

Введение в архитектуру ЭВМ

Юрий Анатольевич Богоявленский, заведующий кафедрой
Информатики и математического обеспечения, к.т.н., доцент:
ybgv@cs.petrSU.ru, <https://cs.petrSU.ru>

Веб страница дисциплины

<https://cs.petrSU.ru/~chistyak/architecture/>

Лекция 4. Системные вызовы. Файлы `stdin`, `stdout`. Разбор работы программы `task3`

План лекции

Системные вызовы.....	2
Системные вызовы. Общие положения.....	2
Порядок запуска системного вызова.....	4
Системный вызов <code>read</code>	5
Системный вызов <code>write</code>	5
Файлы <code>stdin</code> и <code>stdout</code> терминала <code>shell</code>	6
Взаимодействие буфера терминала <code>shell</code> и файла <code>stdin</code>	6
Введение в макропрограммирование в <code>gas</code>.....	7
Определения и термины.....	7
Схема подстановки.....	8
Разбор работы программы <code>task3</code>.....	9

Системные вызовы

Системные вызовы. Общие положения

Для обеспечения безопасности в развитых ОС оперативная память разделена на пространство пользователя и пространство ядра. Привилегии пользователя не позволяют выполнять операции в пространстве ядра. В ядре выполняются такие важные функции ОС как управление:

- памятью;
- устройствами;
- файловыми системами;
- процессами.

То прикладному программисту НЕ надо самому разрабатывать программы этих функций. Он может выполнять необходимые ему функции ядра, запуская выполнение в пространстве ядра специальных функций, т. н. системных вызовов (**syscalls**).

Системный вызов выполняется, когда пользовательский процесс (например **emacs**) требует некоторой функции, реализуемой ядром (например открытие файла), и вызывает для этого системный вызов **open**. В этот момент пользовательский процесс переводится в режим ожидания, ядро пытается выполнить системный вызов и передает результаты пользовательскому процессу, который затем возобновляет свою работу.

Системные вызовы в общем случае защищают доступ к ресурсам, которыми управляет ядро, при этом самые большие категории системных вызовов имеют дело с вводом/выводом (**open, close, read, write, poll**) и многие другие,

процессами (fork, execve, kill и т.д.), временем (time, gettimeofday и т.п.) и памятью (mmap, brk и пр.).

Многие команды языка shell просто обращаются к системным вызовам, например mkdir, kill.

Список всех доступных в конкретной unix системе вызовов находится в файлах /usr/include/asm/unistd_32.h для ISA IA-32 и в файле /usr/include/asm/unistd_64.h для ISA INTEL® 64. Приведем фрагмент файла.

```
#ifndef __ASM_X86_UNISTD_32_H
#define __ASM_X86_UNISTD_32_H 1

#define __NR_restart_syscall 0
#define __NR_exit 1
#define __NR_fork 2
#define __NR_read 3
#define __NR_write 4
#define __NR_open 5
#define __NR_close 6
#define __NR_waitpid 7
#define __NR_creat 8
#define __NR_link 9
#define __NR_unlink 10
#define __NR_execve 11
#define __NR_chdir 12
#define __NR_time 13
#define __NR_mknod 14
#define __NR_chmod 15
```

```
#define __NR_lchown 16
```

```
. . .
```

Имя системного вызова (но без префиксов) нужно использовать для получения его описания, например `man 2 read`. Функции системных вызовов описываются прототипами в стиле языка C.

Порядок запуска системного вызова

Пусть системный вызов с номером N описан прототипом:

```
syscall (p1, p2, p3, p4, p5)
```

Передача параметров системному вызову осуществляется через 32b регистры, в которые следует записать значения параметров, указанные в прототипе. Для запуска системного вызова необходимо выполнить команды:

```
mov N, %eax # задать номер системного вызова.
mov p1, %ebx # задать значение p1
mov p2, %ecx # ... p2
mov p3, %edx # ... p3
mov p4, %esi # ... p4
mov p5, %edi # ... p5
int $0x80    # прочитав из %eax номер системного
              # вызова и выполнить его
```

В таблице векторов прерываний ОС номер 0x80 содержит адрес программы обработки программного прерывания, обеспечивающего передачу управления программе системного вызова.

