
* GNOME\_SHELL\_SESSION\_MODE, GNOME\_TERMINAL\_SCREEN, GNOME\_DESKTOP\_SESSION\_ID, GNOME\_TERMINAL\_SERVICE, GJS\_DEBUG\_OUTPUT, GJS\_DEBUG\_TOPICS, GTK\_MODULES, GTK\_IM\_MODULE, VTE\_VERSION – переменные среды рабочего стола GNOME.
* COLORTERM – определяет поддержку 24-битного цвета.
* USER – имя пользователя, от чьего имени запущен процесс,
* USERNAME – имя пользователя, кто инициировал запуск процесса.
* SSH\_AUTH\_SOCK - путь к сокету, который агент использует для коммуникации с другими процессами.
* TEXTDOMAINDIR, TEXTDOMAIN – директория и имя объекта сообщения, получаемого при вызове gettext.
* PWD – путь к рабочей директории.
* HOME – путь к домашнему каталогу текущего пользователя.
* SSH\_AGENT\_PID - идентификатор процесса ssh-agent.
* TERM – тип запущенного терминала.
* SHELL – путь к предпочтительной оболочке командной строки.
* SHLVL – уровень текущей командной оболочки.
* LOGNAME – имя текущего пользователя.
* PATH - список каталогов, в которых система ищет исполняемые файлы.
* \_ - полная командная строка процесса
* OLDPWD - путь к предыдущему рабочему каталогу.

## /proc/[pid]/stat

### Листинг программы:

#include <stdio.h>

#include <string.h>

#define BUFFSIZE 0x1000

**int** **main**(**int** argc, **char**\*argv)

{

**char** buf[BUFFSIZE];

**int** n = 0;

FILE \*f;

f = fopen("/proc/self/stat","r");

fread(buf, 1, BUFFSIZE, f);

**char**\* p\_ch = strtok(buf, " ");

printf("\n stat: \n");

**while**(p\_ch != NULL)

{

n++;

printf("%d. %s \n", n, p\_ch);

p\_ch = strtok(NULL, " ");

}

fclose(f);

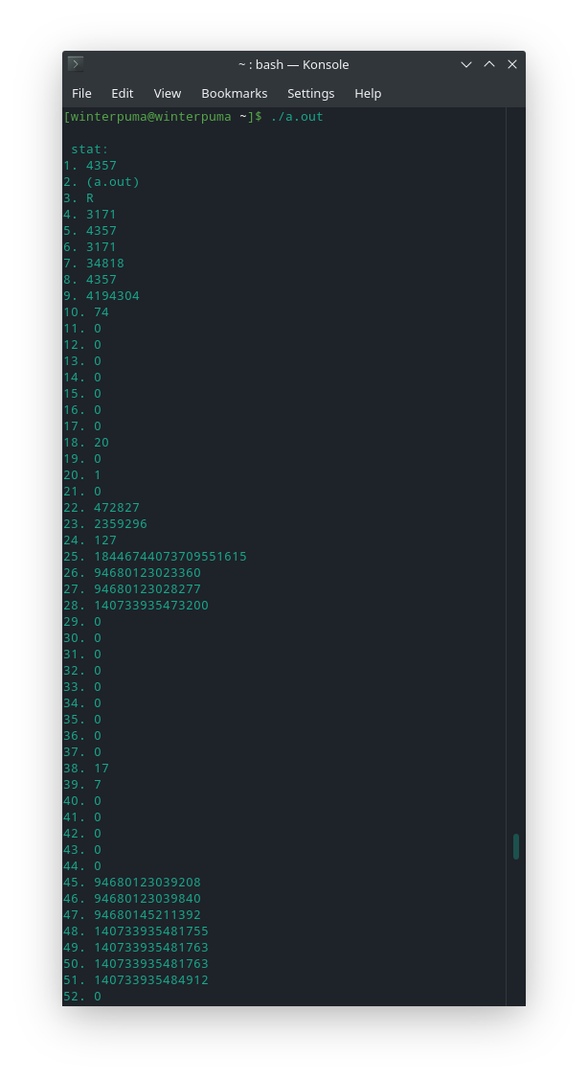
**return** 0;

}

Содержимое файла /proc/[pid]/stat:

