**Введение в архитектуру ЭВМ**

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

**NB. Глубокое понимание этой программы является ключом к успешному выполнению всех лабораторных задач нашей дисциплины.**

## План лекции

- Общая схема работы программы.
- Системные вызовы. Общие положения
  - Порядок запуска системного вызова
  - Системный вызов read
  - Системный вызов write
- Файлы stdin и stdout терминала shell
- Взаимодействие клавиатуры терминала shell и файла stdin
- Макроопределения Getchar и Puts в файле my-macro
  - Макровызов и макрорасширение Getchar
  - Макровызов и макрорасширение Puts
- Разбор исходного файла task3.S
- Таблица символов из файла task3.lst

## Общая схема работы программы

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 оперативной памяти. Коды остальных клавиш игнорируются. Программа завершает работу по нажатию комбинации `ctrl/D`.**

**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.**

**Т.о прикладному программисту НЕ надо самому разрабатывать программы этих функций.**

**Прикладной программист может выполнять необходимые ему функции ядра, запуская выполнение в пространстве ядра специальных функций, т. н. *системных вызовов* (`syscalls`) с помощью приводимого ниже порядка запуска.**

**Системный вызов выполняется, когда пользовательский процесс (например `emacs`) требует некоторой службы, реализуемой ядром (например открытие файла), и вызывает специальную функцию (например, системный вызов `open`). В этот момент пользовательский процесс переводится в режим ожидания. Ядро анализирует запрос,**

**пытается его выполнить 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, settimeofday` и т.п.) и памятью (`mmap, brk` и пр.) Под эти категории подпадают практически все системные вызовы.**

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

**Список всех доступных в конкретной `unix` системе вызовов находится в файле `/usr/include/asm/unistd_32.h` для архитектуры IA-32 и в файле `/usr/include/asm/unistd_64.h` для архитектуры IA-64.**

```
#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:** `man read`**.2. Функции системных вызовов описываются прототипами в стиле языка** `C`**.**

**Пусть системный вызов с номером N описан прототипом**

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

**Передача параметров системному вызову осуществляется через 32-х битовые регистры, в которые следует записать значения параметров, указанные в прототипе, в порядке слева направо. Для запуска системного вызова необходимо выполнить следующую последовательность команд:**

```
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    # прочитать из %eax номер системного
             # вызова и выполнить его
```

**По команде** `int $0x80` **процессор приостанавливает текущий выполняемый процесс, переходит в защищенный режим, читает из** `%eax` **номер системного вызова и загружает для выполнения соответствующую функцию, реализующую системный вызов. Функция читает из регистров параметры и выполняется.**

**Системный вызов** `read`

```
#include <unistd.h>
ssize_t read(int fd, void *buf, size_t count);
```

**Номер системного вызова — 3.** `read()` **пытается прочитать 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 в регистре `%eax` возвращается значение `-1`.**

### Файлы stdin и stdout терминала shell

**NBNB. В задаче `task3` организован диалог с текущим терминалом `shell` (клавиатура/экран) для которого в ОС 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) stdin` формируется нажатием комбинации `ctrl/d` (не символ, не имеет кода). Это событие, которое обнаруживают системные вызовы при чтении файла.**

### Взаимодействие клавиатуры 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).**

### Макроопределения Getchar и Puts в файле my-macro

**Рассмотрим файл** my-macro

```
/*
 * Макроопределение завершения работы.
 * Аргументы:
 * - код завершения программы
 *
 * После выполнения макровызова изменяются регистры:
%eax, %ebx
 * См. также 'man 2 exit'
*/
.macro Exit ret_val
   movl $1, %eax          # номер сист. вызова exit
   movl \ret_val, %ebx    # код выхода
```

```
    int $0x80               # выполнить системный вызов
.endm


/*
 * Макроопределение для считывания одного байта кода
символа из стандартного ввода
 * Аргументы:
 * - адрес буфера для считывания байта
 * Результат:
 * - в %eax количество считанных байтов
 * - по адресу buf_addr - считанный байт
 *
 * После выполнения макровызова изменяются регистры:
 * %eax, %ebx, %ecx, %edx
 * См. также 'man 2 read'
*/
.macro Getchar buf_addr
    movl $3, %eax    # номер сист. вызова read
    movl $0, %ebx    # параметр 1: дескриптор
    стандартного ввода
    movl \buf_addr, %ecx    # параметр 2: адрес буфера
    (он же - фактический
                    # параметр макровызова)
    movl $1, %edx    # параметр 3: количество байтов
```