Команда `int` выполняет вызов программы обработчика прерываний номер которой задается ее операндом.

Соответственно по команде `int $0x80`:

1. Приостанавливается текущий выполняемый процесс;
2. Активируется защищенный режим;
3. Управление передается обработчику программных прерываний по системным вызовам;
4. Из `%eax` читается номер системного вызова;
5. Управление передается реализующей его функции;
6. Из регистров читаются параметры системного вызова и он выполняется.
7. Деактивируется защищенный режим
8. Управление возвращается прерванному для обработки прерывания процессу.

Системный вызов `read`

```
#include <unistd.h>
```

```
ssize_t read(int fd, void *buf, size_t count);
```

Номер системного вызова — **3**. `read()` пытается прочитывать не более `count` байтов из файла, имеющего файловый дескриптор `fd` в буфер в ОП, заданный указателем `*buf`.

Дескриптор файла это небольшое целое число, которое используется в системных вызовах работы с файлами для ссылки на конкретный файл.

Целое `fd` вычисляет и привязывает к файлу системный вызов `open` при открытии файла.

Если байты прочитались в буфер, в регистре `%eax` возвращается их количество. Если достигнут конец файла (**EOF**), то в регистре `%eax` возвращается значение ноль.

Системный вызов `write`

```
#include <unistd.h>
```

```
ssize_t write(int fd, const void *buf, size_t count);
```

Номер системного вызова — 4. `write()` записывает `count` байт из буфера в ОП, заданного указателем `*buf`, в файл, имеющий файловый дескриптор `fd`.

Если байты записались в файл, в регистре `%eax` возвращается их количество, при ошибке возвращается значение `-1`.

Файлы `stdin` и `stdout` терминала `shell`

В задаче `task3` организован диалог с текущим терминалом `shell` (клавиатура/экран) для которого в ОС системно назначены файлы и значения их дескрипторов:

- клавиатура — файл `stdin` (только для чтения), `fd = 0`;
- экран — файл `stdout`, (только для записи), `fd = 1`.

NBNB. при вводе с клавиатуры системное событие «конец файла» (EOF) для файла `stdin` формируется нажатием комбинации `ctrl/d` (не символ, не имеет кода). Это событие, которое обнаруживают системные вызовы при чтении файла.

Взаимодействие буфера терминала `shell` и файла `stdin`

При нажатии клавиш байты кодов соответствующих символов сначала попадают не в файл `stdin` а в буфер терминала. Только после нажатия клавиши `Enter` байты кодов символов всех клавиш, нажатых после предыдущего нажатия клавиши `Enter`, передаются в файл `stdin`. Системный вызов `read` прочтет из файла

заданное параметром `count` количество байтов и разместит их в ОП по адресу `$buf`.

Следующий запуск `read` будет читать еще не прочитанные байты, если они остались в файле `stdin`. Если байтов нет `read` будет ожидать их поступления в `stdin` после очередного нажатия клавиши `Enter`. Нажатие `Enter`, `ctrl/d`, `Enter` приводит к наступлению события `EOF` и закрытию файла `stdin` (он становится недоступным для операций).

Введение в макропрограммирование в `gas`

Программные системы имеют сложную структуру и содержат до десятков миллионов строк кода. Методы борьбы со сложностью опираются на подход, позволяющий уменьшить как объем, так и структурную сложность исходного кода.

Суть подхода — задание имени фрагмента кода и способа его модернизация для использования этого имени (от одной до нескольких строк) для выполнения кода модернизированного фрагмента без повторного размещения программистом его текста в исходном коде.

Этого подход лежит в основе трех различных концепций — *функции*, *объекта* и *макропрограммирования* реализации которых предоставляют инструменты для применения различных методов борьбы со сложностью.

Определения и термины

Макроопределение задает фрагмент кода, его имя и необязательные формальные аргументы для модернизации этого фрагмента при подстановках в исходный текст.

Макрокоманда это заданный макроопределением оператор языка программирования (в нашем случае ассемблера), вместо которого программа ассемблера подставляет макрорасширение.

Формат макрокоманды:

<метка> <имя> пробел/таб <список фактических аргументов>

Термин *аргумент* применяется при описании макрокоманд, а термин *операнд* для описания команд процессора.