1. pid - уникальный идентификатор процесса.
2. comm - имя исполняемого файла в круглых скобках.
3. state - состояние процесса.
4. ppid - уникальный идентификатор процесса-предка.
5. pgrp - уникальный идентификатор группы.
6. session - уникальный идентификатор сессии.
7. tty\_nr – управляющий терминал.
8. tpgid – уникальный идентификатор группы управляющего терминала.
9. flags – флаги.
10. minflt - Количество незначительных сбоев, которые возникли при выполнении процесса, и которые не требуют загрузки страницы памяти с диска.
11. cminflt - количество незначительных сбоев, которые возникли при ожидании окончания работы процессов-потомков.
12. majflt - количество значительных сбоев, которые возникли при работе процесса, и которые потребовали загрузки страницы памяти с диска.
13. cmajflt - количество значительных сбоев, которые возникли при ожидании окончания работы процессов-потомков.
14. utime - количество тиков, которые данный процесс провел в режиме пользователя.
15. stime - количество тиков, которые данный процесс провел в режиме ядра.
16. cutime - количество тиков, которые процесс, ожидающий завершения процессов-потомков, провёл в режиме пользователя.
17. cstime - количество тиков, которые процесс, ожидающий завершения процессов-потомков, провёл в режиме ядра.
18. priority – для процессов реального времени это отрицательный приоритет планирования минус один, то есть число в диапазоне от -2 до -100, соответствующее приоритетам в реальном времени от 1 до 99. Для остальных процессов это необработанное значение nice, представленное в ядре. Ядро хранит значения nice в виде чисел в диапазоне от 0 (высокий) до 39 (низкий), соответствующих видимому пользователю диапазону от -20 до 19.
19. nice - значение для nice в диапазоне от 19 (наиболее низкий приоритет) до   
    -20 (наивысший приоритет).
20. num\_threads – число потоков в данном процессе.
21. itrealvalue – количество мигов до того, как следующий SIGALARM будет послан процессу интервальным таймером. С ядра версии 2.6.17 больше не поддерживается и установлено в 0.
22. starttime - время в тиках запуска процесса после начальной загрузки системы.
23. vsize - размер виртуальной памяти в байтах.
24. rss - резидентный размер: количество страниц, которые занимает процесс в памяти. Это те страницы, которые заняты кодом, данными и пространством стека. Сюда не включаются страницы, которые не были загружены по требованию или которые находятся в своппинге.
25. rsslim - текущий лимит в байтах на резидентный размер процесса.
26. startcode - адрес, выше которого может выполняться код программы.
27. endcode - адрес, ниже которого может выполняться код программ.
28. startstack - адрес начала стека.
29. kstkesp - текущее значение ESP (указателя стека).
30. kstkeip - текущее значение EIP (указатель команд).
31. signal - битовая карта ожидающих сигналов. Устарела, потому что не предоставляет информацию о сигналах реального времени, необходимо использовать /proc/[pid]/status.
32. blocked - битовая карта блокируемых сигналов. Устарела, потому что не предоставляет информацию о сигналах реального времени, необходимо использовать /proc/[pid]/status.
33. sigignore - битовая карта игнорируемых сигналов. Устарела, потому что не предоставляет информацию о сигналах реального времени, необходимо использовать /proc/[pid]/status.
34. sigcatch - битовая карта перехватываемых сигналов. Устарела, потому что не предоставляет информацию о сигналах реального времени, необходимо использовать /proc/[pid]/status.
35. wchan - "канал", в котором ожидает процесс.
36. nswap - количество страниц на своппинге (не обслуживается).
37. сnswap - суммарное nswap для процессов-потомков (не обслуживается).
38. exit\_signal - сигнал, который будет послан предку, когда процесс завершится.
39. processor - номер процессора, на котором последний раз выполнялся процесс.
40. rt\_priority - приоритет планирования реального времени, число в диапазоне от 1 до 99 для процессов реального времени, 0 для остальных.
41. policy - политика планирования.
42. delayacct\_blkio\_ticks - суммарные задержки ввода/вывода в тиках.
43. guest\_time – гостевое время процесса (время, потраченное на выполнение виртуального процессора на гостевой операционной системе) в тиках.
44. cguest\_time - гостевое время для потомков процесса в тиках.
45. start\_data - адрес, выше которого размещаются инициализированные и неинициализированные (BSS) данные программы.
46. end\_data - адрес, ниже которого размещаются инициализированные и неинициализированные (BSS) данные программы.
47. start\_brk - адрес, выше которого куча программы может быть расширена с использованием brk().
48. arg\_start - адрес, выше которого размещаются аргументы командной строки (argv).
49. arg\_end - адрес, ниже которого размещаются аргументы командной строки (argv).
50. env\_start - адрес, выше которого размещается окружение программы.
51. env\_end - адрес, ниже которого размещается окружение программы.
52. exit\_code – статус завершения потока в форме, возвращаемой waitpid().

### Результат работы программы:



## /proc/[pid]/fd/

### Листинг программы:

#include <stdio.h>

#include <string.h>

#include <dirent.h>

#include <unistd.h>

#define BUFFSIZE 0x1000

**int** **main**(**int** argc, **char**\* argv)

{

**struct** dirent \*dirp;

DIR \*dp;

**char** string[BUFFSIZE];

**char** path[BUFFSIZE];

dp = opendir("/proc/self/fd"); // open directory

printf("\nfd:\n");

**while**((dirp = readdir(dp)) != NULL) // read directory

{

**if**((strcmp(dirp->d\_name, ".") !=0 ) &&

(strcmp(dirp->d\_name, "..") != 0))