```
для чтения
    int $0x80        # выполнить системный вызов
.endm
```

**Макровызов и макрорасширение** `Getchar` **(фрагмент файла** `task3.lst`**)**

***Лекционный комментарий.***

**Появление в исходном файле макровызова** `Getchar $c` **приводит к включению в этот файл макрорасширения (команд тела макроопределения), в котором формальный параметр макроопределения** `buf_addr` **заменен на указанный в макровызове фактический параметр** `$c`**, - адрес однобайтового буфера, определенного в секции** `.bss`**.**

**Затем для этого буфера формируются правильные команды запуска системного вызова** `read`**.**

```
 22                     Getchar $c         # макровызов
ввода байта из stdin в
  22 0018 B8030000  >  movl $3,%eax
  22      00
  22 001d BB000000  >  movl $0,%ebx
  22      00
  22 0022 B9640000  >  movl $c,%ecx
  22      00
  22                >
  22 0027 BA010000  >  movl $1,%edx
  22      00

GAS LISTING task3.S             page 3
```

```
  22 002c CD80      >  int $0x80
  23                           # промежуточный буфер с
  24


/*
 * Макроопределение для  вывода строки в файл
стандартного вывода
 * Аргументы:
 * - Строка для вывода.
 *
 * Приметр макровызова:
 *  Puts "Текст выводимой строки"
 *
 * Результат:
 * - выводит в стандартный вывод символы заданной
строки
 *    и вслед за ними символ перевода строки \n
 *
 * После выполнения макровызова изменяются регистры:
%eax, %ebx, %ecx, %edx
 * См. также 'man puts', 'man 2 write'
*/
.macro Puts string
.data
    str\@:    .ascii "\string\n"  # формирование
```

```
фактической строки для вывода
    strlen\@ =   . - str\@      # получение значения
длины строки
.text
   movl $4, %eax    # номер сист. вызова write
   movl $1, %ebx    # параметр 1: дескриптор
стандартного вывода
   movl $str\@, %ecx       # параметр 2: адрес памяти с
выводимыми символами
   movl $strlen\@, %edx    # параметр 3: количество
байтов для вывода
   int $0x80         # выполнить системный вызов
.endm
```

**Макровызов и макрорасширение** `Puts` **(фрагмент файла** `task3.lst`**)**

***Лекционный комментарий.***

**Появление в исходном файле макровызова** `Puts "Цифра! Хорошо."` **приводит к включению в этот файл макрорасширения (команд тела макроопределения), в котором формальный параметр макроопределения** `string` **заменен на указанный в макровызове фактический параметр** `"Цифра! Хорошо."`**.При этом:**

**1) В секцию** `.data` **добавляется директива размещения в объектном файле кодов символов подлежащей выводу строки:**

`str2: .ascii "Цифра! Хорошо.\n"`

**с присоединением к ней условного кода символа перевода строки** `\n`.

2) **Метка этой строки — `str2:` формируется из конструкции `str\@`, где псевдопеременная `\@` представляют собой счетчик выполненных макровызовов (т. е. до выполнения рассматриваемого макровызова были выполнены макровызовы с номерами `0` и `1`).**

3) **Абсолютное символьное имя `strlen2`, также формируемое с помощью псевдопеременной `\@`, получает значение длины строки. Это символьное имя используется затем при формировании команд запуска системного вызова.**

4) **В секции `.text` формируются правильные команды запуска системного вызова write.**

```
 41                      Puts "Цифра! Хорошо." #
сообщения об успехе вводе
 41                   > .data
 41 002c D0A6D0B8     >  str2:.ascii "Цифра! Хорошо.\n"
 41      D184D180
 41      D0B02120
 41      D0A5D0BE
 41      D180D0BE
 41                   >
 41                   >  strlen2 =. - str2
 41                   >
 41                   > .text
 41 005a B8040000     >  movl $4,%eax
 41      00
 41 005f BB010000     >  movl $1,%ebx
 41      00
 41 0064 B92C0000     >  movl $str2,%ecx
 41      00
 41 0069 BA1A0000     >  movl $strlen2,%edx
```