Макрорасширение это подставленный в исходный текст вместо макрокоманды фрагмент кода, заданный макроопределением и модернизированный путем замены вхождений строк формальных аргументов в его тексте на строки фактических аргументов, указанные в макрокоманде.

Мы не будем использовать более поздний термин *макрос*, который в основном применяется для интерактивных макрокоманд, например а приложениях корпорации Microsoft, и на наш взгляд, плохо отражает суть дела.

Схема подстановки

Пусть макрокоманда `Exmple` с формальными аргументами `alpha beta` задана макроопределением как показано в левом столбце таблицы. Тогда макрокоманда:

`Example Day Night`

где `Example` — имя макрокоманды, а `Day Night` — фактические аргументы, будет в исходном текста заменена макрорасширением, показанном в правом столбце таблицы.

Макроопределение	Макрорасширение
<code>.macro Example alpha beta</code>	первая строка
первая строка	...
...	кодDayкод
код\alphaкод	...

<pre> ... код\betaкод ... последняя строка .endm </pre>	<pre> кодNightкод ... последняя строка </pre>
---	---

В макроопределении можно использовать псевдопеременную `\@` — счетчик количества выполненных макрокоманд, см. пример использования ниже в файле листинга в макроопределении `.macro Puts string`. В `gas` определены также директивы, позволяющие включать/исключать блоки кода в зависимости от условий.

Макропрограммирование:

- упрощает разработку и отладку;
- снижает дублирование кода;
- улучшает читаемость кода;
- обеспечивает повторное использование.
- позволяет формировать блоки кода в зависимости от платформы, архитектуры, версии компилятора.

Например, данные ниже в файле листинга макроопределения

`.macro Getchar buf_addr` и `.macro Puts string`, представляют собой «обертки» системных вызовов `read` и `write` и директив определения данных, которые позволяют упростить и автоматизировать их использование.

Разбор работы программы task3

Программа получает байты кодов символов с клавиатуры терминала и после нажатия клавиши `Enter` помещает их в файл стандартного ввода `stdin`, откуда эти байты системным вызовом `read` читаются в промежуточный буфер `C` для анализа.

Если прочитан код цифровой клавиши, то он помещается в очередной байт буфера `buf` в ОП. Коды остальных клавиш игнорируются. Программа завершает работу по нажатию комбинации `ctrl/D`.

Исходный текст.

```
/*
 * Программа ввода кодов цифровых символов в буфер в ОП
 */

#include "my-macro"

buf_length = 100

/*
    buf_length - постоянное сммвольное имя. Его значение равно
    выбранному кол-ву байтов в буфере и задается выражением времени
    ассемблирования. Позволяет избежать использования "магического
    числа" - числового литерала, например ниже в директиве .lcomm
*/

.bss
    .lcomm buf, buf_length # буфер из buf_length байтов
                           # для кодов прочитанных символов

    .lcomm c, 1 # однобайтовый буфер для чтения из файла stdin

.text
.global _start

_start:

    sub %esi, %esi # номер байта buf (%esi - индексный регистр)

show_prompt:
    Puts "Вводите цифру, друг мой!" # макрокоманда, подсказка

kbd_input:

    Getchar $c # макрокоманда чтения байта из stdin в буфер c

# Анализ события EOF

    cmpl $0, %eax # наступило событие EOF (конец файла stdin) ?

    je stop      # Да - на завершение программы, НЕТ - далее
```

Анализ кода байта в буфере с

```
cmpb $'\n', c    # это символ перевода строки ?
je kbd_input      # ДА - на ввод следующего символа
cmpb $'9', c      # код в с больше кода символа '9' ?
ja print_err_msg  # ДА - на вывод сообщения об ошибке
cmpb $'0', c      # код в с меньше кода символа '0' ?
jb print_err_msg  # ДА - на вывод сообщения об ошибке
movb c, %al       # передать код цифры из с в %al
movb %al, buf(%esi) # передать код цифры из al в байт
                    # буфера по адресу &buf + esi
incl %esi         # указать на следующий байт буфера для
                    # следующего кода
```

Лекционный комментарий.

Каждый введенный в цикле код цифрового символа надо записывать в последовательные байты `buf`, увеличивая в каждом цикле адрес записи на один.

Для решения таких задач существует механизм режимов адресации, частным случаем которого является команда `movb %al, buf(%esi)`.