{

sprintf(path, "%s%s", "/proc/self/fd/", dirp->d\_name);

readlink(path, string, BUFFSIZE);

path[BUFFSIZE] = '\0';

printf("%s -> %s\n", dirp->d\_name, string);

}

}

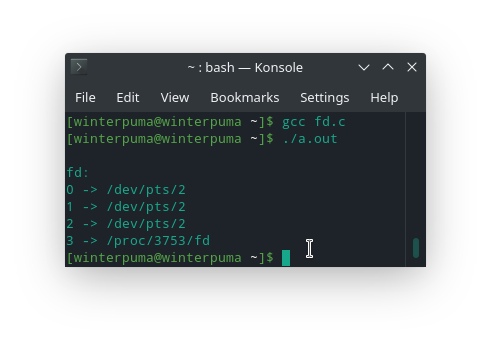
closedir(dp);

**return** 0;

}

Данная поддиректория содержит одну запись для каждого файла, который открыт процессом. Имя каждой такой записи соответствует номеру файлового дескриптора и является символьной ссылкой на реальный файл.

### Результат работы программы:



# Часть 2

## Загружаемый модуль ядра

Написать загружаемый модуль ядра, создать файл в файловой системе proc, sysmlink, subdir. Используя соответствующие функции передать данные из пространства пользователя в пространство ядра (введенные данные вывести в файл ядра) и из пространства ядра в пространство пользователя.

### Листинг программы:

#include <linux/module.h>

#include <init.h>

#include <linux/kernel.h>

#include <linux/proc\_fs.h>

#include <linux/string.h>

#include <linux/vmalloc.h>

#include <asm/uaccess.h>

#include <linux/uaccess.h>

#define COOKIE\_BUF\_SIZE PAGE\_SIZE

MODULE\_LICENSE("Dual BSD/GPL");

MODULE\_AUTHOR("Obergan T.M.");

**int** fortune\_init(**void**);

ssize\_t fortune\_read(**struct** file \*file, **char** \*buf, size\_t count, loff\_t \*f\_pos);

ssize\_t fortune\_write(**struct** file \*file, **const** **char** \*buf, size\_t count, loff\_t \*f\_pos);

**void** fortune\_exit(**void**);

**struct** file\_operations fops = {

.owner = THIS\_MODULE,

.read = fortune\_read,

.write = fortune\_write,

};

**char** \*cookie\_buf;

**struct** proc\_dir\_entry \*proc\_file;

**unsigned** **int** read\_index;

**unsigned** **int** write\_index;

**int** fortune\_init(**void**)

{

cookie\_buf = vmalloc(COOKIE\_BUF\_SIZE);

**if** (!cookie\_buf)

{

printk(KERN\_INFO "Error: can't malloc cookie buffer\n");

**return** -ENOMEM;

}

memset(cookie\_buf, 0, COOKIE\_BUF\_SIZE);

proc\_file = proc\_create("fortune", 0666, NULL, &fops);

**if** (!proc\_file)

{

vfree(cookie\_buf);

printk(KERN\_INFO "Error: can't create fortune file\n");

**return** -ENOMEM;

}

read\_index = 0;

write\_index = 0;

proc\_mkdir("Dir\_fort", NULL);

proc\_symlink("Symbolic\_fort", NULL, "/proc/fortune");

printk(KERN\_INFO "Fortune module loaded successfully\n");

**return** 0;

}

ssize\_t fortune\_read(**struct** file \*file, **char** \*buf, size\_t count, loff\_t \*f\_pos)

{

**int** len;

**if** (write\_index == 0 || \*f\_pos > 0)

**return** 0;

**if** (read\_index >= write\_index)

read\_index = 0;

len = sprintf(buf, "%s\n", &cookie\_buf[read\_index]);

read\_index += len;

\*f\_pos += len;

**return** len;

}

ssize\_t fortune\_write(**struct** file \*file, **const** **char** \*buf, size\_t count, loff\_t \*f\_pos)

{

**int** free\_space = (COOKIE\_BUF\_SIZE - write\_index) + 1;

**if** (count > free\_space)

{

printk(KERN\_INFO "Error: cookie pot is full\n");

**return** -ENOSPC;

}

**if** (copy\_from\_user(&cookie\_buf[write\_index], buf, count))

**return** -EFAULT;

write\_index += count;

cookie\_buf[write\_index-1] = 0;

**return** count;

}

**void** fortune\_exit(**void**)

{

remove\_proc\_entry("fortune", NULL);

**if** (cookie\_buf)

vfree(cookie\_buf);

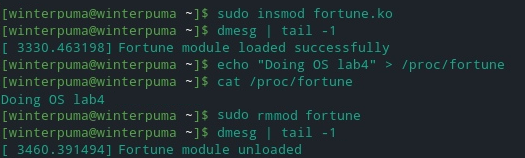
printk(KERN\_INFO "Fortune module unloaded\n");

}

module\_init(fortune\_init);

module\_exit(fortune\_exit);

### Результат работы программы:



Созданный файл:

****

Символьная ссылка:

****

Поддиректория в /proc:

****