```
  41       00
  41 006e CD80       >  int $0x80
```

**Разбор исходного файла task3.S**

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

.include "my-macro"

.bss
   .lcomm buf, 100         # 100 байтовый буфер для кодов прочитанных символов
   .lcomm c, 1             # однобайтовый буфер для чтения байта из файла stdin
```

***Лекционный комментарий.***

**Секция `.bss` используется для хранения неинициализированных данных, байты которых при запуске программы имеют нулевые значения. Данные этой секция не хранится в объектном и исполняемом файлах, память для нее распределяет загрузчик исполняемого файла. Символьные имена в секции `bss` определяются директивой `.lcomm`, задающей имя и длину выделяемой ему области ОП в байтах.**

```
.text
.global _start

_start:
   sub     %esi, %esi      # указатель адреса байта в буфере buf
(индексный регистр)

show_prompt:
   Puts "Вводите цифру, друг мой!"       # макровызов вывода строки в
                                         # файл stdout (подсказка ввода)
```

```
kbd_input:
   Getchar $c              # макровызов ввода байта из stdin в
                  # промежуточный буфер c

   cmpl $0, %eax  # наступило событие EOF (конец файла stdin) ?
   je stop                 # Да - на завершение программы

   cmpb $'\n', c           # это символ перевода строки ?
   je kbd_input            # ДА - на ввод следующего символа
   cmpb $'9', c            # код больше кода символа '9' ?
   ja print_err_msg        # ДА - на вывод сообщения об ошибке
   cmpb $'0', c            # код меньше кода символа '0' ?
   jb print_err_msg        # ДА - на вывод сообщения об ошибке

   movb c, %al             # передать код символа цифры из c в al
   movb %al, buf(%esi)       # передать код символа цифры из al в байт
                  # буфера по адресу &buf + esi
   incl %esi               # указать на следующий байт буфера для
                  # следующего кода
```

***Лекционный комментарий.***

**1. Т.к. оба буфера** `c` **и** `buf` **находятся в ОП, нельзя передать значение из** `c` **в** `buf` **одной командой, сперва передаем его в** `%al` **командой** `movb c, %al`.

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

| **Адрес байта** | `buf+0` | `buf+1` | `buf+2` | `..` | `buf+n` |
|---|---|---|---|---|---|
| **Значение** `%esi` | `0` | `1` | `2` | | `n` |
| **№ веденного байта кода** | `1` | `2` | `3` | | `n+1` |

**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 выше. Увеличение значения** `%esi` **в каждом цикле обеспечивает команда** `incl %esi`.

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

    jmp show_prompt # на ввод следующего символа

print_err_msg:
    Puts "Не цифровая клавиша. Повторите ввод"   #
вывод сообщения об ошибке
    jmp show_prompt    # на ввод следующего байта

stop:
    Exit $0

.end
```

**Таблица символов из файла task3.lst**

```
DEFINED SYMBOLS
      task3.S:8      .bss:0000000000000000 buf
      task3.S:8      .bss:0000000000000064 c
      task3.S:14     .text:0000000000000000 _start
      task3.S:17     .text:0000000000000002 show_prompt
      task3.S:18     .data:0000000000000000 str0
      task3.S:18     *ABS*:000000000000002c strlen0
      task3.S:21     .text:0000000000000018 kbd_input
      task3.S:49     .text:000000000000008d stop
      task3.S:45     .text:0000000000000072 print_err_msg
      task3.S:41     .data:000000000000002c str2
      task3.S:41     *ABS*:000000000000001a strlen2
      task3.S:46     .data:0000000000000046 str3
```

```
    task3.S:46     *ABS*:0000000000000042 strlen3

NO UNDEFINED SYMBOLS
```

**NBNB. Видно, что определяемые в макрорасширениях `Puts` строки последовательно располагаются в единственной секции `.data`.**