При такой записи операнда приемника говорят, что `%esi` является базовым регистром, а адрес этого операнда приемника — `A` вычисляется перед каждым выполнением этой команды по формуле $A = \&buf + \%esi$, как это показано в таблице ниже. Увеличение значения `%esi` в каждом цикле обеспечивает команда `incl %esi`.

Адрес байта	buf+0	buf+1	buf+2	..	buf+n
Значение <code>%esi</code>	0	1	2		n
№ введенного байта кода	1	2	3		n+1

```
Puts "Цифра! Хорошо." # сообщения об успехе вводе
```

```

    jmp show_prompt # на ввод следующего символа

print_err_msg:

#    вывод сообщения об ошибке

    Puts "Не цифровая клавиша. Повторите ввод"

    jmp show_prompt # на ввод следующего символа

stop:

    Exit $0

.end

```

Т. о. использование макропрограммирования позволило получить короткую (81 строка), легко читаемую, изящную программу с использованием макрокоманд как «оберток» весьма сложных конструкций команд системных вызовов и директив определения данных.

Рассмотрим теперь файл листинга (419 строк), показывающий текст макроопределений и макрорасширений, которые были построены `gas` при ассемблировании.

GAS LISTING task3.S

page 1

```

1      /*
2      * Программа ввода кодов цифровых символов в буфер в ОП
3      */
4
5      .include "my-macro"
6
7      /*
8      Макроопределение завершения работы.
9      Аргументы:
10         - код завершения программы
11
12         После выполнения макрокоманды изменяются
13         регистры: %eax, %ebx
14         См. также 'man 2 exit'
15
16      */

```

```

13
14 .macro Exit ret_val
15
16     movl $1, %eax    # номер сист. вызова exit
17     movl \ret_val, %ebx # код выхода
18     int $0x80        # вызов обработчика
19                     # прерываний по системным вызовам
20 .endm
21
22 /*
23 Макроопределение для считывания одного
24 байта кода символа из файла стандартного
25 ввода stdin
26
27 Аргументы:
28     - адрес буфера для считывания байта
29
30 Результат:
31
32 Во время вссемблирования:
33
34     - формирует код сисвызова read для
35     чтения в буфер ОП по адресу
36     фактического аргумента одного байта
37
38 Во время выполнения:
39
40     - выполняет код сисвызова read, который:
41     если в stdin нет байтов - ожидает
42     если есть - читает 1 байт и пишет
43     его по адресу фактического аргумента
44
45 После выполнения макрокоманды изменяются
46 регистры: %eax, %ebx, %ecx, %edx
47
48 См. также 'man 2 read'
49 */
50
51 .macro Getchar buf_addr
52

```

```

53     movl $3, %eax    # номер сист. вызова read
54
55     movl $0, %ebx    # 1-й параметр: дескриптор stdin
56     movl \buf_addr, %ecx # 2-й параметр: адрес буфера,
57                     # задается фактически аргументом макро

```

```

58      movl $1, %edx      # 3-й параметр: сколько байт
59                          # для чтения в буфер
60
61      int $0x80          # вызов обработчика
программных
62                          # прерываний по системным
вызовам
63
64      .endm
65
66      /*
67      Макроопределение для вывода строки
68      в файл стандартного вывода stdout
69
70      Аргументы:
71      - Строковый литерал для вывода.
72
73      Пример макрокоманды:
74
75      Puts "Текст выводимой строки"
76
77      Результат:
78
79      Во время всемирлирования:
80
81      - Задаёт в секции данных строку аргумента
82      с меткой str\@: завершая её escape
83      последовательностью перевода строки \n
84
85      - вычисляет и присваивает символному имени
86      str\@ значение длины строки
87
88      - формирует код сис вызова write
89      для вывода строки в файл stdout
90
91      где:
92
93      \@ - псевдопеременная - счетчик количества
94      выполненных макрокоманд, после выполнения
95      любой макрокоманды увеличивается на 1.
96      Это обеспечивает уникальность имен строк
97      при многократном выполнении макроеоманд.
98
99      Во время выполнения:
100
101      - выполняет код сисвызова write для
102      вывода строки аргумента в файл stdout
103
104      После выполнения макрокоманды изменяются

```

```
105     регистры: %eax, %ebx, %ecx, %edx
106
107     См. также 'man puts', 'man 2 write'
108
109     */
```

GAS LISTING task3.S

page 3

```
110
111     .macro Puts string
112
113     .data
114     str\: .ascii "\string\n"  # строка для вывода
115
116     strlen\@ = . - str\@      # выражение времени
117     # ассемблирования вычисляющее значение длины строки
118
119     /*
120
121     Специальное символьное имя "." имеет значение
122     текущего адреса в секции куда размещаются при
123     ассемблировании коды данных и команд.
124     В каждой секции свой независимый от
125     других текущий адрес.
126
127     Здесь "." имеет значение адреса байта,
128     следующего за последним байтом строки,
129     размещенным директивой
130
131     Str\: .ascii "\string\n",
132
133     а метка str\: - значение адреса первого
134     байта этой строки.
135     Т.е. выражение
136
137     strlen\@ = . - str\@
138
139     дает искомую длину
140
141     */
142
143     .text
144
145     movl $4, %eax      # номер сист. вызова write
146     movl $1, %ebx      # дескриптор stdout
147     movl $str\@, %ecx   # адрес строки для вывода
148     movl $strlen\@, %edx # ол-во байт для вывода
```

```

143         int $0x80          # вызов обработчика программных
144         # прерываний по системным вызовам#
145     .endm
6
7     buf_length = 100
8
9     /*
10         buf_length - постоянное сммвольное имя. Его значение
11     равно выбранному кол-ву байтов в буфере и задается
    выражением ассемблирования.
    Это Позволяет избежать использования "магического
13     числа" - числового литерала, например ниже,
    в директиве bss
14
15     */
16
17     .bss
18         .lcomm buf, buf_length # буфер из buf_length байтов
19                                 # для кодов прочитанных
СИМВОЛОВ
20
21         .lcomm c, 1 # однобайтовый буфер для чтения из файла
stdin
22
23     .text
24     .global _start
25
26     _start:

```

GAS LISTING task3.S

page 4

```

27
28 0000 29F6          sub %esi, %esi # номер байта buf (%esi -
индексный регистр)
29
30     show_prompt:
31         Puts "Вводите цифру, друг мой!" # подсказка
31     >
31     > .data
31 0000 D092D0B2 > str0:.ascii "Вводите цифру, друг мой!\n"
31      D0BED0B4
31      D0B8D182
31      D0B520D1
31      86D0B8D1
31     >
31     > strlen0 =. - str0
31     >
31     > .text

```

```

31      >
31 0002 B8040000 > movl $4,%eax
31      00
31 0007 BB010000 > movl $1,%ebx
31      00
31 000c B9000000 > movl $str0,%ecx
31      00
31 0011 BA2C0000 > movl $strlen0,%edx
31      00
31      >
31 0016 CD80      > int $0x80
31      >
32
33     kbd_input:
34
35         Getchar $c # чтение байта из stdin в буфер c
35     >
35 0018 B8030000 > movl $3,%eax
35      00
35     >
35 001d BB000000 > movl $0,%ebx
35      00
35 0022 B9640000 > movl $c,%ecx

```

GAS LISTING task3.S

page 5

```

35      00
35     >
35 0027 BA010000 > movl $1,%edx
35      00
35     >
35     >
35 002c CD80      > int $0x80
35     >
35     >
36
37     #     Анализ события EOF
38
39 002e 83F800      cmpl $0, %eax # наступило событие EOF
(конец файла stdin) ?
40
41 0031 745A      je stop      # Да - на завершение
программы, НЕТ - далее
42
43     #     Анализ кода байта в буфере c
44
45

```

```

46 0033 803D6400      cmpb $'\n', c      # это символ перевода
строки ?
46          00000A
47
48 003a 74DC          je kbd_input      # ДА - на ввод следующего
символа
49
50 003c 803D6400      cmpb $'9', c      # код в c больше кода
символа '9' ?
50          000039
51
52 0043 772D          ja print_err_msg    # ДА - на вывод
сообщения об ошибке
53
54 0045 803D6400      cmpb $'0', c      # код в c меньше кода
символа '0' ?
54          000030
55
56 004c 7224          jb print_err_msg    # ДА - на вывод
сообщения об ошибке
57
58 004e A0640000      movb c, %al      # передать код цифры
из c в %al
58          00
59
60 0053 88860000      movb %al, buf(%esi) # передать код цифры
из al в байт
60          0000
61                      # буфера по адресу &buf + esi
62 0059 46            incl %esi      # указать на следующий
байт буфера для
63                      # следующего кода
64
65          Puts "Цифра! Хорошо." # сообщения об успехе вводе
65      >
65      > .data
65 002c D0A6D0B8      > str2:.ascii "Цифра! Хорошо.\n"
65          D184D180
65          D0B02120
65          D0A5D0BE
65          D180D0BE
65      >
65      > strlen2 =. - str2

```

```

65      > .text
65      >
65 005a B8040000      > movl $4,%eax

```

```

65      00      00
65 005f BB010000 > movl $1,%ebx
65      00
65 0064 B92C0000 > movl $str2,%ecx
65      00
65 0069 BA1A0000 > movl $strlen2,%edx
65      00
65 >
65 006e CD80 > int $0x80
65 >
66
67 0070 EB90 jmp show_prompt # на ввод следующего
СИМВОЛА
68
69 print_err_msg:
70
71 # вывод сообщения об ошибке
72
73 puts "Не цифровая клавиша. Повторите ввод"
73 >
73 > .data
73 0046 D09DD0B5 > str3:.ascii "Не цифровая клавиша.
Повторите ввод\n"
73      20D186D0
73      B8D184D1
73      80D0BED0
73      B2D0B0D1
73 >
73 > strlen3 =. - str3
73 >

```

GAS LISTING task3.S

page 7

```

73 > .text
73 >
73 0072 B8040000 > movl $4,%eax
73      00
73 0077 BB010000 > movl $1,%ebx
73      00
73 007c B9460000 > movl $str3,%ecx
73      00
73 0081 BA420000 > movl $strlen3,%edx
73      00
73 >
73 0086 CD80 > int $0x80
73 >
74
75 0088 E975FFFF jmp show_prompt # на ввод следующего
СИМВОЛА

```

```

75         FF
76
77     stop:
78
79         Exit $0
79     >
79 008d B8010000 > movl $1,%eax
79         00
79 0092 BB000000 > movl $0,%ebx
79         00
79 0097 CD80      > int $0x80
79     >
80
81     .end

```

GAS LISTING task3.S

page 8

DEFINED SYMBOLS

```

task3.S:7      *ABS*:000000064 buf_length
task3.S:18     .bss:000000000 buf
task3.S:18     .bss:000000064 c
task3.S:26     .text:000000000 _start
task3.S:30     .text:000000002 show_prompt
task3.S:31     .data:000000000 str0
task3.S:31     *ABS*:00000002c strlen0
task3.S:33     .text:000000018 kbd_input
task3.S:77     .text:00000008d stop
task3.S:69     .text:000000072 print_err_msg
task3.S:65     .data:00000002c str2
task3.S:65     *ABS*:00000001a strlen2
task3.S:73     .data:000000046 str3
task3.S:73     *ABS*:000000042 strlen3

```

NO UNDEFINED SYMBOLS

Выводы

- Для обеспечения безопасности прикладной программист может запускать в недоступном ему напрямую адресном пространстве ядра функции управления ОС - системные вызовы (`syscalls`).

- Функции системных вызовов описываются прототипами в стиле языка C.

- Передача параметров системному вызову осуществляется через 32b регистры, причем конкретный системный вызов задается его номером, помещаемым в `%eax`.

- Запуск системного вызова производится командой программного прерывания `int $0x80`.

- Для терминала `shell` (клавиатура/экран) в ОС системно назначены файлы и значения их дескрипторов:

- клавиатура — файл `stdin` (только для чтения), `fd = 0`;
- экран — файл `stdout`, (только для записи), `fd = 1`

- При нажатии клавиш байты кодов символов попадают в буфер терминала в ОП, а не в файл `stdin`, куда они передаются только после нажатия клавиши `Enter`.

- Макропрограммирование это одна из концепций борьбы со сложностью ПО.

- Макрокоманда при ассемблировании заменяется текстом макроопределения в котором выполнена подстановка текстов формальных аргументов макроопределения на тексты фактических аргументов макрокоманды.